



Performance and Tuning Series: Query
Processing and Abstract Plans

Adaptive Server[®] Enterprise

15.7

DOCUMENT ID: DC00743-01-1570-01

LAST REVISED: September 2011

Copyright © 2011 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

IBM and Tivoli are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

CHAPTER 1	Understanding Query Processing	1
	Query optimizer	3
	Factors analyzed in optimizing queries	5
	Transformations for query optimization	6
	Handling search arguments and useful indexes	10
	Handling joins	12
	Optimization goals	15
	Limiting the time spent optimizing a query	16
	Parallelism	17
	Optimization issues	17
	Lava query execution engine	20
	Lava query plans	21
	How update operations are performed	27
	Direct updates	27
	Deferred updates	30
	Deferred index inserts	31
	Restrictions on update modes through joins	33
	Optimizing updates	34
	Using sp_sysmon while tuning updates	36
 CHAPTER 2	 Using showplan	 37
	Displaying a query plan	37
	Query plans in Adaptive Server Enterprise 15.0 and later	38
	Using set showplan with noexec	39
	Statement-level output	44
	Query plan shape	47
	Query plan operators	52
	EMIT operator	52
	SCAN operator	52
	FROM cache message	52
	FROM or LIST	53
	FROM TABLE	54
	Union operators	90
	UNION ALL operator	90

	MERGE UNION operator	91
	HASH UNION.....	91
	SCALAR AGGREGATE operator.....	93
	RESTRICT operator	94
	SORT operator.....	94
	STORE operator.....	95
	SEQUENCER operator	97
	REMOTE SCAN operator.....	99
	SCROLL operator.....	99
	RID JOIN operator.....	100
	SQLFILTER operator	102
	EXCHANGE operator	104
	INSTEAD-OF TRIGGER operators.....	106
	INSTEAD-OF TRIGGER operator.....	107
	CURSOR SCAN operator	108
	deferred_index and deferred_varcol messages	110
CHAPTER 3	Displaying Query Optimization Strategies and Estimates.....	111
	set commands for text format messages	111
	set commands for XML format messages.....	112
	Using show_execio_xml to diagnose query plans.....	114
	Showing cached plans in XML	116
	Diagnostic usage scenarios	119
	Permissions for set commands	122
	Analyzing dynamic parameters	122
	Dynamic parameter example analysis	123
CHAPTER 4	Finding Slow Running Queries	125
	Saving diagnostics to a trace file.....	125
	Set options that save diagnostic information to a trace file ...	127
	Which sessions are being traced?	128
	Rebinding a trace	129
	Displaying SQL text.....	129
	Retaining session settings	132
CHAPTER 5	Parallel Query Processing	133
	Vertical, horizontal, and pipelined parallelism	133
	Queries that benefit from parallel processing.....	134
	Enabling parallelism	135
	number of worker processes	135
	max parallel degree.....	136
	max resource granularity	136

max repartition degree	137
max scan parallel degree	138
prod-consumer overlap factor	138
min pages for parallel scan	138
max query parallel degree	139
Controlling parallelism at the session level	139
set command examples	140
Controlling query parallelism	141
Query-level parallel clause examples	141
Using parallelism selectively	141
Using parallelism with large numbers of partitions	143
When parallel query results differ	144
Queries that use set rowcount	145
Queries that set local variables	145
Understanding parallel query plans	146
Adaptive Server parallel query execution model	148
EXCHANGE operator	148
Using parallelism in SQL operations	154
Partition elimination	197
Partition skew	198
Why queries do not run in parallel	199
Runtime adjustment	199
Recognizing and managing runtime adjustments	200

CHAPTER 6	Eager and Lazy Aggregation	203
	Overview	203
	Eager aggregation	204
	Aggregation and query processing	205
	Examples	208
	Using eager aggregation	215
	Enabling eager aggregation	215
	Checking for eager aggregation	215
	Forcing eager aggregation with abstract plans	218

CHAPTER 7	Controlling Optimization	221
	Special optimizing techniques	221
	Viewing current optimizer settings	222
	Setting the optimization level	225
	Optimizer Diagnostic Utility	229
	Configuring Adaptive Server to run sp_opt_querystats	230
	Running sp_opt_querystats	230
	Specifying query processor choices	231
	Specifying table order in joins	232

Specifying the number of tables considered by the query processor..	233
Specifying query index	234
Specifying I/O size in a query	236
Index type and large I/O size.....	237
When prefetch specification cannot be followed	238
setting prefetch.....	239
Specifying cache strategy	239
In select, delete, and update statements.....	240
Controlling large I/O and cache strategies	241
Getting information on cache strategies	241
Asynchronous log service	241
Understanding the user log cache (ULC) architecture	243
When to use ALS	243
Using the ALS	244
Enabling and disabling merge joins	244
Enabling and disabling hash joins	245
Enabling and disabling join transitive closure	245
Controlling literal parameterization.....	246
Suggesting a degree of parallelism for a query	248
Query level parallel clause examples.....	250
Optimization goals.....	250
Setting optimization goals	251
Optimization criteria	252
Limiting optimization time	255
Controlling parallel optimization	256
number of worker processes	257
Specifying the number of worker processes available for parallel processing	257
max resource granularity	257
max repartition degree	258
Concurrency optimization for small tables	258
Changing the locking scheme	259

CHAPTER 8

Optimization for Cursors	261
Definition	261
Set-oriented versus row-oriented programming	262
Example	263
Resources required at each stage	264
Memory use and execute cursors	266
Cursor modes.....	266
Index use and requirements for cursors.....	267
Allpages-locked tables	267
Data-only-locked tables	268

Comparing performance with and without cursors	269
Sample stored procedure without a cursor	269
Sample stored procedure with a cursor	270
Cursor versus noncursor performance comparison	271
Locking with read-only cursors	272
Isolation levels and cursors	273
Partitioned heap tables and cursors	274
Optimizing tips for cursors	274
Optimizing for cursor selects using a cursor	275
Using union instead of or clauses or in lists	275
Declaring the cursor's intent	275
Specifying column names in the for update clause	276
Using set cursor rows	277
Keeping cursors open across commits and rollbacks	277
Opening multiple cursors on a single connection	278

CHAPTER 9

Query Processing Metrics	279
Overview	279
Executing QP metrics	280
Accessing metrics	280
sysquerymetrics view	280
Using metrics	282
Examples	283
Clearing metrics	284
Restricting query metrics capture	285
Understanding the UID in sysquerymetrics	286

CHAPTER 10

Using Statistics to Improve Performance	287
Statistics maintained in Adaptive Server	287
Importance of statistics	288
Nonbinary character set histogram interpolation	289
Updating statistics	290
Adding statistics for unindexed columns	290
Limitations for updating statistics on proxy tables and views	291
update statistics commands	291
Using sampling for update statistics	293
Automatically updating statistics	294
datachange function	295
Configuring automatic update statistics	297
Using Job Scheduler to update statistics	297
Examples of updating statistics with datachange	299
Column statistics and statistics maintenance	300
Creating and updating column statistics	302

	When additional statistics may be useful.....	303
	Adding statistics for a column with update statistics.....	305
	Adding statistics for minor columns with update index statistics ..	305
	Adding statistics for all columns with update all statistics.....	306
	Choosing step numbers for histograms.....	306
	Choosing a step number.....	307
	Scan types, sort requirements, and locking.....	307
	Sorts for unindexed or nonleading columns	308
	Locking, scans, and sorts during update index statistics.....	308
	Locking, scans and sorts during update all statistics.....	309
	Using the with consumers clause	309
	Reducing the impact of update statistics on concurrent processes	309
	Using the delete statistics command	310
	When row counts may be inaccurate	311
CHAPTER 11	Introduction to Abstract Plans.....	313
	Overview.....	313
	Managing abstract plans	314
	Relationship between query text and query plans.....	315
	Limits of options for influencing query plans.....	315
	Full versus partial plans.....	316
	Creating a partial plan.....	317
	Abstract plan groups.....	318
	How abstract plans are associated with queries	318
	Abstract plans in cached statements	319
CHAPTER 12	Creating and Using Abstract Plans	321
	Using set commands to capture and associate plans	321
	Enabling plan capture mode with set plan dump	322
	Associating queries with stored plans.....	323
	Using replace mode during plan capture	323
	Using dump, load, and replace modes simultaneously	324
	Compile-time changes for some set parameters	326
	set plan exists check option.....	327
	Using other set options with abstract plans	328
	Using show_abstract_plan to view plans.....	328
	Using showplan	329
	Using noexec.....	329
	Using fmtonly	330
	Using forceplan.....	330
	Server-wide abstract plan capture and association modes	330

	Creating plans using SQL.....	331
	Using create plan.....	331
	Using the plan clause	333
CHAPTER 13	Abstract Query Plan Guide.....	335
	Overview.....	335
	Abstract plan language	336
	Identifying tables.....	339
	Identifying indexes	341
	Specifying join order	341
	Specifying the join type.....	345
	Specifying partial plans and hints	346
	Creating abstract plans for subqueries	349
	Abstract plans for materialized processing of views	356
	Abstract plans for queries containing aggregates.....	357
	Abstract plans for queries containing unions	358
	Using abstract plans when queries need ordering.....	359
	Specifying the reformatting strategy	360
	Specifying the OR strategy	361
	When the store operator is not specified	361
	Abstract plans for parallel processing.....	361
	Tips on writing abstract plans	363
	Using abstract plans at the query level.....	363
	Operator name alignment for abstract plan and optimizer criteria	
	365	
	Extending the optimizer criteria set syntax	366
	Comparing plans before and after	366
	Effects of enabling server-wide capture mode.....	367
	Time and space to copy plans	368
	Abstract plans for stored procedures.....	368
	Procedures and plan ownership	369
	Procedures with variable execution paths and optimization ..	369
	Ad hoc queries and abstract plans	370
CHAPTER 14	Managing Abstract Plans with System Procedures.....	373
	Managing an abstract plan group	373
	Creating a group	373
	Dropping a group	374
	Getting information about a group	374
	Renaming a group	376
	Finding abstract plans	377
	Managing individual abstract plans	377
	Viewing a plan	378

Copying a plan to another group	378
Dropping an individual abstract plan.....	379
Comparing two abstract plans	379
Changing an existing plan	380
Managing all plans in a group.....	381
Copying all plans in a group	381
Comparing all plans in a group	382
Dropping all abstract plans in a group	384
Importing and exporting groups of plans	385
Exporting plans to a user table	385
Importing plans from a user table	385
Index.....	387

Understanding Query Processing

This chapter provides an overview of the query processor in Adaptive Server[®] Enterprise.

Topic	Page
Query optimizer	3
Optimization goals	15
Parallelism	17
Optimization issues	17
Lava query execution engine	20

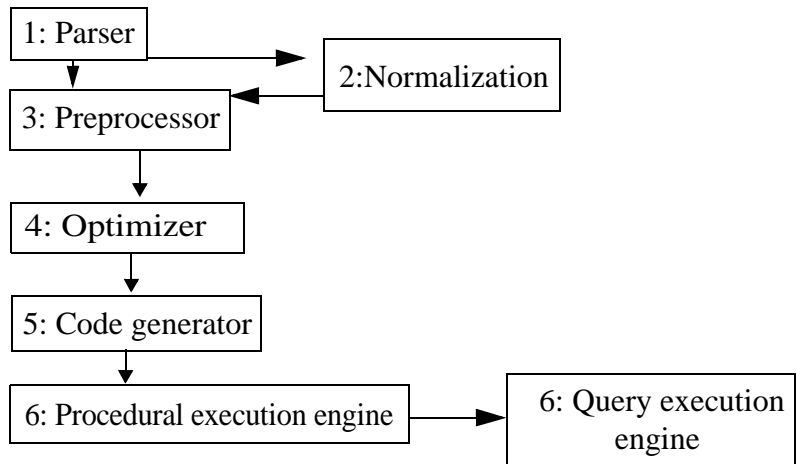
The query processor is designed to process queries you specify. The processor yields highly efficient query plans that execute using minimal resources, and ensure that results are consistent and correct.

To process a query efficiently, the query processor uses:

- The specified query
- Statistics about the tables, indexes, and columns named in the query
- Configurable variables

The query processor has to execute several steps, using several modules, to successfully process a query:

Figure 1-1: : Query processor modules



- 1 The parser converts the text of the SQL statement to an internal representation called a query tree.
- 2 This query tree is normalized. This involves determining column and table names, transforming the query tree into conjugate normal form (CNF), and resolving datatypes. At this point, you can determine if the statement may benefit from using the statement cache.
- 3 The preprocessor transforms the query tree for some types of SQL statements, such as SQL statements with subqueries and views, to a more efficient query tree.
- 4 The optimizer analyzes the possible combinations of operations (join ordering, access and join methods, parallelism) to execute the SQL statement, and selects an efficient one based on the cost estimates of the alternatives.
- 5 The code generator converts the query plan generated by the optimizer into a format more suitable for the query execution engine.
- 6 The procedural engine executes command statements such as create table, execute procedure, and declare cursor directly. For data manipulation language (DML) statements, such as select, insert, delete, and update, the engine sets up the execution environment for all query plans and calls the query execution engine.
- 7 The query execution engine executes the ordered steps specified in the query plan provided by the code generator.

Query optimizer

The query optimizer provides speed and efficiency for online transaction processing (OLTP) and operational decision-support systems (DSS) environments. You can choose an optimization strategy that best suits your query environment.

The query optimizer is self-tuning, and requires fewer interventions than versions of Adaptive Server Enterprise earlier than 15.0. It relies infrequently on worktables for materialization between steps of operations; however, the query optimizer may use more worktables when it determines that hash and merge operations are more effective.

Some of the key features in the release 15.0 query optimizer include support for:

- New optimization techniques and query execution operator supports that enhance query performance, such as:
 - On-the-fly grouping and ordering operator support using in-memory sorting and hashing for queries with group by and order by clauses
 - hash and merge join operator support for efficient join operations
 - index union and index intersection strategies for queries with predicates on different indexes

Table 1-1 on page 4 is a list of optimization techniques and operator support provided in Adaptive Server Enterprise. Many of these techniques map directly to the operators supported in the query execution. See “Lava query execution engine” on page 20.

- Improved index selection, especially for joins with or clauses, and joins with and search arguments (SARGs) with mismatched but compatible datatypes
- Improved costing that employs join histograms to prevent inaccuracies that might otherwise arise due to data skews in joining columns
- New cost-based pruning and timeout mechanisms in join ordering and plan strategies for large, multiway joins, and for star and snowflake schema joins
- New optimization techniques to support data and index partitioning (building blocks for parallelism) that are especially beneficial for very large data sets
- Improved query optimization techniques for vertical and horizontal parallelism. See Chapter 5, “Parallel Query Processing.”

- Improved problem diagnosis and resolution through:
 - Searchable XML format trace outputs
 - Detailed diagnostic output from new set commands. See Chapter 3, “Displaying Query Optimization Strategies and Estimates.”

Table 1-1: Optimization operator support

operator	Description
hash join	Determines whether the query optimizer may use the hash join algorithm. hash join may consume more runtime resources, but is valuable when the joining columns do not have useful indexes or when a relatively large number of rows satisfy the join condition, compared to the product of the number of rows in the joined tables.
hash union distinct	Determines whether the query optimizer may use the hash union distinct algorithm, which is inefficient if most rows are distinct.
merge join	Determines whether the query optimizer may use the merge join algorithm, which relies on ordered input. merge join is most valuable when input is ordered on the merge key, for example, from an index scan. merge join is less valuable if sort operators are required to order input.
merge union all	Determines whether the query optimizer may use the merge algorithm for union all. merge union all maintains the ordering of the result rows from the union input. merge union all is particularly valuable if the input is ordered and a parent operator (such as merge join) benefits from that ordering. Otherwise, merge union all may require sort operators that reduce efficiency.
merge union distinct	Determines whether the query optimizer may use the merge algorithm for union. merge union distinct is similar to merge union all, except that duplicate rows are not retained. merge union distinct requires ordered input and provides ordered output.
nested-loop-join	The nested-loop-join algorithm is the most common type of join method and is most useful in simple OLTP queries that do not require ordering.
append union all	Determines whether the query optimizer may use the append algorithm for union all.
distinct hashing	Determines whether the query optimizer may use a hashing algorithm to eliminate duplicates, which is very efficient when there are few distinct values compared to the number of rows.
distinct sorted	Determines whether the query optimizer may use a single-pass algorithm to eliminate duplicates. distinct sorted relies on an ordered input stream, and may increase the number of sort operators if its input is not ordered.
group-sorted	Determines whether the query optimizer may use an on-the-fly grouping algorithm. group-sorted relies on an input stream sorted on the grouping columns, and it preserves this ordering in its output.
distinct sorting	Determines whether the query optimizer may use the sorting algorithm to eliminate duplicates. distinct sorting is useful when the input is not ordered (for example, if there is no index) and the output ordering generated by the sorting algorithm could benefit; for example, in a merge join.

operator	Description
group hashing	Determines whether the query optimizer may use a group hashing algorithm to process aggregates.
Technique	Description
multi table store ind	Determines whether the query optimizer may use reformatting on the result of a multiple table join. Using multi table store ind may increase the use of worktables.
opportunistic distinct view	Determines whether the query optimizer may use a more flexible algorithm when enforcing distinctness.
index intersection	Determines whether the query optimizer may use the intersection of multiple index scans as part of the query plan in the search space.

Factors analyzed in optimizing queries

Query plans consist of retrieval tactics and an ordered set of execution steps, which retrieve the data needed by the query. In developing query plans, the query optimizer examines:

- The size of each table in the query, both in rows and data pages, and the number of OAM and allocation pages to be read.
- The indexes that exist on the tables and columns used in the query, the type of index, and the height, number of leaf pages, and cluster ratios for each index.
- The index coverage of the query; that is, whether the query can be satisfied by retrieving data from the index leaf pages without accessing the data pages. Adaptive Server can use indexes that cover queries, even if no where clauses are included in the query.
- The density and distribution of keys in the indexes.
- The size of the available data cache or caches, the size of I/O supported by the caches, and the cache strategy to be used.
- The cost of physical and logical reads; that is, reads of physical I/O pages from the disk, and of logical I/O reads from main memory.
- join clauses, with the best join order and join type, considering the costs and number of scans required for each join and the usefulness of indexes in limiting the I/O.

- Whether building a worktable (an internal, temporary table) with an index on the join columns is faster than repeated table scans if there are no useful indexes for the inner table in a join.
- Whether the query contains a max or min aggregate that can use an index to find the value without scanning the table.
- Whether data or index pages must be used repeatedly, to satisfy a query such as a join, or whether a fetch-and-discard strategy can be employed because the pages need to be scanned only once.

For each plan, the query optimizer determines the total cost by computing the costs of logical and physical I/Os, and CPU processing. If there are proxy tables, additional network related costs are evaluated as well. The query optimizer then selects the cheapest plan.

The query processor for Adaptive Server versions 15.0.2 and later defers the optimization of statements in a stored procedure until it executes the statement. This benefits the query processor because the values for local variables are available for optimization for their respective statements.

Earlier versions of Adaptive Server used default guesses for selectivity estimates on predicates using local variables.

Transformations for query optimization

After a query is parsed and preprocessed, but before the query optimizer begins its plan analysis, the query is transformed to increase the number of clauses that can be optimized. The transformation changes made by the optimizer are transparent unless the output of such query tuning tools as showplan, dbcc(200), statistics io, or the set commands is examined. If you run queries that benefit from the addition of optimized search arguments, the added clauses are visible. In showplan output, it appears as “Keys are” messages for tables for which you specify no search argument or join.

Search arguments converted to equivalent arguments

The optimizer looks for query clauses to convert to the form used for search arguments. These are listed in Table 1-2.

Table 1-2: Search argument equivalents

Clause	Conversion
between	Converted to \geq and \leq clauses. For example, between 10 and 20 is converted to ≥ 10 and ≤ 20 .
like	<p>If the first character in the pattern is a constant, like clauses may be converted to greater than or less than queries. For example, like "sm%" becomes \geq "sm" and $<$ "sn".</p> <p>If the first character is a wildcard, a clause such as like "%x" cannot use an index for access, but histogram values can be used to estimate the number of matching rows.</p>
in(values_list)	Converted to a list of or queries, that is, <code>int_col in (1, 2, 3)</code> becomes <code>int_col = 1 or int_col = 2 or int_col = 3</code> . The maximum number of elements in an in-list is 1025

Search argument transitive closure applied where applicable

The optimizer applies transitive closure to search arguments. For example, the following query joins titles and titleauthor on title_id and includes a search argument on titles.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
      and a.au_id = ta.au_id
      and t.title_id = "T81002"
```

This query is optimized as if it also included the search argument on titleauthor.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
      and a.au_id = ta.au_id
      and t.title_id = "T81002"
      and ta.title_id = "T81002"
```

With this additional clause, the query optimizer can use index statistics on titles.title_id to estimate the number of matching rows in the titleauthor table. The more accurate cost estimates improve index and join order selection.

equi-join predicate transitive closure applied where applicable

The optimizer applies transitive closure to join columns for a normal equi-join. The following query specifies the equi-join of t1.c11 and t2.c21, and the equi-join of t2.c21 and t3.c31:

```
select *
from t1, t2, t3
where t1.c11 = t2.c21
      and t2.c21 = t3.c31
      and t3.c31 = 1
```

Without join transitive closure, the only join orders considered are (t1, t2, t3), (t2, t1, t3), (t2, t3, t1), and (t3, t2, t1). By adding the join on t1.c11 = t3.c31, the query processor expands the list of join orders to include: (t1, t3, t2) and (t3, t1, t2). Search argument transitive closure applies the condition specified by t3.c31 = 1 to the join columns of t1 and t2.

Similarly, equi-join transitive closure is also applied to equi-joins with or predicates as follows:

```
select *
from R, S
where R.a = S.a
      and (R.a = 5 OR S.b = 6)
```

The query optimizer infers that this would be equivalent to:

```
select *
from R, S
where R.a = S.a
      and (S.a = 5 or S.b = 6)
```

The or predicate could be evaluated on the scan of S and possibly be used for an optimization, thereby effectively using the indexes of S.

Another example of join transitive closure is its application to complex SARGs, so that a query such as:

```
select *
from R, S
where R.a = S.a and (R.a + S.b = 6)
```

is transformed and inferred as:

```
select *
from R, S
where R.a = S.a
      and (S.a + S.b = 6)
```

The complex predicate could be evaluated on the scan of S, resulting in significant performance improvements due to early result set filtering.

Transitive closure is used only for normal equi-joins, as shown. join transitive closure is not performed for:

- Non-equi-joins; for example, t1.c1 > t2.c2

- Outer joins; for example, `t1.c11 *= t2.c2`, or left join or right join
- Joins across subquery boundaries
- Joins used to check referential integrity or the with check option on views

Note As of Adaptive Server Enterprise 15.0, the `sp_configure` option to turn on or off join transitive closure and sort merge join is discontinued. Whenever applicable, join transitive closure is always applied in Adaptive Server Enterprise 15.0 and later.

Predicate transformation and factoring to provide additional optimization paths

Predicate transformation and factoring increases the number of choices available to the query processor. It adds optimizable clauses to a query by extracting clauses from blocks of predicates linked with `or` into clauses linked by `and`. The additional optimized clauses mean there are more access paths available for query execution. The original `or` predicates are retained to ensure query correctness.

During predicate transformation:

- 1 Simple predicates (joins, search arguments, and in lists) that are an exact match in each `or` clause are extracted. In the query in step 3, below, this clause matches exactly in each block, so it is extracted:

```
t.pub_id = p.pub_id
```

between clauses are converted to greater-than-or-equal and less-than-or-equal clauses before predicate transformation. The sample query uses between 15 in both query blocks (though the end ranges are different). The equivalent clause is extracted by step 1:

```
price >=15
```

- 2 Search arguments on the same table are extracted; all terms that reference the same table are treated as a single predicate during expansion. Both `type` and `price` are columns in the `titles` table, so the extracted clauses are:

```
(type = "travel" and price >=15 and price <= 30)
or
(type = "business" and price >= 15 and price <= 50)
```

- 3 in lists and `or` clauses are extracted. If there are multiple in lists for a table within a blocks, only the first is extracted. The extracted lists for the sample query are:

```
p.pub_id in ("P220", "P583", "P780")  
or  
p.pub_id in ("P651", "P066", "P629")
```

Since these steps can overlap and extract the same clause, duplicates are eliminated.

Each generated term is examined to determine whether it can be used as an optimized search argument or a join clause. Only those terms that are useful in query optimization are retained.

The additional clauses are added to the query clauses specified by the user.

For example, all clauses optimized in this query are enclosed in the or clauses:

```
select p.pub_id, price  
from publishers p, titles t  
where (  
    t.pub_id = p.pub_id  
    and type = "travel"  
    and price between 15 and 30  
    and p.pub_id in ("P220", "P583", "P780")  
)  
or (  
    t.pub_id = p.pub_id  
    and type = "business"  
    and price between 15 and 50  
    and p.pub_id in ("P651", "P066", "P629")  
)
```

Predicate transformation pulls clauses linked with and from blocks of clauses linked with or, such as those shown above. It extracts only clauses that occur in all parenthesized blocks. If the example above had a clause in one of the blocks linked with or that did not appear in the other clause, that clause would not be extracted.

Handling search arguments and useful indexes

It is important that you distinguish between where and having clause predicates that can be used to optimize the query and those that are used later during query processing to filter the returned rows.

You can use search arguments to determine the access path to the data rows when a column in the where clause matches an index key. The index can be used to locate and retrieve the matching data rows. Once the row has been located in the data cache or has been read into the data cache from disk, any remaining clauses are applied.

For example, if the authors table has on an index on au_lname and another on city, either index can be used to locate the matching rows for this query:

```
select au_lname, city, state
from authors
where city = "Washington"
and au_lname = "Catmull"
```

The query optimizer uses statistics, including histograms, the number of rows in the table, the index heights, and the cluster ratios for the index and data pages to determine which index provides the cheapest access. The index that provides the cheapest access to the data pages is chosen and used to execute the query, and the other clause is applied to the data rows once they have been accessed.

Nonequality operators

The non-equality operators, $<>$ and \neq , are special cases. The query optimizer checks whether it should cover nonclustered indexes if the column is indexed, and uses a nonmatching index scan if an index covers the query. However, if the index does not cover the query, the table is accessed through a row ID lookup of the data pages during the index scan.

Examples of search argument optimization

Shown below are examples of clauses that can be fully optimized. If there are statistics on these columns, they can be used to help estimate the number of rows the query will return. If there are indexes on the columns, the indexes can be used to access the data.

```
au_lname = "Bennett"
price >= $12.00
advance > $10000 and advance < $20000
au_lname like "Ben%" and price > $12.00
```

These search arguments cannot be optimized unless a functional index is built on them:

```
advance * 2 = 5000 /*expression on column side
                    not permitted */
substring(au_lname,1,3) = "Ben" /* function on
```

column name */

These two clauses can be optimized if written in this form:

```
advance = 5000/2
au_lname like "Ben%"
```

Consider this query, with the only index on au_lname:

```
select au_lname, au_fname, phone
  from authors
 where au_lname = "Gerland"
       and city = "San Francisco"
```

The clause qualifies as a search argument:

```
au_lname = "Gerland"
```

- There is an index on au_lname
- There are no functions or other operations on the column name.
- The operator is a valid search argument operator.

This clause matches all the criteria above except the first; there is no index on the city column. In this case, the index on au_lname is used for the query. All data pages with a matching last name are brought into cache, and each matching row is examined to see if the city matches the search criteria.

Handling *joins*

The query optimizer processes join predicates the same way it processes search arguments, in that it uses statistics, number of rows in the table, index heights, and the cluster ratios for the index and data pages to determine which index and join method provides the cheapest access. In addition, the query optimizer also uses join density estimates derived from join histograms that give accurate estimates of qualifying joining rows and the rows to be scanned in the outer and inner tables. The query optimizer also must decide on the optimal join ordering that will yield the most efficient query plan. The next sections describe the key techniques used in processing joins.

Join density and join histograms

The query optimizer uses a cost model for joins that uses table-normalized histograms of the joining attributes. This technique gives an exact value for the skewed values (that is, frequency count) and uses the range cell densities from each histogram to estimate the cell counts of corresponding range cells.

The join density is dynamically computed from the “join histogram,” which considers the joining of histograms from both sides of the join operator. The first histogram join occurs typically between two base tables when both attributes have histograms. Every histogram join creates a new histogram on the corresponding attribute of the parent join's projection.

The outcome of the join histogram technique is accurate join selectivity estimates, even if data distributions of the joining columns are skewed, resulting in superior join orders and performance.

Expression histogramming selectivity estimates

Versions of Adaptive Server earlier than 15.0.2 used default “guesses” for selectivity estimates.

Adaptive Server versions 15.0.2 and later apply histogramming estimates to single column predicates if the histogram exists on the column. This results in more accurate row estimates, and improves the join order selection for query plans.

In this example, if the expression is very selective, it may be better to place table t1 at the beginning of the join order:

```
select * from t1,t2 where substring(t1.charcol, 1, 3)
= "LMC" and t1.a1 = t2.b
```

Joins with mixed datatypes

A basic requirement is the ability to build keys for index lookups whenever possible, without regard to mixed datatypes of any of the join predicates versus the index key. Consider the following query:

```
create table T1 (c1 int, c2 int)
create table T2 (c1 int, c2 float)
create index i1 on T1(c2)
create index i1 on T2(c2)

select * from T1, T2 where T1.c2=T2.c2
```

Assume that T1.c2 is of type int and has an index on it, and that T2.c2 is of type float with an index.

As long as datatypes are implicitly convertible, the query optimizer can use index scans to process the join. In other words, the query optimizer uses the column value from the outer table to position the index scan on the inner table, even when the lookup value from the outer table has a different datatype than the respective index attribute of the inner table.

Joins with expressions and *or* predicates

See “Predicate transformation and factoring to provide additional optimization paths” on page 9 for description of how the query optimizer handles joins with expressions and *or* predicates.

join ordering

One of the key tasks of the query optimizer is to generate a query plan for join queries so that the order of the relations in the joins processed during query execution is optimal. This involves elaborate plan search strategies that can consume significant time and memory. The query optimizer uses several effective techniques to obtain the optimal join ordering. The key techniques are:

- Use of a greedy strategy to obtain an initial good ordering that can be used as an upper boundary to prune out other, subsequent join orderings. The greedy strategy employs join row estimates and the nested-loop-join method to arrive at the initial ordering.
- An exhaustive ordering strategy follows the greedy strategy. In this strategy, a potentially better join ordering replaces the join ordering obtained in the greedy strategy. This ordering may employ any join method.
- Use of extensive cost-based and rule-based pruning techniques eliminates undesirable join orders from consideration. The key aspect of the pruning technique is that it always compares partial join orders (the prefix of a potential join ordering) against the best complete join ordering to decide whether to proceed with the given prefix. This significantly improves the time required to determine an optimal join order.

- The query optimizer can recognize and process star or snowflake schema joins and process their join ordering in the most efficient way. A typical star schema join involves a large Fact table that has equi-join predicates that join it with several Dimension tables. The Dimension tables have no join predicates connecting each other; that is, there are no joins between the Dimension tables themselves, but there are join predicates between the Dimension tables and the Fact table. The query optimizer employs special join ordering techniques during which the large Fact table is pushed to the end of the join order and the Dimension tables are pulled up front, yielding highly efficient query plans. The query optimizer does not, however, use this technique if the star schema joins contain subqueries, outer joins, or or predicates.

Optimization goals

Optimization goals are a convenient way to match query demands with the best optimization techniques, thus ensuring optimal use of the optimizer's time and resources. The query optimizer allows you to configure three types of optimization goals, which you can specify at three tiers: server level, session level, and query level.

Set the optimization goal at the desired level. The server-level optimization goal is overridden at the session level, which is overridden at the query level.

These optimization goals allow you to choose an optimization strategy that best fits your query environment:

- `allows_mix` – the default goal, and the most useful goal in a mixed-query environment. `allows_mix` balances the needs of OLTP and DSS query environments.
- `allows_dss` – the most useful goal for operational DSS queries of medium to high complexity. Currently, this goal is provided on an experimental basis.
- `allows_oltp` – the optimizer considers only nested-loop joins.

At the server level, use `sp_configure`. For example:

```
sp_configure "optimization goal", 0, "allows_mix"
```

At the session level, use `set plan optgoal`. For example:

```
set plan optgoal allows_dss
```

At the query level, use a `select` or other DML command. For example:

```
select * from A order by A.a plan
" (use optgoal allrows_dss) "
```

In general, you can set query-level optimization goals using `select`, `update`, and `delete` statements. However, you cannot set query-level optimization goals in pure insert statements, although you can set optimization goals in `insert...select` statements.

Limiting the time spent optimizing a query

Long-running and complex queries can be time-consuming and costly to optimize. The timeout mechanism helps to limit that time while supplying a satisfactory query plan. The query optimizer provides a mechanism by which the optimizer can limit the time taken by long-running and complex queries; timing out allows the query processor to stop optimizing when it is reasonable to do so.

The optimizer triggers timeout during optimization when both these circumstances are met:

- At least one complete plan has been retained as the best plan, and
- The user configured timeout percentage limit has been exceeded.

You can limit the amount of time Adaptive Server spends optimizing a query at every level, using the `optimization timeout limit` parameter, which you can set to any value between 0 and 1000. `optimization timeout limit` represents the percentage of estimated query execution time that Adaptive Server must spend to optimize the query. For example, specifying a value of 10 tells Adaptive Server to spend 10% of the estimated query execution time in optimizing the query. Similarly, a value of 1000 tells Adaptive Server to spend 1000% of the estimated query execution time, or 10 times the estimated query execution time, in optimizing the query.

See Chapter 5, “Setting Configuration Parameters,” in the *System Administration Guide* for more information about optimization timeout limit.

A large timeout value may be useful for optimization of stored procedures with complex queries. Generally, longer optimization time of the stored procedures yields better plans; the longer optimization time can be amortized over several executions of the stored procedure.

A small timeout value may be used when you want a faster compilation time from complex ad hoc queries that normally take a long time to compile. However, for most queries, the default timeout value of 10 should suffice.

Use `sp_configure` to set the optimization timeout limit configuration parameter at the server level. For example, to limit optimization time to 10% of total query processing time, enter:

```
sp_configure "optimization timeout limit", 10
```

Use `set` to set timeout at the session level:

```
set plan opttimeoutlimit <n>
```

where *n* is any integer between 0 and 4000.

Use `select` to limit optimization time at the query level:

```
select * from <table> plan "(use opttimeoutlimit <n>)"
```

where *n* is any integer between 0 and 1000.

Parallelism

Adaptive Server supports horizontal and vertical parallelism for query execution. Vertical parallelism is the ability to run multiple operators at the same time by employing different system resources such as CPUs, disks, and so on. Horizontal parallelism is the ability to run multiple instances of an operator on the specified portion of the data.

See Chapter 5, “Parallel Query Processing,” for a more detailed discussion of parallel query optimization in Adaptive Server.

Optimization issues

Although the query optimizer can efficiently optimize most queries, these issues may effect the optimizer’s efficiency:

- If statistics have not been updated recently, the actual data distribution may not match the values used to optimize queries.

- The rows referenced by a specified transaction may not fit the pattern reflected by the index statistics.
- An index may access a large portion of the table.
- where clauses (SARGS) are written in a form that cannot be optimized.
- No appropriate index exists for a critical query.
- A stored procedure was compiled before significant changes to the underlying tables were performed.
- No statistics exists for the SARG or joining columns.

These situations highlight the need to follow some best practices that allow the query optimizer to perform at its full potential:

Creating search arguments

Follow these guidelines when you write search arguments for your queries:

- Avoid functions, arithmetic operations, and other expressions on the column side of search clauses. When possible, move functions and other operations to the expression side of the clause.
- Use as many search arguments as you can, to give the query processor as much as possible to work with.
- If a query has more than 400 predicates for a table, put the most potentially useful clauses near the beginning of the query, since only the first 102 SARGs on each table are used during optimization. (All of the search conditions are used to qualify the rows.)
- Queries using > (greater than) may perform better if you can rewrite them to use >= (greater than or equal to). For example, this query, with an index on int_col, uses the index to find the first value where int_col equals 3, and then scans forward to find the first value that is greater than 3. If there are many rows where int_col equals 3, the server must scan many pages to find the first row where int_col is greater than 3:

```
select * from table1 where int_col > 3
```

It is more efficient to write the query this way:

```
select * from table1 where int_col >= 4
```

This optimization is more difficult with character strings and floating-point data.

- Check the showplan output to see which keys and indexes are used.

- If an index is not being used when you expect it to be, use output from the set commands in Table 3-1 on page 112 to see whether the query processor is considering the index.

Use of SQL derived tables

Queries expressed as a single SQL statement exploit the query processor better than queries expressed in two or more SQL statements. SQL-derived tables enable you to express, in a single step, what might otherwise require several SQL statements and temporary tables, especially where intermediate aggregate results must be stored. For example:

```
select dt_1.* from
  (select sum(total_sales)
   from titles_west group by total_sales)
   dt_1(sales_sum),
  (select sum(total_sales)
   from titles_east group by total_sales)
   dt_2(sales_sum)
where dt_1.sales_sum = dt_2.sales_sum
```

Here, aggregate results are obtained from the SQL derived tables `dt_1` and `dt_2`, and a join is computed between the two SQL derived tables. Everything is accomplished in a single SQL statement.

For more information, see Chapter 9, “SQL Derived Tables,” in the *Transact-SQL User's Guide*.

Tuning according to object sizes

To understand query and system behavior, know the sizes of your tables and indexes. At several stages of tuning work, you need size data to:

- Understand statistics i/o reports for a specific query plan.
- Understand the query processor's choice of query plan. The Adaptive Server cost-based query processor estimates the physical and logical I/O required for each possible access method and selects the cheapest method.
- Determine object placement, based on the sizes of database objects and on the expected I/O patterns on the objects.

To improve performance, distribute database objects across physical devices, so that reads and writes to disk are evenly distributed.

Object placement is described in the *Performance and Tuning Series: Physical Database Tuning*.

- Understand changes in performance. If objects grow, their performance characteristics can change. For example, consider a table that is heavily used and is usually 100 percent cached. If the table grows too large for its cache, queries that access the table can suffer poor performance. This is particularly true of joins that require multiple scans.

- Do capacity planning. Whether you are designing a new system or planning for the growth of an existing system, you must know the space requirements to plan for physical disks and memory needs.
- Understand output from `sp_sysmon` reports on physical I/O.

See the *System Administration Guide: Volume 2* for more information on sizing.

Lava query execution engine

In Adaptive Server, all query plans are submitted to the procedural execution engine for execution. The Procedural Execution Engine drives execution of the query plan by:

- Executing simple SQL statements such as `set`, `while`, and `goto` directly.
- Calling out to the utility modules of the query plan to execute `create table` and `create index` and other utility commands.
- Setting up the context for and driving the execution of stored procedures and triggers.
- Setting up the execution context and calling the query execution engine to execute query plans for `select`, `insert`, `delete`, and `update` statements.
- Setting up the cursor execution context for `cursor open`, `fetch`, and `close` statements, and calling the query execution engine to execute these statements.
- Doing transaction processing and post execution cleanup.

To support the demands of today's applications, the query execution engine for Adaptive Server 15.0 and later has been completely rewritten. With a new query execution engine and query optimizer in place, the procedural execution engine in Adaptive Server 15.0 and later passes all query plans generated by the new query optimizer to the Lava query execution engine.

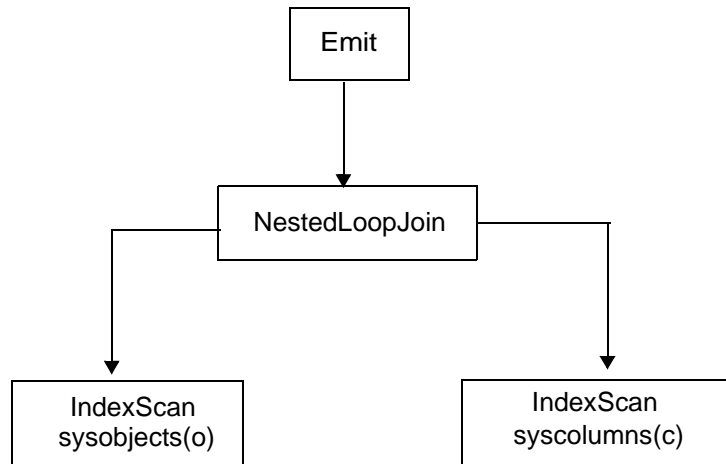
The Lava query execution engine executes Lava query plans. All query plans chosen by the optimizer are compiled into Lava query plans. However, SQL statements that are not optimized, such as `set` or `create`, are compiled into query plans like those in versions of Adaptive Server earlier than 15.0, and are not executed by the Lava query execution engine. Non-Lava query plans are either executed by the procedural execution engine or by utility modules called by the procedural engine. Adaptive Server version 15.0 and later has two distinct kinds of query plans and this is clearly seen in the showplan output (see Chapter 3, “Displaying Query Optimization Strategies and Estimates”).

Lava query plans

A Lava query plan is built as an upside down tree of Lava operators: The top Lava operator can have one or more child operators, which in turn can have one or more child operators, and so on, thus building a bottom-up tree of operators. The exact shape of the tree and the operators in it are chosen by the optimizer.

An example of a Lava query plan for this query is shown in Figure 1-2:

```
Select o.id from sysobjects o, syscolumns c
where o.id <= 1 and o.id < 2
```

Figure 1-2: Lava query plan

The Lava query plan for this query consists of four Lava operators. The top operator is an Emit (also called Root) operator that dispatches the results of query execution either by sending the rows to the client or by assigning values to local variables.

The only child operator of the Emit is a NestedLoopJoin (NL Join) that uses the nested loop join algorithm to join the rows coming from its two child operators, (1) the Scan of sysobjects and (2) the scan of syscolumns.

Since the optimizer optimizes all select, insert, delete and update statements, these are always compiled into Lava query plans and executed by the Lava query engine.

Some SQL statements are compiled into hybrid query plans. Such plans have multiple steps, some of which are executed by the Utility modules, and a final step that is a Lava query plan. An example is the select into statement; select into is compiled into a two-step query plan:

- create table creates the target table of the statement.
- A Lava query plan to inserts the rows into the target table. To execute this query plan, the Procedural Execution Engine calls the create table utility to execute the first step to create the table.

Then the procedural engine calls the Lava query execution engine to execute the Lava query plan to select and insert the rows into the target table.

The two other SQL statements that generate hybrid query plans are `alter table` (but only when data copying is required) and `reorg rebuild`.

A Lava query plan is also generated and executed to support `bcu`. The support for `bcu` in Adaptive Server has always been in the `bcu` utility. Now, in version 15.0 and later, the `bcu` utility generates a Lava query plan and calls the Lava query execution engine to execute the plan.

See Chapter 2, “Using `showplan`” for more examples of Lava query plans.

Lava operators

Lava query plans are built of Lava operators. Each Lava operator is a self-contained software object that implements one of the basic physical operations that the optimizer uses to build query plans. Each Lava operator has five methods that can be called by its parent operator. These five methods correspond to the five phases of query execution:

- `Acquire`
- `Open`
- `Next`
- `Close`,
- `Release`

Because the Lava operators all provide the same methods (that is, the same API), they can be interchanged like building blocks in a Lava query plan. For example, you can replace the `NLJoin` operator in Figure 1-2 with a `MergeJoin` operator or a `HashJoin` operator without impacting any of the other three operators in the query plan.

The Lava operators that can be chosen by the optimizer to build Lava query plans are listed in Table 1-3:

Table 1-3: Lava operators

Operator	Description
<code>BulkOp</code>	Executes the part of <code>bcu</code> processing that is done in the Lava query engine. Only found in query plans that are created by the <code>bcu</code> utility, not those created by the optimizer.
<code>CacheScanOp</code>	Reads rows from an in-memory table.
<code>DelTextOp</code>	Deletes text page chains as part of <code>alter table drop column</code> processing.
<code>DeleteOp</code>	Deletes rows from a local table. Deletes rows from a proxy table when the entire SQL statement cannot be shipped to the remote server. See also <code>RemoteScanOp</code> .

Operator	Description
EmitOp (RootOp)	Routes query execution result rows. Can send results to the client or assign result values to local variables or fetch into variables. An EmitOp is always the top operator in a Lava query plan.
EmitExchangeOp	Routes result rows from a subplan that is executed in parallel to the ExchangeOp in the parent plan fragment. EmitExchangeOp always appears directly under an ExchangeOp. See Chapter 5, “Parallel Query Processing.”
GroupSortedOp (Aggregation)	Performs vector aggregation (group by) when the input rows are already sorted on the group-by columns. See also HashVectorAggOp.
GroupSorted (Distinct)	Eliminates duplicate rows. Requires the input rows to be sorted on all columns. See also HashDistinctOp and SortOp (Distinct).
HashVectorAggOp	Performs vector aggregation (group by). Uses a Hash algorithm to group the input rows, so no requirements on ordering of the input rows. See also GroupSortedOp (Aggregation).
HashDistinctOp	Eliminates duplicate rows using a hashing algorithm to find duplicate rows. See also GroupSortedOp (Distinct) and SortOp (Distinct).
HashJoinOp	Performs a join of two input row streams using the HashJoin algorithm.
HashUnionOp	Performs a union operation of two or more input row streams using a hashing algorithm to find and eliminate duplicate rows. See also MergeUnionOp and UnionAllOp.
InsScrollOp	Implements extra processing needed to support insensitive scrollable cursors. See also SemiInsScrollOp.
InsertOp	Inserts rows to a local table. Inserts rows to a proxy table when the entire SQL statement cannot be shipped to the remote server. See also RemoteScanOp.
MergeJoinOp	Performs a join of two streams of rows that are sorted on the joining columns using the mergejoin algorithm.
MergeUnionOp	Performs a union or union all operation on two or more sorted input streams. Guarantees that the output stream retains the ordering of the input streams. See also HashUnionOp and UnionAllOp.
NestedLoopJoinOp	Performs a join of two input streams using the NestedLoopJoin algorithm.
NaryNestedLoopJoinOp	Performs a join of three or more input streams using an enhanced NestedLoopJoin algorithm. This operator replaces a left-deep tree of NestedLoopJoin operators and can lead to significant performance improvements when rows of some of the input streams can be skipped.
OrScanOp	Inserts the in or or values into an in-memory table, sorts the values, and removes the duplicates. Then returns the values, one at a time. Only used for SQL statements with in clauses or multiple or clauses on the same column.
PtnScanOp	Reads rows from a local table (partitioned or not) using either a table scan or an index scan to access the rows.
RIDJoinOp	Receives one or more row identifiers (RIDs) from its left-child operator and calls on its right-child operator (PtnScanOp) to find the corresponding rows. Used only on SQL statements with or clauses on different columns of the same table.

Operator	Description
RIFilterOp (Direct)	Drives the execution of a subplan to enforce referential integrity constraints that can be checked on a row-by-row basis. Appears only in insert, delete, or update queries on tables with referential integrity constraints.
RIFilterOp (Deferred)	Drives the execution of a sub-plan to enforce referential integrity constraints that can only be checked after all rows that are affected by the query have been processed.
RemoteScanOp	Accesses proxy tables. The RemoteScanOp can: <ul style="list-style-type: none"> • Read rows from a single proxy table for further processing in a Lava query plan on the local host. • Pass complete SQL statements to a remote host for execution: insert, delete, update, and select statements. In this case, the Lava query plan consists of an EmitOp with a RemoteScanOp as its only child operator. • Pass an arbitrarily complex query plan fragment to a remote host for execution and read in the result rows (function shipping).
RestrictOp	Evaluates expressions.
SQFilterOp	Drives the execution of a subplan to execute one or more subqueries.
ScalarAggOp	Performs scalar aggregation, such as aggregates without group by.
SemilnsScrollOp	Performs extra processing to support semi-insensitive scrollable cursors. See also InsScrollOp.
SequencerOp	Enforces sequential execution of different sub-plans in the query plan.
SortOp	Sorts its input rows based upon specified keys.
SortOp (Distinct)	Sorts its input and removes duplicate rows. See also HashDisitnctOp and GroupSortedOp (Distinct).
StoreOp	Creates and coordinates the filling of a worktable, and creates a clustered index on the worktable if required. StoreOp can only have an InsertOp as a child; the InsertOp populates the worktable.
UnionAllOp	Performs a union all operation on two or more input streams. See also HashUnionOp and MergeUnionOp.
UpdateOp	Changes the value of columns in rows of a local table or of a proxy table when the entire update statement cannot be sent to the remote server. See also RemoteScanOp.
ExchangeOp	Enables and coordinates parallel execution of Lava query plans. The ExchangeOp can be inserted between almost any two Lava operators in a query plan to divide the plan into sub-plans that can be executed in parallel. See Chapter 5, “Parallel Query Processing.”

Lava query execution

Execution of a Lava query plan involves five phases:

- 1 Acquire – acquires resources needed for execution, such as memory buffers and creating worktables.
- 2 Open – prepares to return result rows.
- 3 Next – generates the next result row.
- 4 Close – cleans up; for example, notifies the access layer that scanning is complete or truncates worktables
- 5 Release – releases resources acquired during Acquire, such as memory buffers, drops worktables.

Each Lava operator has a method with the same name as the phase, which is invoked for each of these phases.

Figure 1-2 on page 22 demonstrates query plan execution:

- Acquire phase

The acquire method of the Emit operator is invoked. The Emit operator calls Acquire of its child, the NLJoin operator, which in turn calls Acquire on its left-child operator (the index scan of *sysobjects*) and then on its right child operator (the index scan of *syscolumns*).

- Open phase

The Open method of the Emit operator is invoked. The Emit operator calls Open on the NLJoin operator, which calls Open only on its left-child operator.

- Next phase

The Next method of the Emit operator is invoked. Emit calls Next on the NLJoin operator, which calls Next on its left child, the Index Scan of *sysobjects*. The index scan operator reads the first row from *sysobjects* and returns it to the NLJoin operator. The NLJoin operator then calls the Open method of its right child operator, the Index Scan of *syscolumns*. Then the NLJoin operator calls the Next method of the Index Scan of *syscolumns* to get a row that matches the joining key of the row from *sysobjects*. When a matching row has been found, it is returned to the Emit operator, which sends it back to the client. Repeated invocations of the Next method of the Emit operator generate more result rows.

- Close phase

After all rows have been returned, the Close method of the Emit operator is invoked, which in turn calls Close of the NLJoin operator, which in turn calls Close on both of its child operators.

- Release phase

The Release method of the Emit operator is invoked and the calls to the Release method of the other operators is propagated down the query plan.

After successfully completing the Release phase of execution, the Lava query engine returns control to the Procedural Execution Engine for final statement processing.

How update operations are performed

Adaptive Server handles updates in different ways, depending on the changes being made to the data and the indexes used to locate the rows. The two major types of updates are **deferred updates** and **direct updates**. Adaptive Server performs direct updates whenever possible.

Direct updates

Adaptive Server performs direct updates in a single pass:

- It locates the affected index and data rows.
- It writes the log records for the changes to the transaction log.
- It makes the changes to the data pages and any affected index pages.

There are three techniques for performing direct updates:

- In-place updates
- Cheap direct updates
- Expensive direct updates

Direct updates require less overhead than deferred updates and are generally faster, as they limit the number of log scans, reduce logging, save traversal of index B-trees (reducing lock contention), and save I/O because Adaptive Server does not have to refetch pages to perform modifications based on log records.

In-place updates

Adaptive Server performs in-place updates whenever possible.

When Adaptive Server performs an in-place update, subsequent rows on the page are not moved; row IDs remain the same and the pointers in the row offset table are not changed.

For an in-place update, the following requirements must be met:

- The row being changed cannot change its length.
- The column being updated cannot be the key, or part of the key, of a clustered index on an allpages-locked table. Because the rows in a clustered index on an allpages-locked table are stored in key order, a change to the key almost always means that the row location is changed.
- One or more indexes must be unique or must allow duplicates.
- The update statement satisfies the conditions listed in “Restrictions on update modes through joins” on page 33.
- The affected columns are not used for referential integrity.
- There cannot be a trigger on the column.
- The table cannot be replicated (via Replication Server).

An in-place update is the fastest type of update because it makes a single change to the data page. It changes all affected index entries by deleting the old index rows and inserting the new index row. In-place updates affect only indexes whose keys are changed by the update, since the page and row locations are not changed.

Cheap direct updates

If Adaptive Server cannot perform an update in place, it tries to perform a cheap direct update—changing the row and rewriting it at the same offset on the page. Subsequent rows on the page are moved up or down so that data remains contiguous on the page, but row IDs remain the same. The pointers in the row offset table change to reflect the new locations.

A cheap direct update must meet these requirements:

- The length of the data in the row is changed, but the row still fits on the same data page, or the row length is not changed, but there is a trigger on the table or the table is replicated.
- The column being updated cannot be the key, or part of the key, of a clustered index. Because Adaptive Server stores the rows of a clustered index in key order, a change to the key almost always means that the row location is changed.

- One or more indexes must be unique or must allow duplicates.
- The update statement satisfies the conditions listed in “Restrictions on update modes through joins” on page 33.
- The affected columns are not used for referential integrity.

Cheap direct updates are almost as fast as in-place updates. They require the same amount of I/O, but slightly more processing. Two changes are made to the data page (the row and the offset table). Any changed index keys are updated by deleting old values and inserting new values. Cheap direct updates affect only indexes whose keys are changed by the update, since the page and row ID are not changed.

Expensive direct updates

If the data does not fit on the same page, Adaptive Server performs an expensive direct update, if possible. An expensive direct update deletes the data row, including all index entries, and then inserts the modified row and index entries.

Adaptive Server uses a table scan or an index to find the row in its original location and then deletes the row. If the table has a clustered index, Adaptive Server uses the index to determine the new location for the row; otherwise, Adaptive Server inserts the new row at the end of the heap.

An expensive direct update must meet these requirements:

- The length of a data row is changed so that the row no longer fits on the same data page, and the row is moved to a different page, or the update affects key columns for the clustered index.
- The index used to find the row is not changed by the update.
- The update statement satisfies the conditions listed in “Restrictions on update modes through joins” on page 33.
- The affected columns are not used for referential integrity.

An expensive direct update is the slowest type of direct update. The delete is performed on one data page, and the insert is performed on a different data page. All index entries must be updated, since the row location is changed.

Deferred updates

Adaptive Server uses deferred updates when direct update conditions are not met. A deferred update is the slowest type of update.

In a deferred update, Adaptive Server:

- Locates the affected data rows, writing the log records for deferred delete and insert of the data pages as rows are located.
- Reads the log records for the transaction and performs the deletes on the data pages and any affected index rows.
- Reads the log records a second time, and performs all inserts on the data pages, and inserts any affected index rows.

When deferred updates are required

Deferred updates are always required for:

- Updates that use self-joins
- Updates to columns used for self-referential integrity
- Updates to a table referenced in a correlated subquery

Deferred updates are also required when:

- An update moves a row to a new page while the table is being accessed by a table scan or a clustered index.
- Duplicate rows are not allowed in the table, and there is no unique index to prevent them.
- The index used to find the data row is not unique, and the row is moved because the update changes the clustered index key or because the new row does not fit on the page.

Deferred updates incur more overhead than direct updates because they require Adaptive Server to reread the transaction log to make the final changes to the data and indexes. This involves additional traversal of the index trees.

For example, if there is a clustered index on title, this query performs a deferred update:

```
update titles set title = "Portable C Software" where  
title = "Designing Portable Software"
```


Deferred index inserts

Adaptive Server performs deferred index updates when the update affects the index used to access the table or when the update affects columns in a unique index. In this type of update, Adaptive Server:

- Deletes the index entries in direct mode
- Updates the data page in direct mode, writing the deferred insert records for the index
- Reads the log records for the transaction and inserts the new values in the index in deferred mode

Deferred index insert mode must be used when the update changes the index used to find the row or when the update affects a unique index. A query must update a single, qualifying row only once—deferred index update mode ensures that a row is found only once during the index scan and that the query does not prematurely violate a uniqueness constraint.

The update in Figure 1-3 on page 32 changes only the last name, but the index row is moved from one page to the next. To perform the update, Adaptive Server:

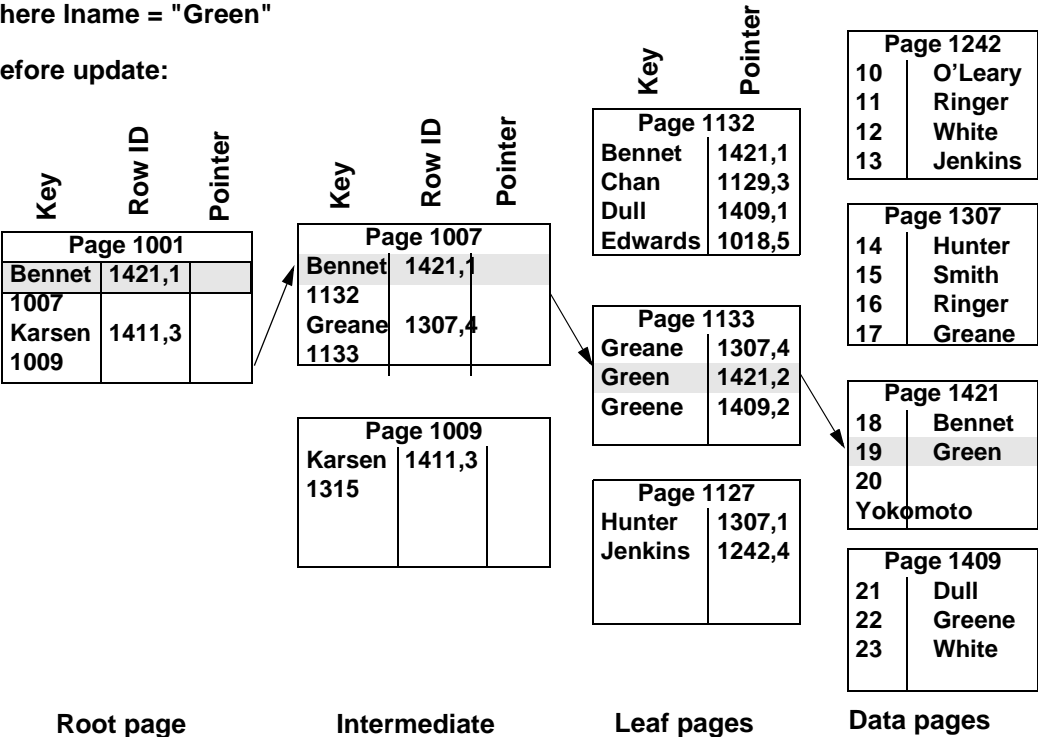
- 1 Reads index page 1133, deletes the index row for “Greene” from that page, and logs a deferred index scan record.
- 2 Changes “Green” to “Hubbard” on the data page in direct mode and continues the index scan to see if more rows need to be updated.
- 3 Inserts the new index row for “Hubbard” on page 1127.

Figure 1-3 shows the index and data pages prior to the deferred update operation, and the sequence in which the deferred update changes the data and index pages.

Figure 1-3: Deferred index update

update employee
set Iname = "Hubbard"
where Iname = "Green"

Before update:



Root page

Intermediate

Leaf pages

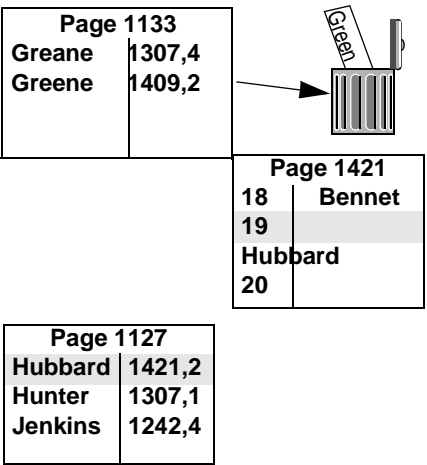
Data pages

Update steps

Step 1: Write log records, then delete index row.

Step 2: Change data page.

Step 3: Read log, insert index row.



Assume a similar update to the titles table:

```
update titles
set title = "Computer Phobic's Manual",
    advance = advance * 2
where title like "Computer Phob%"
```

This query shows a potential problem. If a scan of the nonclustered index on the title column found “Computer Phobia Manual,” changed the title, and multiplied the advance by 2, and then found the new index row “Computer Phobic’s Manual” and multiplied the advance by 2, the advance would be very skewed against the reality.

A deferred index delete may be faster than an expensive direct update, or it may be substantially slower, depending on the number of log records that need to be scanned and whether the log pages are still in cache.

During deferred update of a data row, there can be a significant time interval between the delete of the index row and the insert of the new index row. During this interval, there is no index row corresponding to the data row. If a process scans the index during this interval at isolation level 0, it does not return the old or new value of the data row.

Restrictions on update modes through joins

Updates and deletes that involve joins can be performed in direct, deferred_varcol, or deferred_index mode when the table being updated is the outermost table in the join order, or when it is preceded in the join order by tables where only a single row qualifies.

Joins and subqueries in update and delete statements

The use of the from clause to perform joins in update and delete statements is a Transact-SQL extension to ANSI SQL. Subqueries in ANSI SQL form can be used in place of joins for some updates and deletes.

This example uses the from syntax to perform a join:

```
update t1 set t1.c1 = t1.c1 + 50
from t1, t2
where t1.c1 = t2.c1
and t2.c2 = 1
```

The following example shows the equivalent update using a subquery:

```
update t1 set c1 = c1 + 50
```

```
where t1.c1 in (select t2.c1
               from t2
               where t2.c2 = 1)
```

The update mode that is used for the join query depends on whether the updated table is the outermost query in the join order—if it is not the outermost table, the update is performed in deferred mode. The update that uses a subquery is always performed as a direct, `deferred_varcol`, or `deferred_index` update.

For a query that uses the `from` syntax and performs a deferred update due to the join order, use `showplan` and `statistics io` to determine whether rewriting the query using a subquery can improve performance. Not all queries using `from` can be rewritten to use subqueries.

Deletes and updates in triggers versus referential integrity

Triggers that join user tables with the `deleted` or `inserted` tables are run in deferred mode. If you are using triggers solely to implement referential integrity, and not to cascade updates and deletes, then using declarative referential integrity in place of triggers may avoid the penalty of deferred updates in triggers.

Optimizing updates

`showplan` messages provide information about whether an update is performed in direct mode or deferred mode. If a direct update is not possible, Adaptive Server updates the data row in deferred mode. There are times when the optimizer cannot know whether a direct update or a deferred update will be performed, so two `showplan` messages are provided:

- The “`deferred_varcol`” message shows that the update may change the length of the row because a variable-length column is being updated. If the updated row fits on the page, the update is performed in direct mode; if the update does not fit on the page, the update is performed in deferred mode.
- The “`deferred_index`” message indicates that the changes to the data pages and the deletes to the index pages are performed in direct mode, but the inserts to the index pages are performed in deferred mode.

These types of direct updates depend on information that is available only at runtime, since the page actually has to be fetched and examined to determine whether the row fits on the page.

Designing for direct updates

When you design and code your applications, be aware of the differences that can cause deferred updates. To help avoid deferred updates:

- Create at least one unique index on the table to encourage more direct updates.
- Whenever possible, use nonkey columns in the where clause when updating a different key.
- If you do not use null values in your columns, declare them as not null in your create table statement.

Effects of update types and indexes on update modes

Table 1-4 on page 36 shows how indexes affect the update mode for three different types of updates. In all cases, duplicate rows are not allowed. For the indexed cases, the index is on title_id. The three types of updates are:

- Update of a variable-length key column:

```
update titles set title_id = value
where title_id = "T1234"
```

- Update of a fixed-length nonkey column:

```
update titles set pub_date = value
where title_id = "T1234"
```

- Update of a variable-length nonkey column:

```
update titles set notes = value
where title_id = "T1234"
```

Table 1-4 shows how a unique index can promote a more efficient update mode than a nonunique index on the same key. Pay particular attention to the differences between direct and deferred in the shaded areas of the table. For example, with a unique clustered index, all of these updates can be performed in direct mode, but they must be performed in deferred mode if the index is nonunique.

For a table with a nonunique clustered index, a unique index on any other column in the table provides improved update performance. In some cases, you may want to add an IDENTITY column to a table to include the column as a key in an index that would otherwise be nonunique.

Table 1-4: Effects of indexing on update mode

Index	Update to:		
	Variable-length key	Fixed-length column	Variable-length column
No index	N/A	direct	deferred_varcol
Clustered, unique	direct	direct	direct
Clustered, not unique	deferred	deferred	deferred
Clustered, not unique, with a unique index on another column	deferred	direct	deferred_varcol
Nonclustered, unique	deferred_varcol	direct	direct
Nonclustered, not unique	deferred_varcol	direct	deferred_varcol

If the key for an index is fixed length, the only difference in update modes from those shown in the table occurs for nonclustered indexes. For a nonclustered, nonunique index, the update mode is deferred_index for updates to the key. For a nonclustered, unique index, the update mode is direct for updates to the key.

If the length of varchar or varbinary is close to the maximum length, use char or binary instead. Each variable-length column adds row overhead and increases the possibility of deferred updates.

Using max_rows_per_page to reduce the number of rows allowed on a page increases direct updates, because an update that increases the length of a variable-length column may still fit on the same page.

For more information on using max_rows_per_page, see the *Performance and Tuning Series: Physical Database Tuning*.

Using sp_sysmon while tuning updates

You can use showplan to determine whether an update is deferred or direct, but showplan does not give you detailed information about the type of deferred or direct update. Output from the sp_sysmon or Adaptive Server Monitor supplies detailed statistics about the types of updates performed during a sample interval.

Run sp_sysmon as you tune updates, and look for reduced numbers of deferred updates, reduced locking, and reduced I/O.

See the *Performance and Tuning Series: Monitoring Adaptive Server with sp_sysmon*.

This chapter describes the messages printed by the *showplan* utility, which displays the query plan in a text-based format for each SQL statement in a batch or stored procedure.

Topic	Page
Displaying a query plan	37
Statement-level output	44
Query plan shape	47
Union operators	90
INSTEAD-OF TRIGGER operators	106

Displaying a query plan

To see query plans, use:

```
set showplan on
```

To stop displaying query plans, use:

```
set showplan off
```

You can use *showplan* in conjunction with other *set* commands.

To display query plans for a stored procedure, but not execute them, use the *set fmtonly* command.

See Chapter 12, “Creating and Using Abstract Plans,” for information on how options interact.

Note Do not use *set noexec* with stored procedures—compilation and execution does not occur and you do not receive the necessary output.

Query plans in Adaptive Server Enterprise 15.0 and later

Adaptive Server traditionally classifies Transact-SQL statements into two groups:

- **Optimizable.** For example, this query is optimizable because it has many relations (tables):

```
select * from t1, t2, t3, t4
where t1.c1 = t2.c1 and . . .
order by t3.c4
```

The query processor requires the join order, type of join, search arguments, and ordering to be optimized.

- **Nonoptimizable.** Utility commands like update statistics and dbcc are not optimized.

In Adaptive Server version 15.0 and later the optimizer and most of the execution engine was rewritten. The utility commands currently generate nearly identical showplan output compared with earlier versions. However, versions 15.0 and later generate new showplan output for optimizable statements.

Some of the new features of the query plans that showplan must display include:

- **Plan elements** – query plans can be composed from over thirty different operators.
- **Plan shape** – query plans are upside-down trees of operators. In general, more operators in a query plan result in more combinations of possible tree shapes. Query plans in version 15.0 and later can be more complex than those found in earlier versions. Nested indentation is provided to assist in visualizing the tree shape of these query plans.
- **Subplans** that are executed in parallel.

Why do I get different query plans for the same query?

The query processor may return a different query plan depending on whether you configure set plan optgoal for allrows_oltp, allrows_mix, or allrows_dss (unless you force a plan with forceplan):

- **allrows_oltp** – the query processor uses the nested-loop join operator.

- `allows_mix` – the query processor allows both nested-loop joins and merge joins. The query processor measures their relative costs to determine which join it uses.
- `allows_dss` – the query processor uses nested-loop, merge-, or hash-joins. The query processor measures their relative costs to determine which join it uses.

Using set showplan with noexec

You can use `set noexec` with `set showplan on` to view a query plan without executing the query. For example, this query prints the query plan but also executes the queries, which might be time consuming:

```
set showplan on
go
select * from really_big_table
select * from really_really_big_table
go
```

However, if you include `set noexec`, you can view the query plan without running the query.

Stored procedures are compiled when they are first used, or if the resultant compiled plan is already in use by another session, so `set noexec` can have unexpected results, and Sybase® recommends that you use `set fmtonly on` instead. If you include a stored procedure inside another stored procedure, the second stored procedure is not run when you enable `set noexec`. For example, if you create two stored procedures:

```
create procedure sp_B
as
begin
    select * from authors
end
```

and

```
create procedure sp_A
as
begin
    select * from titles
    execute sp_B
end
```

Individually, their query plans look like this:

```
set showplan on
sp_B
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
    The type of query is EXECUTE.
QUERY PLAN FOR STATEMENT 1 (at line 4).
```

```
STEP 1

The type of query is SELECT.
1 operator(s) under root
|ROOT:EMIT Operator (VA = 1)
|
| |SCAN Operator (VA = 0)
| |  FROM TABLE
| |  titles
| |  Table Scan.
| |  Forward Scan.
| |  Positioning at start of table.
| |  Using I/O Size 2 Kbytes for data pages.
| |  With LRU Buffer Replacement Strategy for data pages.
```

If you enable set noexec:

```
set noexec on
go
set showplan on
go
exec proc A
go
```

Adaptive Server produces no showplan output for procedure B because noexec is enabled, so Adaptive Server is not actually executing or compiling procedure B, and does not print any showplan output. If noexec was not enabled, Adaptive Server would have compiled and printed plans for both A and B stored procedures.

But if you use set fmtonly on:

```
use pubs2
go
create procedure sp_B
as
begin
    select * from authors
```

```
end
go
create procedure sp_A
as
begin
    select * from titles
    execute sp_B
end
go
set showplan on
go
set fmtonly on
go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SET OPTION ON.

```
sp_B
go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is EXECUTE.

QUERY PLAN FOR STATEMENT 1 (at line 4).

STEP 1

The type of query is SELECT.
1 operator(s) under root
|ROOT:EMIT Operator (VA = 1)
|
| |SCAN Operator (VA = 0)
| | |FROM TABLE
| | |authors
| | |Table Scan.
| | |Forward Scan.
| | |Positioning at start of table.
| | |Using I/O Size 2 Kbytes for data pages.
| | |With LRU Buffer Replacement Strategy for data pages.

au_id	au_lname	phone	address	city	state	country	postalcode	au_fname
-----	-----	-----	-----	-----	-----	-----	-----	-----

(0 rows affected)
(return status = 0)
sp_A
go

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is EXECUTE.

QUERY PLAN FOR STATEMENT 1 (at line 4).

STEP 1

The type of query is SELECT.
1 operator(s) under root
|ROOT:EMIT Operator (VA = 1)
|
| |SCAN Operator (VA = 0)
| | |FROM TABLE
| | |titles
| | |Table Scan.
| | |Forward Scan.
| | |Positioning at start of table.
| | |Using I/O Size 2 Kbytes for data pages.
| | |With LRU Buffer Replacement Strategy for data pages.

QUERY PLAN FOR STATEMENT 2 (at line 5).

STEP 1
The type of query is EXECUTE.

title_id	title
----------	-------

Statement-level output

The first section of showplan output for each query plan presents statement-level information, including the statement and line number in the batch or stored procedure of the query for which the query plan was generated:

```
QUERY PLAN FOR STATEMENT N (at line N).
```

This message may be followed by a series of messages that apply to the statement's entire query plan. If the query plan was generated using an abstract plan about how the abstract plan was forced:

- If an explicit abstract plan was given by a plan clause in the SQL statement, the message is:

```
Optimized using the Abstract Plan in the PLAN clause.
```

- If an abstract plan has been internally generated (that is, for alter table and reorg commands that are executed in parallel), the message is:

```
Optimized using the forced options (internally  
generated Abstract Plan).
```

- If a new statement is cached, the output includes:

```
STEP 1
```

```
The type of query is EXECUTE.  
Executing a newly cached statement.
```

- If a cached statement is reused, the output includes:

```
STEP 1
```

```
The type of query is EXECUTE.  
Executing a previously cached statement.
```

- If the query recompiles the statement, the output includes:

```
QUERY PLAN IS RECOMPILED DUE TO SCHEMACT.
```

```
THE RECOMPILED QUERY PLAN IS:
```

```
. . .
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1)
```

```
. . .
```

- If an abstract plan has been retrieved from sysqueryplans because automatic abstract plan usage is enabled, the message is:

Optimized using an Abstract Plan (ID : N).

- If the query plan is a parallel query plan, the following message shows the number of processes (coordinator plus worker) that are required to execute the query plan:

Executed in parallel by coordinating process and N worker processes.

- If the query plan was optimized using simulated statistics, this message appears next:

Optimized using simulated statistics.

- The output includes VA= which indicates the virtual address for the operator, and the order in which each operator is executed. The query processor starts at VA=0. Generally, scan nodes (leaf nodes) are executed first.

Note The VA= in the showplan output is available for Adaptive Server version 15.0.2 ESD #2 and later. You will not see VA= in earlier versions of Adaptive Server.

- Adaptive Server uses a scan descriptor for each database object that is accessed during query execution. By default, each connection (or each worker process for parallel query plans) has 28 scan descriptors. If the query plan requires access to more than 28 database objects, auxiliary scan descriptors are allocated from a global pool. If the query plan uses auxiliary scan descriptors, this message is printed, showing the total number required:

Auxiliary scan descriptors required: N

- This message shows the total number of operators appearing in the query plan:

N operator(s) under root

- The next message shows the type of query for the query plan. For query plans, the query type is select, insert, delete, or update:

The type of query is SELECT.

- A final statement-level message is printed at the end of showplan output if Adaptive Server is configured to enable resource limits. The message displays the optimizer's total estimated cost of logical and physical I/O:

Total estimated I/O cost for statement N (at line M) :
X.

The following query, with showplan output, shows some of these messages:

```
use pubs2
go
set showplan on
go
select stores.stor_name, sales.ord_num
from stores, sales, salesdetail
where salesdetail.stor_id = sales.stor_id
and stores.stor_id = sales.stor_id
plan " ( m_join ( i_scan salesdetailind salesdetail)
( m_join ( i_scan salesind sales ) ( sort ( t_scan stores ) ) ) ) "
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.

STEP 1
The type of query is SELECT.

6 operator(s) under root

ROOT:EMIT Operator (VA = 6)

```
|MERGE JOIN Operator (Join Type: Inner Join) (VA = 5)
|  Using Worktable3 for internal storage.
|    Key Count: 1
|    Key Ordering: ASC
|
|  |SCAN Operator (VA = 0)
|  |  FROM TABLE
|  |  salesdetail
|  |  Index : salesdetailind
|  |  Forward Scan.
|  |  Positioning at index start.
|  |  Index contains all needed columns. Base table will not be read.
|  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  With LRU Buffer Replacement Strategy for index leaf pages.
|
|  |MERGE JOIN Operator (Join Type: Inner Join) (VA = 4)
|  |  Using Worktable2 for internal storage.
|  |    Key Count: 1
|  |    Key Ordering: ASC
|  |
|  |  |SCAN Operator (VA = 1)
|  |  |  FROM TABLE
|  |  |  sales
|  |  |  Table Scan.
```



```

Forward Scan.
Positioning at start of table.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.

SORT Operator (VA = 3)
Using Worktable1 for internal storage.

  SCAN Operator (VA = 2)
    FROM TABLE
    stores
    Table Scan.
    Forward Scan.
    Positioning at start of table.
    Using I/O Size 2 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.

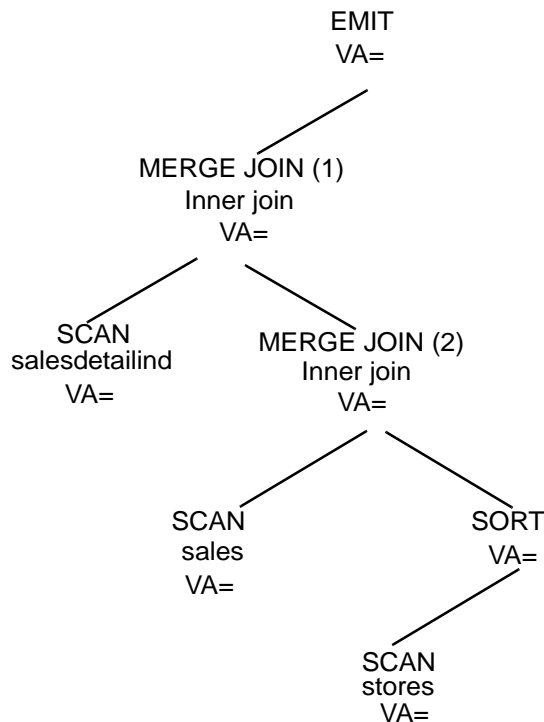
```

After the statement level output, the query plan appears. The showplan output of the query plan consists of two components:

- The names of the operators (some provide additional information) to show which operations are being executed in the query plan.
- Vertical bars (the “|” symbol) with indentation to show the shape of the query plan operator tree.

Query plan shape

The position of each operator in the tree determines its order of execution. Execution starts down the left-most branch of the tree and proceeds to the right. To illustrate execution, this section steps through the execution of the query plan for the example in the previous section. Figure 2-1 shows a graphical representation of the query plan.

Figure 2-1: Query plan

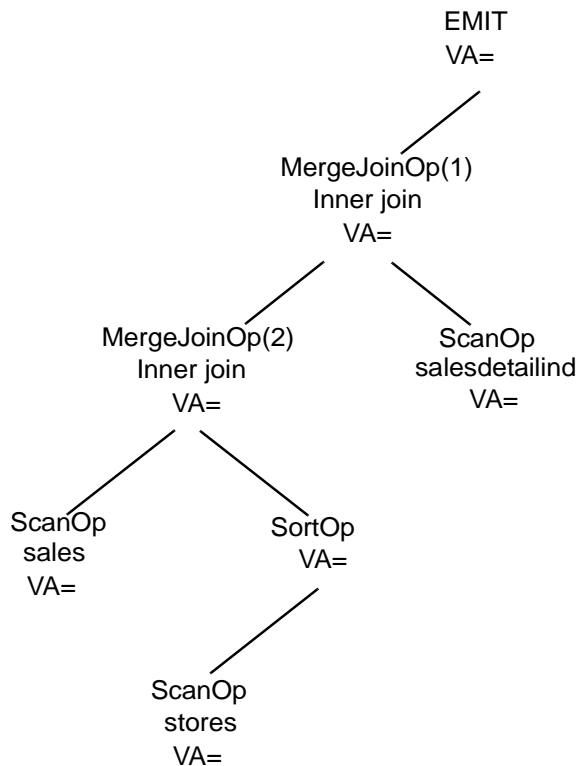
To generate a result row, the **EMIT** operator calls for a row from its child, the **MERGE JOIN** operator (1), which calls for a row from its left child, the **SCAN** operator for **salesdetailind**. When **EMIT** receives a row from its left child, **MERGE JOIN** operator (1) calls for a row from its right child, **MERGE JOIN** operator (2). **MERGE JOIN** operator (2) calls for a row from its left child, the **SCAN** operator for **sales**.

When it receives a row from its left child, **MERGE JOIN** operator (2) calls for a row from its right child, the **SCAN** operator. The **SCAN** operator is a data-blocking operator. That is, it needs all of its input rows before it can sort them, so the **SORT** operator keeps calling for rows from its child, the **SCAN** operator for **stores**, until all rows have been returned. Then the **SORT** operator sorts the rows and passes the first row to the **MERGE JOIN** operator (2).

The MERGE JOIN operator (2) keeps calling for rows from either the left or right child operators until it gets two rows that match on the joining keys. The matching row is then passed up to MERGE JOIN operator (1). MERGE JOIN operator (1) also calls for rows from its child operators until a match is found, which is then passed up to the EMIT operator to be returned to the client. In effect, the operators are processed using a left-deep postfix recursive strategy.

Figure 2-2 shows a graphical representation of an alternate query plan for the same example query. This query plan contains all of the same operators, but the shape of the tree is different.

Figure 2-2: Alternate query plan



The showplan output corresponding to the query plan in Figure 2-2 is:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
6 operator(s) under root
```

```
The type of query is SELECT.
```

ROOT:EMIT Operator

```

|MERGE JOIN Operator (Join Type: Inner Join)
|  Using Worktable3 for internal storage.
|    Key Count: 1
|    Key Ordering: ASC
|
|  |MERGE JOIN Operator (Join Type: Inner Join)
|  |  Using Worktable2 for internal storage.
|  |    Key Count: 1
|  |    Key Ordering: ASC
|  |
|  |  |SCAN Operator
|  |  |  FROM TABLE
|  |  |    sales
|  |  |    Table Scan.
|  |  |    Forward Scan.
|  |  |    Positioning at start of table.
|  |  |    Using I/O Size 2 Kbytes for data pages.
|  |  |    With LRU Buffer Replacement Strategy for data pages.
|  |
|  |  |SORT Operator
|  |  |  Using Worktable1 for internal storage.
|  |
|  |  |  |SCAN Operator
|  |  |  |  FROM TABLE
|  |  |  |    stores
|  |  |  |    Table Scan.
|  |  |  |    Forward Scan.
|  |  |  |    Positioning at start of table.
|  |  |  |    Using I/O Size 2 Kbytes for data pages.
|  |  |  |    With LRU Buffer Replacement Strategy for data pages.
|  |
|  |  |SCAN Operator
|  |  |  FROM TABLE
|  |  |    salesdetail
|  |  |    Index : salesdetailind
|  |  |    Forward Scan.
|  |  |    Positioning at index start.
|  |  |    Index contains all needed columns. Base table will not be read.
|  |  |    Using I/O Size 2 Kbytes for index leaf pages.
|  |  |    With LRU Buffer Replacement Strategy for index leaf pages.

```

The showplan output conveys the shape of the query plan by using indentation and the pipe (“|”) symbol to indicate which operators are under which and which ones are on the same or different branches of the tree. There are two rules to interpreting the tree shape:

- The pipe “|” symbols form a vertical line that starts at the operator’s name and continue down past all of the operators that are under it on the same branch.
- Child operators are indented to the left for each level of nesting.

Using these rules, the shape of the query plan in Figure 2-2 can be derived from the previous showplan output with the following steps:

- 1 The ROOT or EMIT operator is at the top of the query plan tree.
- 2 MERGE JOIN operator (1) is the left child of the ROOT. The vertical line that starts at MERGE JOIN operator (1) travels down the length of the entire output, so all of the other operators are below MERGE JOIN operator (1) and on the same branch.
- 3 The left child operator of the MERGE JOIN operator (1) is MERGE JOIN operator (2).
- 4 The vertical line that starts at MERGE JOIN operator (2) travels down past a SCAN, a SORT, and another SCAN operator before it ends. These operators are all nested as a subbranch under MERGE JOIN operator (2).
- 5 The first SCAN under MERGE JOIN operator (2) is its left child, the SCAN of the sales table.
- 6 The SORT operator is the right child of MERGE JOIN operator (2) and the SCAN of the stores table is the only child of the SORT operator.
- 7 Below the output for the SCAN of the stores table, several vertical lines end. This indicates that a branch of the tree has ended.
- 8 The next output is for the SCAN of the salesdetail table. It has the same indentation as MERGE JOIN operator (2), indicating that it is on the same level. In fact, this SCAN is the right child of MERGE JOIN operator (1).

Note Most operators are either unary or binary. That is, they have either a single child operator or two child operators directly beneath. Operators that have more than two child operators are called “nary”. Operators that have no children are leaf operators in the tree and are termed “nullary.”

Another way to get a graphical representation of the query plan is to use the command `set statistics plancost on`. See *Adaptive Server Reference Manual: Commands* for more information. This command is used to compare the estimated and actual costs in a query plan. It prints its output as a semigraphical tree representing the query plan tree. It is a very useful tool for diagnosing query performance problems.

Query plan operators

The query plan operators, and a description of each, are listed in Table 1-3 on page 23. This section contains additional messages that give more detailed information about each operator.

EMIT operator

The `EMIT` operator appears at the top of every query plan. `EMIT` is the root of the query plan tree and always has exactly one child operator. The `EMIT` operator routes the result rows of the query by sending them to the client (an application or another Adaptive Server instance) or by assigning values from the result row to local variables or fetch into variables.

SCAN operator

The `SCAN` operator reads rows into the query plan and makes them available for further processing by the other operators in the query plan. The `SCAN` operator is a leaf operator; that is, it never has any child operators. The `SCAN` operator can read rows from multiple sources, so the `showplan` message identifying it is always followed by a `FROM` message to identify what kind of `SCAN` is being performed. The `FROM` messages are: `FROM CACHE`, `FROM OR`, `FROM LIST`, and `FROM TABLE`.

FROM cache message

This message shows that a `CACHE SCAN` operator is reading a single-row in-memory table.

FROM or LIST

An OR list has as many as N rows; one for each distinct OR or IN value specified in the query.

The first message shows that an OR scan is reading rows from an in-memory table that contains values from an IN list or multiple or clauses on the same column. The OR list appears only in query plans that use the special or strategy for in lists. The second message shows the maximum number of rows (N) that the in-memory table can have. Since OR list eliminates duplicate values when filling the in-memory table, N may be less than the number of values appearing in the SQL statement. As an example, the following query generates a query plan with the special or strategy and an OR list:

```
select s.id from sysobjects s where s.id in (1, 0, 1, 2, 3)
go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

4 operator(s) under root

ROOT:EMIT Operator (VA = 4)

```
| NESTED LOOP JOIN Operator (VA = 3) (Join Type: Inner Join)
|
|   | SCAN Operator (VA = 2)
|   |   FROM OR List
|   |   OR List has up to 5 rows of OR/IN values.
|
|   | RESTRICT Operator (VA = 2) (0) (0) (0) (8) (0)
|   |   | SCAN Operator (VA = 1)
|   |   |   FROM TABLE
|   |   |   sysobjects
|   |   |   s
|   |   |   Using Clustered Index.
|   |   |   Index : csysobjects
|   |   |   Forward Scan.
|   |   |   Positioning by key.
|   |   |   Index contains all needed columns. Base table will not be read.
|   |   |   Keys are:
|   |   |   id ASC
|   |   |   Using I/O Size 2 Kbytes for index leaf pages.
|   |   |   With LRU Buffer Replacement Strategy for index leaf pages.
```

This example has five values in the `IN` list, but only four are distinct, so the `OR` list puts only the four distinct values in its in-memory table. In the example query plan, the `OR` list is the left-child operator of the `NESTED LOOP JOIN` operator and a `SCAN` operator is the right child of the `NESTED LOOP JOIN` operator. When this plan executes, the `NESTED LOOP JOIN` operator calls the `or` command to return a row from its in-memory table, then the `NESTED LOOP JOIN` operator calls on the `SCAN` operator to find all matching rows (one at a time), using the clustered index for lookup. This example query plan is much more efficient than reading all of the rows of `sysobjects` and comparing the value of `sysobjects.id` in each row to the five values in the `IN` list.

FROM TABLE

`FROM TABLE` shows that a `PARTITION SCAN` operator is reading a database table. A second message gives the table name, and, if there is a correlation name, it is printed on the next line. Under the `FROM TABLE` message in the previous example output, `sysobjects` is the table name and `s` is the correlation name. The previous example also shows additional messages under the `FROM TABLE` message. These messages give more information about how the `PARTITION SCAN` operator is directing the access layer of Adaptive Server to get the rows from the table being scanned.

The messages below indicate whether the scan is a table scan or an index scan:

- `Table Scan` – the rows are fetched by reading the pages of the table.
- `Using Clustered Index` – a clustered index is used to fetch the rows of the table.
- `Index: indexname` – an index is used to fetch the table rows. If this message is not preceded by “`using clustered index`,” a nonclustered index is used. *indexname* is the name of the index that will be used.

These messages indicate the direction of a table or index scan. The scan direction depends on the ordering specified when the indexes were created and the order specified for columns in the `order by` clause or other useful orderings that can be exploited by operators further up in the query plan (for example, a sorted ordering for a merge-join strategy).

Backward scans can be used when the `order by` clause contains the ascending or descending qualifiers on index keys, exactly opposite of those in the `create index` clause.

Forward scan

Backward scan

The scan-direction messages are followed by positioning messages, which describe how access to a table or to the leaf level of an index takes place:

- Positioning at start of table – a table scan that starts at the first row of the table and goes forward.
- Positioning at end of table – a table scan that starts at the last row of the table and goes backward.
- Positioning by key – the index is used to position the scan at the first qualifying row.
- Positioning at index start/positioning at index end – these messages are similar to the corresponding messages for table scans, except that an index is being scanned instead of a table.

If the scan can be limited due to the nature of the query, the following messages describe how:

- Scanning only the last page of the table – appears when the scan uses an index and is searching for the maximum value for scalar aggregation. If the index is on the column whose maximum is sought, and the index values are in ascending order, the maximum value will be on the last page.
- Scanning only up to the first qualifying row – appears when the scan uses an index and is searching for the minimum value for scalar aggregation.

Note If the index key is sorted in descending order, the above messages for minimum and maximum aggregates are reversed.

In some cases, the index being scanned contains all of the columns of the table that are needed in the query. In such a case, this message is printed:

```
Index contains all needed columns. Base table will not  
be read.
```

If an index contains all the columns needed by the query, the optimizer may choose an `Index Scan` over a `Table Scan` even though there are no useful keys on the index columns. The amount of I/O required to read the index can be significantly less than that required to read the base table. Index scans that do not require base table pages to be read are called *covered index scans*.

If an index scan is using keys to position the scan, this message prints:

Keys are:
Key <ASC/DESC>

This message shows the names of the columns used as keys (each key on its own output line) and shows the index ordering on that key: ASC for ascending and DESC for descending.

After the messages that describe the type of access being used by the scan operator, messages about the I/O sizes and buffer cache strategy are printed.

I/O size messages

The I/O messages are:

Using I/O size *N* Kbytes for data pages.

Using I/O size *N* Kbytes for index leaf pages.

These messages report the I/O sizes used in the query. Possible I/O sizes are 2, 4, 8, and 16 kilobytes.

If the table, index, or database used in the query uses a data cache with large I/O pools, the optimizer can choose large I/O. It can choose to use one I/O size for reading index leaf pages, and a different size for data pages. The choice depends on the pool size available in the cache, the number of pages to be read, the cache bindings for the objects, and the cluster ratio for the table or index pages.

Either (or both) of these messages can appear in the showplan output for a SCAN operator. For a table scan, only the first message is printed; for a covered index scan, only the second message is printed. For an Index Scan that requires base table access, both messages are printed.

After each I/O size message, a cache strategy message is printed:

With <LRU/MRU> Buffer Replacement Strategy for data pages.

With <LRU/MRU> Buffer Replacement Strategy for index leaf pages.

In an LRU replacement strategy, the most recently accessed pages are positioned in the cache to be retained as long as possible. In an MRU Replacement Strategy, the most recently accessed pages are positioned in the cache for quick replacement.

Sample I/O and cache messages are shown in the following query:

```
use pubs2
go
```

```
set showplan on
go
select au_fname, au_lname, au_id from authors
where au_lname = "Williams"
go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator (VA = 1)

```
|SCAN Operator (VA = 0)
|  FROM TABLE
|  authors
|  Index : aunmind
|  Forward Scan.
|  Positioning by key.
|  Keys are:
|    au_lname ASC
|  Using I/O Size 2 Kbytes for index leaf pages.
|  With LRU Buffer Replacement Strategy for index leaf pages.
|  Using I/O Size 2 Kbytes for data pages.
|  With LRU Buffer Replacement Strategy for data pages.
```

The SCAN operator of the authors table uses the index aunmind, but must also read the base table pages to get all of the required columns from authors. In this example, there are two I/O size messages, each followed by the corresponding buffer replacement message.

There are two kinds of table SCAN operators that have their own messages—the RID SCAN and the LOG SCAN.

RID scan

The Positioning by Row Identifier (RID) scan is found only in query plans that use the second or strategy that the optimizer can choose, the general or strategy. The general or strategy may be used when multiple or clauses are present on different columns. An example of a query for which the optimizer can choose a general or strategy and its showplan output is:

```
use pubs2
go
set showplan on
```

```
go
select id from sysobjects where id = 4 or name = 'foo'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator (VA = 6)

```
|RID JOIN Operator (VA = 5)
|  Using Worktable2 for internal storage.
|
|  |HASH UNION Operator has 2 children.
|  |  Using Worktable1 for internal storage.
|  |
|  |  |SCAN Operator (VA = 0)
|  |  |  FROM TABLE
|  |  |    sysobjects
|  |  |    Using Clustered Index.
|  |  |    Index : csysobjects
|  |  |    Forward Scan.
|  |  |    Positioning by key.
|  |  |    Index contains all needed columns. Base table will not be read.
|  |  |    Keys are:
|  |  |      id ASC
|  |  |    Using I/O Size 2 Kbytes for index leaf pages.
|  |  |    With LRU Buffer Replacement Strategy for index leaf pages.
|  |
|  |  |SCAN Operator (VA = 1)
|  |  |  FROM TABLE
|  |  |    sysobjects
|  |  |    Index : ncsysobjects
|  |  |    Forward Scan.
|  |  |    Positioning by key.
|  |  |    Index contains all needed columns. Base table will not be read.
|  |  |    Keys are:
|  |  |      name ASC
|  |  |    Using I/O Size 2 Kbytes for index leaf pages.
|  |  |    With LRU Buffer Replacement Strategy for index leaf pages.
|  |
|  |RESTRICT Operator (VA = 4) (0) (0) (0) (11) (0)
|  |
|  |  |SCAN Operator (VA = 3)
|  |  |  FROM TABLE
```

```

|   |   | sysobjects
|   |   | Using Dynamic Index.
|   |   | Forward Scan.
|   |   | Positioning by Row Identifier (RID).
|   |   | Using I/O Size 2 Kbytes for data pages.
|   |   | With LRU Buffer Replacement Strategy for data pages.

```

In this example, the where clause contains two disjunctions, each on a different column (id and name). There are indexes on each of these columns (csysobjects and ncsysobjects), so the optimizer chose a query plan that uses an index scan to find all rows whose id column is 4 and another index scan to find all rows whose name is “foo.”

Since it is possible that a single row has both an ID of 4 and a name of “foo,” that row would appear twice in the result set. To eliminate these duplicate rows, the index scans return only the row identifiers (RIDs) of the qualifying rows. The two streams of RIDs are concatenated by the `HASH UNION` operator, which also removes any duplicate RIDs.

]The stream of unique RIDs is passed to the `RID JOIN` operator. The `rid join` operator creates a worktable and fills it with a single-column row with each RID. The `RID JOIN` operator then passes its worktable of RIDs to the `RID SCAN` operator. The `RID SCAN` operator passes the worktable to the access layer, where it is treated as a keyless nonclustered index and the rows corresponding to the RIDs are fetched and returned.

The last `SCAN` in the showplan output is the `RID SCAN`. As can be seen from the example output, the `RID SCAN` output contains many of the messages already discussed above, but it also contains two messages that are printed only for the `RID SCAN`:

- `Using Dynamic Index`—indicates the `SCAN` is using the worktable with RIDs that was built during execution by the `RID JOIN` operator as an index to locate the matching rows.
- `Positioning by Row Identifier (RID)`—indicates the rows are being located directly by the RID.

Log Scan

`Log Scan` appears only in triggers that access inserted or deleted tables. These tables are dynamically built by scanning the transaction log when the trigger is executed. Triggers are executed only after insert, delete, or update queries modify a table with a trigger defined on it for the specific query type. The following example is a delete query on the titles table, which has a delete trigger called `deltitle` defined on it:

```
use pubs2
go
set showplan on
go
delete from titles where title_id = 'xxxx'
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is DELETE.

2 operator(s) under root

|ROOT:EMIT Operator (VA = 2)

|DELETE Operator (VA = 1)

| The update mode is direct.

| |SCAN Operator (VA = 0)

| | FROM TABLE

| | titles

| | Using Clustered Index.

| | Index : titleidind

| | Forward Scan.

| | Positioning by key.

| | Keys are:

| | title_id ASC

| | Using I/O Size 2 Kbytes for data pages.

| | With LRU Buffer Replacement Strategy for data pages.

| TO TABLE

| titles

| Using I/O Size 2 Kbytes for data pages.

The showplan output up to this point is for the actual delete query. The output below is for the trigger, deltitle.

QUERY PLAN FOR STATEMENT 1 (at line 5).

STEP 1

The type of query is COND.

6 operator(s) under root

ROOT:EMIT Operator (VA = 6)

```

|RESTRICT Operator (VA = 5) (0) (0) (0) (5) (0)
|
|  |SCALAR AGGREGATE Operator (VA = 4)
|  |  Evaluate Ungrouped COUNT AGGREGATE.
|  |
|  |  |MERGE JOIN Operator (Join Type: Inner Join) (VA = 3)
|  |  |  Using Worktable2 for internal storage.
|  |  |    Key Count: 1
|  |  |    Key Ordering: ASC
|  |  |
|  |  |  |SORT Operator (VA = 1)
|  |  |  |  Using Worktable1 for internal storage.
|  |  |  |
|  |  |  |  |SCAN Operator (VA = 0)
|  |  |  |  |  FROM TABLE
|  |  |  |  |  titles
|  |  |  |  |  Log Scan.
|  |  |  |  |  Forward Scan.
|  |  |  |  |  Positioning at start of table.
|  |  |  |  |  Using I/O Size 2 Kbytes for data pages.
|  |  |  |  |  With MRU Buffer Replacement Strategy for data pages.
|  |  |  |
|  |  |  |SCAN Operator (VA = 2)
|  |  |  |  FROM TABLE
|  |  |  |  salesdetail
|  |  |  |  Index : titleidind
|  |  |  |  Forward Scan.
|  |  |  |  Positioning at index start.
|  |  |  |  Index contains all needed columns. Base table will not be
|  |  |  |  read.
|  |  |  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  |  |  With LRU Buffer Replacement Strategy for index leaf pages.

```

QUERY PLAN FOR STATEMENT 2 (at line 8).

STEP 1

The type of query is ROLLBACK TRANSACTION.

QUERY PLAN FOR STATEMENT 3 (at line 9).

STEP 1

The type of query is PRINT.

QUERY PLAN FOR STATEMENT 4 (at line 0).

STEP 1

The type of query is GOTO.

The procedure that defines the `deltitle` trigger consists of four SQL statements. Use `sp_helptext deltitle` to display the text of `deltitle`. The first statement in `deltitle` has been compiled into a query plan, the other three statements are compiled into legacy query plans and are executed by the procedural execution engine, not the query execution engine.

The showplan output for the `SCAN` operator for the `titles` table indicates that it is doing a scan of the log by printing `Log Scan`.

DELETE, INSERT, and UPDATE operators

The `DELETE`, `INSERT`, and `UPDATE` operators usually have only one child operator. However, they can have as many as two additional child operators to enforce referential integrity constraints and to deallocate text data in the case of `alter table drop` of a text column.

These operators modify data by inserting, deleting, or updating rows belonging to a target table.

Child operators of DML operators can be `SCAN` operators, `JOIN` operators, or any data streaming operator.

The data modification can be done using different update modes, as specified by this message:

The Update Mode is *<Update Mode>*.

The table update mode may be `direct`, `deferred`, `deferred for an index`, or `deferred for a variable column`. The update mode for a `worktable` is always `direct`.

The target table for the data modification is displayed in this message:

TO TABLE
<Table Name>

Also displayed is the I/O size used for the data modification:

Using I/O Size *<N>* Kbytes for data pages.

The next example uses the `DELETE` operator:

```
use pubs2
go
set showplan on
go
delete from authors where postalcode = '90210'
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```


STEP 1

The type of query is DELETE.

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

```
|DELETE Operator (VA = 1)
|  The update mode is direct.
|
|  |SCAN Operator (VA = 0)
|  |  FROM TABLE
|  |  authors
|  |  Table Scan.
|  |  Forward Scan.
|  |  Positioning at start of table.
|  |  Using I/O Size 4 Kbytes for data pages.
|  |  With LRU Buffer Replacement Strategy for data pages.
|
|  TO TABLE
|  authors
|  Using I/O Size 4 Kbytes for data pages.
```

TEXT DELETE operator

Another type of query plan where DELETE, INSERT, and UPDATE operator can have more than one child operator is the alter table drop textcol command, where textcol is the name of a column whose datatype is text, image, or unitext. This version of command used the TEXT DELETE operator in its query plan. For example:

```
use tempdb
go
create table t1 (c1 int, c2 text, c3 text)
go
set showplan on
go
alter table t1 drop c2
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Optimized using the Abstract Plan in the PLAN clause.

STEP 1

The type of query is ALTER TABLE.

5 operator(s) under root

ROOT:EMIT Operator (VA = 5)

```

| INSERT Operator (VA = 52)
|   The update mode is direct.
|
|   | RESTRICT Operator (VA = 1) (0) (0) (3) (0) (0)
|   |
|   |   | SCAN Operator (VA = 0)
|   |   |   FROM TABLE
|   |   |   t1
|   |   |   Table Scan.
|   |   |   Forward Scan.
|   |   |   Positioning at start of table.
|   |   |   Using I/O Size 2 Kbytes for data pages.
|   |   |   With LRU Buffer Replacement Strategy for data pages.
|   |
|   | TEXT DELETE Operator
|   |   The update mode is direct.
|   |
|   |   | SCAN Operator (VA = 3)
|   |   |   FROM TABLE
|   |   |   t1
|   |   |   Table Scan.
|   |   |   Forward Scan.
|   |   |   Positioning at start of table.
|   |   |   Using I/O Size 2 Kbytes for data pages.
|   |   |   With LRU Buffer Replacement Strategy for data pages.
|   |
| TO TABLE
| #syb__altab
| Using I/O Size 2 Kbytes for data pages.

```

One of the two text columns in t1 is dropped, using the alter table command. The showplan output has the appearance of a select into query plan because alter table internally generated a select into query plan.

The INSERT operator calls on its left child operator, the SCAN of t1, to read the rows of t1, and builds new rows with only the c1 and c3 columns inserted into #syb__altab. When all the new rows have been inserted into #syb__altab, the INSERT operator calls on its right child, the TEXT DELETE operator, to delete the text page chains for the c2 columns that have been dropped from t1.

Postprocessing replaces the original pages of t1 with those of #syb__altab to complete the alter table command.

The `TEXT DELETE` operator appears only in alter table commands that drop some, but not all text columns of a table, and it always appears as the right child of an `INSERT` operator.

The `TEXT DELETE` operator displays the update mode message, exactly like the `INSERT`, `UPDATE`, and `DELETE` operators.

Query plans for referential integrity enforcement

When the `INSERT`, `UPDATE`, and `DELETE` operators are used on a table that has one or more referential integrity constraints, the showplan output also shows the `DIRECT RI FILTER` and `DEFERRED RI FILTER` child operators of the DML operator. The type of referential integrity constraint determines whether one or both of these operators are present.

The following example is for an insert into the titles table of the pubs3 database. This table has a column called `pub_id` that references the `pub_id` column of the publishers table. The referential integrity constraint on `titles.pub_id` requires that every value that is inserted into `titles.pub_id` must have a corresponding value in `publishers.pub_id`.

The query and its query plan are:

```
use pubs3
go
set showplan on
insert into titles values ("AB1234", "Abcdefg", "test", "9999", 9.95, 1000.00,
10, null, getdate(),1)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
```

```
The type of query is INSERT.
```

```
4 operator(s) under root
```

```
ROOT:EMIT Operator (VA = 3)
```

```
| INSERT Operator (VA = 2)
|   The update mode is direct.
|
|   | SCAN Operator (VA = 1)
|   |   FROM CACHE
|   |
|   | DIRECT RI FILTER Operator has 1 children.
```

```

|      |      | SCAN Operator (VA = 0)
|      |      |   FROM TABLE
|      |      |   publishers
|      |      |   Index : publishers_6240022232
|      |      |   Forward Scan.
|      |      |   Positioning by key.
|      |      |   Index contains all needed columns. Base table will not be
|      |      |   read.
|      |      |   Keys are:
|      |      |       pub_id ASC
|      |      |   Using I/O Size 2 Kbytes for index leaf pages.
|      |      |   With LRU Buffer Replacement Strategy for index leaf pages.
|
|      |      | TO TABLE
|      |      | titles
|      |      | Using I/O Size 2 Kbytes for data pages.

```

In the query plan, the `INSERT` operator's left child operator is a `CACHE SCAN`, which returns the row of values to be inserted into titles. The `INSERT` operator's right child is a `DIRECT RI FILTER` operator.

The `DIRECT RI FILTER` operator executes a scan of the publishers table to find a row with a value of `pub_id` that matches the value of `pub_id` to be inserted into titles. If a matching row is found, the `DIRECT RI FILTER` operator allows the insert to proceed, but if a matching value of `pub_id` is not found in publishers, the `DIRECT RI FILTER` operator aborts the command.

In this example, the `DIRECT RI FILTER` can check and enforce the referential integrity constraint on titles for each row that is inserted, as it is inserted.

The next example shows a `DIRECT RI FILTER` operating in a different mode, together with a `DEFERRED RI FILTER` operator:

```

use pubs3
go
set showplan on
go
update publishers set pub_id = '0001'

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is UPDATE.

13 operator(s) under root

ROOT:EMIT Operator (VA = 13)

```

```

| UPDATE Operator (VA = 1)
|   The update mode is deferred_index.
|
|   | SCAN Operator (VA = 0)
|   |   FROM TABLE
|   |   publishers
|   |   Table Scan.
|   |   Forward Scan.
|   |   Positioning at start of table.
|   |   Using I/O Size 2 Kbytes for data pages.
|   |   With LRU Buffer Replacement Strategy for data pages.
|
|   | DIRECT RI FILTER Operator (VA = 7) has 1 children.
|   |
|   |   | INSERT Operator (VA = 6)
|   |   |   The update mode is direct.
|   |   |
|   |   |   | SQFILTER Operator (VA = 5) has 2 children.
|   |   |   |
|   |   |   |   | SCAN Operator (VA = 2)
|   |   |   |   |   FROM CACHE
|   |   |   |
|   |   |   |   Run subquery 1 (at nesting level 0).
|   |   |   |
|   |   |   |   QUERY PLAN FOR SUBQUERY 1 (at nesting level 0 and at
|   |   |   |   line 0).
|   |   |   |
|   |   |   |   Non-correlated Subquery.
|   |   |   |   Subquery under an EXISTS predicate.
|   |   |   |
|   |   |   |   | SCALAR AGGREGATE Operator (VA = 4)
|   |   |   |   |   Evaluate Ungrouped ANY AGGREGATE.
|   |   |   |   |   Scanning only up to the first qualifying row.
|   |   |   |   |
|   |   |   |   |   | SCAN Operator (VA = 3)
|   |   |   |   |   |   FROM TABLE
|   |   |   |   |   |   titles
|   |   |   |   |   |   Table Scan.
|   |   |   |   |   |   Forward Scan.
|   |   |   |   |   |   Positioning at start of table.
|   |   |   |   |   |   Using I/O Size 2 Kbytes for data pages.
|   |   |   |   |   |   With LRU Buffer Replacement strategy for data
|   |   |   |   |   |   pages.
|   |   |   |   |
|   |   |   |   END OF QUERY PLAN FOR SUBQUERY 1.

```


The referential integrity constraint on titles requires that for every value of titles.pub_id there must exist a value of publishers.pub_id. However, this example query is changing the values of publisher.pub_id, so a check must be made to maintain the referential integrity constraint.

The example query can change the value of publishers.pub_id for several rows in publishers, so a check to make sure that all of the values of titles.pub_id still exist in publisher.pub_id cannot be done until all rows of publishers have been processed.

This example calls for deferred referential integrity checking: as each row of publishers is read, the UPDATE operator calls upon the DIRECT RI FILTER operator to search titles for a row with the same value of pub_id as the value that is about to be changed. If a row is found, it indicates that this value of pub_id must still exist in publishers to maintain the referential integrity constraint on titles, so the value of pub_id is inserted into WorkTable1.

After all of the rows of publishers have been updated, the UPDATE operator calls upon the DEFERRED RI FILTER operator to execute its subquery to verify that all of the values in Worktable1 still exist in publishers. The left child operator of the DEFERRED RI FILTER is a SCAN which reads the rows from Worktable1. The right child is a SQFILTER operator that executes an existence subquery to check for a matching value in publishers. If a matching value is not found, the command is aborted.

The examples in this section used simple referential integrity constraints, between only two tables. Adaptive Server allows up to 192 constraints per table, so it can generate much more complex query plans. When multiple constraints must be enforced, there is still only a single DIRECT RI FILTER or DEFERRED RI FILTER operator in the query plan, but these operators can have multiple subplans, one for each constraint that must be enforced.

JOIN operators

Adaptive Server provides four primary JOIN operator strategies: NESTED LOOP JOIN, MERGE JOIN, HASH JOIN, and NARY NESTED LOOP JOIN, which is a variant of NESTED LOOP JOIN. In versions earlier than 15.0, NESTED LOOP JOIN was the primary JOIN strategy. MERGE JOIN was also available, but was, by default, not enabled.

Each JOIN operator is described in further detail below, including a general description of the each algorithm. These descriptions give a high-level overview of the processing required for each JOIN strategy.

NESTED LOOP JOIN

NESTED LOOP JOIN, the simplest join strategy, is a binary operator with the left child forming the outer data stream and the right child forming the inner data stream.

For every row from the outer data stream, the inner data stream is opened. Often, the right child is a scan operator. Opening the inner data stream effectively positions the scan on the first row that qualifies all of the searchable arguments.

The qualifying row is returned to the NESTED LOOP JOIN's parent operator. Subsequent calls to the join operator continue to return qualifying rows from the inner stream.

After the last qualifying row from the inner stream is returned for the current outer row, the inner stream is closed. A call is made to get the next qualifying row from the outer stream. The values from this row provide the searchable arguments used to open and position the scan on the inner stream. This process continues until the NESTED LOOP JOIN's left child returns End Of Scan.

```
-- Collect all of the title ids for books written by "Bloom".
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"
go
```

QUERY PLAN FOR STATEMENT 1 (at line 2).

STEP 1

The type of query is SELECT.

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

```
| NESTED LOOP JOIN Operator (Join Type: Inner Join)
|
| | SCAN Operator (VA = 0)
| |   FROM TABLE
| |   authors
| |   a
| |   Index : aunmind
| |   Forward Scan.
| |   Positioning by key.
| |   Keys are:
```



```

| au_lname ASC
| Using I/O Size 2 Kbytes for index leaf pages.
| With LRU Buffer Replacement Strategy for index leaf pages.
| Using I/O Size 2 Kbytes for data pages.
| With LRU Buffer Replacement Strategy for data pages.
|
| SCAN Operator (VA = 1)
| FROM TABLE
| titleauthor
| ta
| Using Clustered Index.
| Index : taind
| Forward Scan.
| Positioning by key.
| Keys are:
|   au_id ASC
| Using I/O Size 2 Kbytes for data pages.
| With LRU Buffer Replacement Strategy for data pages.

```

The authors table is joined with the titleauthor table. A NESTED LOOP JOIN strategy has been chosen. The NESTED LOOP JOIN operator's type is "Inner Join." First, the authors table is opened and positioned on the first row (using the aunmind index) containing an l_name value of "Bloom." Then, the titleauthor table is opened and positioned on the first row with an au_id equal to the au_id value of the current authors' row using the clustered index "taind." If there is no useful index for lookups on the inner stream, the optimizer may generate a reformatting strategy.

Generally, a NESTED LOOP JOIN strategy is effective when there is a useful index available for qualifying the join predicates on the inner stream.

MERGE JOIN

The `MERGE JOIN` operator is a binary operator. The left and right children are the outer and inner data streams, respectively. Both data streams must be sorted on the `MERGE JOIN`'s key values.

First, a row from the outer stream is fetched. This initializes the MERGE JOIN's join key values. Then, rows from the inner stream are fetched until a row with key values that match or are greater than (less than if key column is descending) is encountered. If the join key matches, the qualifying row is passed on for additional processing, and a subsequent next call to the MERGE JOIN operator continues fetching from the currently active stream.

If the new values are greater than the current comparison key, these values are used as the new comparison join key while fetching rows from the other stream. This process continues until one of the data streams is exhausted.

Generally, the `MERGE JOIN` strategy is effective when a scan of the data streams requires that most of the rows must be processed, and that, if any of the input streams are large, they are already sorted on the join keys.

```
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"
go
```

QUERY PLAN FOR STATEMENT 1 (at line 2).

```
STEP 1
    The type of query is EXECUTE.
    Executing a newly cached statement.
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

```
STEP 1
The type of query is SELECT.
```

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

```
|MERGE JOIN Operator (Join Type: Inner Join)
|  Using Worktable2 for internal storage.
|  Key Count: 1
|  Key Ordering: ASC
|
|  |SORT Operator
|  |  Using Worktable1 for internal storage.
|  |
|  |  |SCAN Operator
|  |  |  FROM TABLE
|  |  |  authors
|  |  |  a
|  |  |  Index : aunmind
|  |  |  Forward Scan.
|  |  |  Positioning by key.
|  |  |  Keys are:
|  |  |    au_lname ASC
|  |  |  Using I/O Size 2 Kbytes for index leaf pages.
```

```

|      |      | With LRU Buffer Replacement Strategy for index leaf pages.
|      |      | Using I/O Size 2 Kbytes for data pages.
|      |      | With LRU Buffer Replacement Strategy for data pages.
|
| SCAN Operator
| FROM TABLE
| titleauthor
| ta
| Index : auidind
| Forward Scan.
| Positioning at index start.
| Using I/O Size 2 Kbytes for index leaf pages.
| With LRU Buffer Replacement Strategy for index leaf pages.
| Using I/O Size 2 Kbytes for data pages.
| With LRU Buffer Replacement Strategy for data pages.

```

In this example, a sort operator is the left child, or outer stream. The data source for the sort operator is the authors table. The sort operator is required because the authors table has no index on `au_id` that would otherwise provide the necessary sorted order. A scan of the `titleauthor` table is the right child/inner stream. The scan uses the `auidind` index, which provides the necessary ordering for the `MERGE JOIN` strategy.

A row is fetched from the outer stream (the authors table is the original source) to establish an initial join key comparison value. Then rows are fetched from the `titleauthor` table until a row with a join key equal to or greater than the comparison key is found.

Inner stream rows with matching keys are stored in a cache in case they need to be refetched. These rows are refetched when the outer stream contains duplicate keys. When a `titleauthor.au_id` value that is greater than the current join key comparison value is fetched, the `MERGE JOIN` operator starts fetching from the outer stream until a join key value equal to or greater than the current `titleauthor.au_id` value is found. The scan of the inner stream resumes at that point.

The `MERGE JOIN` operator's showplan output contains a message indicating the worktable to be used for the inner stream's backing store. The worktable is written to if the inner rows with duplicate join keys no longer fits in cached memory. The width of a cached row is limited to 64 kilobytes.

HASH JOIN

The `HASH JOIN` operator is a binary operator. The left child generates the build input stream. The right child generates the probe input stream. The build set is generated by completely draining the build input stream when the first row is requested from the `HASH JOIN` operator. Every row is read from the input stream and hashed into an appropriate bucket using the hash key.

If there is not enough memory to hold the entire build set, then a portion of it spills to disk. This portion is referred to as a *hash partition* and should not be confused with table partitions. A hash partition consists of a collection of hash buckets. After the entire left child's stream has been drained, the probe input is read.

Each row from the probe set is hashed. A lookup is done in the corresponding build bucket to check for rows with matching hash keys. This occurs if the build set's bucket is memory resident. If it has been spilled, the probe row is written to the corresponding spilled probe partition. When a probe row's key matches a build row's key, then the necessary projection of the two row's columns is passed up for additional processing.

Spilled partitions are processed in subsequent recursive passes of the `HASH JOIN` algorithm. New hash seeds are used in each pass so that the data is redistributed across different hash buckets. This recursive processing continues until the last spilled partition is completely memory resident. When a hash partition from the build set contains many duplicates, the `HASH JOIN` operator reverts back to `NESTED LOOP JOIN` processing.

Generally, the `HASH JOIN` strategy is good in cases where most of the rows from the source sets must be processed and there are no inherent useful orderings on the join keys or there are no interesting orderings that can be promoted to calling operators (for example, an order by clause on the join key). `HASH JOINS` perform particularly well if one of the data sets is small enough to be memory resident. In this case, no spilling occurs and no I/O is needed to perform that `HASH JOIN` algorithm.

```
select ta.title_id
from titleauthor ta, authors a
where a.au_id = ta.au_id
and au_lname = "Bloom"
```

QUERY PLAN FOR STATEMENT 1 (at line 2).

3 operator(s) under root

The type of query is `SELECT`.

ROOT:EMIT Operator

```

|HASH JOIN Operator (Join Type: Inner Join)
|  Using Worktable1 for internal storage.
|
|  |SCAN Operator
|  |  FROM TABLE
|  |  authors
|  |  a
|  |  Index : aunmind
|  |  Forward Scan.
|  |  Positioning by key.
|  |  Keys are:
|  |    au_lname ASC
|  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  With LRU Buffer Replacement Strategy for index leaf pages.
|  |  Using I/O Size 2 Kbytes for data pages.
|  |  With LRU Buffer Replacement Strategy for data pages.
|
|  |SCAN Operator
|  |  FROM TABLE
|  |  titleauthor
|  |  ta
|  |  Index : auidind
|  |  Forward Scan.
|  |  Positioning at index start.
|  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  With LRU Buffer Replacement Strategy for index leaf pages.
|  |  Using I/O Size 2 Kbytes for data pages.
|  |  With LRU Buffer Replacement Strategy for data pages.

```

In this example, the source of the build input stream is an index scan of `author.aunmind`.

Only rows with an `au_lname` value of “Bloom” are returned from this scan. These rows are then hashed on their `au_id` value and placed into their corresponding hash bucket. After the initial build phase is completed, the probe stream is opened and scanned. Each row from the source index, `titleauthor.auidind`, is hashed on the `au_id` column. The resulting hash value is used to determine which bucket in the build set should be searched for matching hash keys. Each row from the build set’s hash bucket is compared to the probe row’s hash key for equality. If the row matches, the `titleauthor.au_id` column is returned to the EMIT operator.

The `HASH JOIN` operator's showplan output contains a message indicating the worktable to be used for the spilled partition's backing store. The input row width is limited to 64 kilobytes.

NARY NESTED LOOP JOIN operator

The `NARY NESTED LOOP JOIN` strategy is never evaluated or chosen by the optimizer. It is an operator that is constructed during code generation. If the compiler finds series of two or more left-deep `NESTED LOOP JOINS`, it attempts to transform them into a `NARY NESTED LOOP JOIN` operator. Two additional requirements allow for transformation scan; each `NESTED LOOP JOIN` operator has an "inner join" type and the right child of each `NESTED LOOP JOIN` is a `SCAN` operator. A `RESTRICT` operator is permitted above the `SCAN` operator.

`NARY NESTED LOOP JOIN` execution has a performance benefit over the execution of a series of `NESTED LOOP JOIN` operators. The example below demonstrates a fundamental difference between the two methods of execution.

With a series of `NESTED LOOP JOIN`, a scan may eliminate rows based on searchable argument values initialized by an earlier scan. That scan may not be the one that immediately preceded the failing scan. With a series of `NESTED LOOP JOINS`, the previous scan would be completely drained although it has no effect on the failing scan. This could result in a significant amount of needless I/O. With `NARY NESTED LOOP JOINS`, the next row fetched comes from the scan that produced the failing searchable argument value, which is far more efficient.

```
select a.au_id, au_fname, au_lname
from titles t, titleauthor ta, authors a
where a.au_id = ta.au_id
and ta.title_id = t.title_id
and a.au_id = t.title_id
and au_lname = "Bloom"
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

STEP 1

The type of query is `SELECT`.

4 operator(s) under root

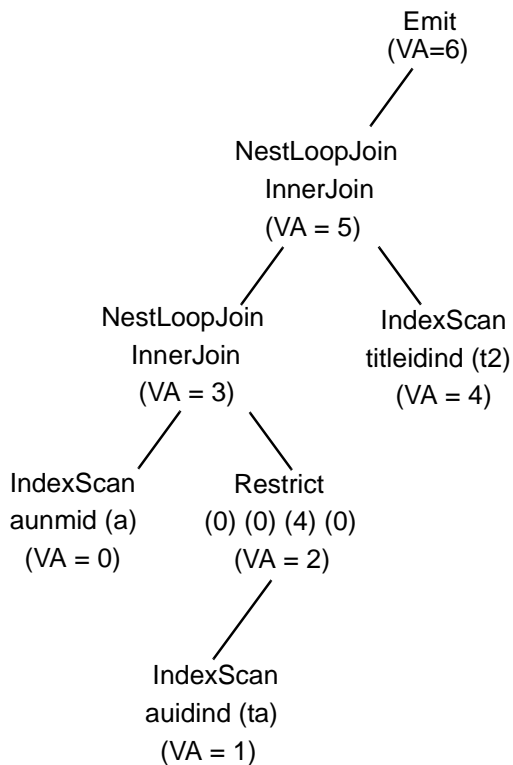
```
| ROOT:EMIT Operator (VA = 4)
|
|   | N-ARY NESTED LOOP JOIN Operator (VA = 3) has 3 children.
|   |
```

```

|      |      | SCAN Operator (VA = 0)
|      |      | FROM TABLE
|      |      | authors
|      |      | a
|      |      | Table Scan.
|      |      | Forward Scan.
|      |      | Positioning at start of table.
|      |      | Using I/O Size 2 Kbytes for data pages.
|      |      | With LRU Buffer Replacement Strategy for data pages.
|
|      |      | SCAN Operator (VA = 1)
|      |      | FROM TABLE
|      |      | titleauthor
|      |      | ta
|      |      | Table Scan.
|      |      | Forward Scan.
|      |      | Positioning at start of table.
|      |      | Using I/O Size 2 Kbytes for data pages.
|      |      | With LRU Buffer Replacement Strategy for data pages.
|
|      |      | SCAN Operator (VA = 2)
|      |      | FROM TABLE
|      |      | titles
|      |      | t
|      |      | Index : titles_6720023942
|      |      | Forward Scan.
|      |      | Positioning by key.
|      |      | Index contains all needed columns. Base table will not be read.
|      |      | Keys are:
|      |      | title_id ASC
|      |      | Using I/O Size 2 Kbytes for index leaf pages.
|      |      | With LRU Buffer Replacement Strategy for index leaf pages.

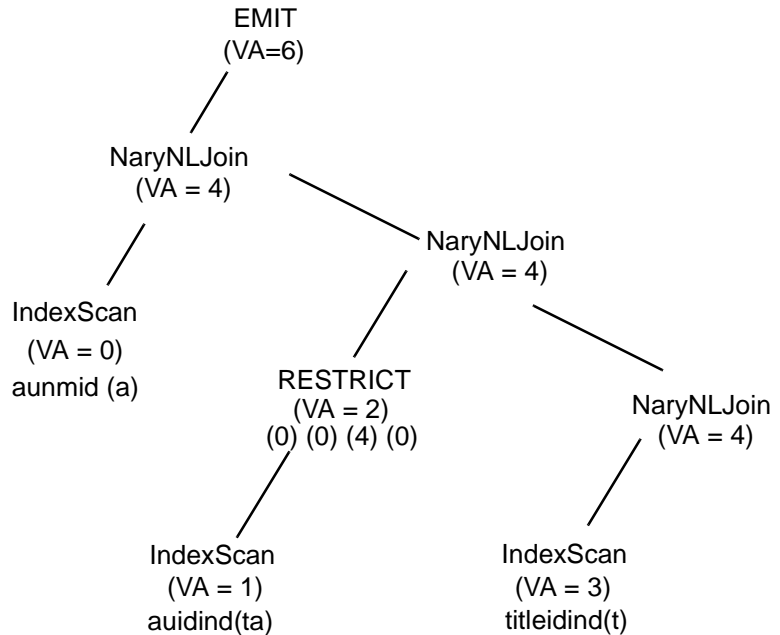
```

Figure 2-3 depicts a series of NESTED LOOP JOINS.

Figure 2-3: Emit operator tree with Nested loop joins

All query processor operators are assigned a virtual address. The lines in Figure 2-3 with VA = report the virtual address for a given operator.

The effective join order is authors, titleauthor, titles. A **RESTRICT** operator is the parent operator of the scan on titleauthors. This plan is transformed into the **NARY NESTED LOOP JOIN** plan below:

Figure 2-4: NARY NESTED LOOP JOIN operator

The transformation retains the original join order of authors, titleauthor, and titles. In this example, the scan of titles has two searchable arguments on it—`ta.title_id = t.title_id` and `a.au_id = t.title_id`. So, the scan of titles fails because of the searchable argument value established by the scan of titleauthor, or it fails because of the searchable argument value established by the scan of authors. If no rows are returned from a scan of titles because of the searchable argument value set by the scan of authors, there is no point in continuing the scan of titleauthor. For every row fetched from titleauthor, the scan of titles fails. It is only when a new row is fetched from authors that the scan of titles might succeed. This is why **NARY NESTED LOOP JOINS** have been implemented; they eliminate the useless draining of tables that have no impact on the rows returned by successive scans.

In the example, the **NARY NESTED LOOP JOIN** operator closes the scan of titleauthor, fetches a new row from authors, and repositions the scan of titleauthor based on the `au_id` fetched from authors. Again, this can be a significant performance improvement as it eliminates the needless draining of the titleauthor table and the associated I/O that could occur.

semijoin

The semijoin is a variant of `NESTED LOOP JOIN` operator, and includes the `NESTED LOOP JOIN` operator in its result set. When you make a semi-join between two tables, Adaptive Server returns the rows from the first table that contain one or more matches in the second table (a regular join returns the matching rows from the first table only once). That is, instead of scanning a table to return all matching values, an semijoin returns rows when it finds the first matching value and then stops processing. Semijoins are also known as “existence joins.”

For example, if you perform a semijoin on the titles and titleauthor tables:

```
select title
from titles
where title_id in (select title_id from titleauthor)
and title like "A Tutorial%"
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

STEP 1

The type of query is SELECT.

4 operator(s) under root

|ROOT:EMIT Operator (VA = 4)

```
|
|  |NESTED LOOP JOIN Operator (VA = 3) (Join Type: Left Semi Join)
|  |
|  |  |RESTRICT Operator (VA = 1) (0) (0) (0) (6) (0)
|  |  |
|  |  |  |SCAN Operator (VA = 0)
|  |  |  |  FROM TABLE
|  |  |  |  titles
|  |  |  |  Index : titleind
|  |  |  |  Forward Scan.
|  |  |  |  Positioning by key.
|  |  |  |  Keys are:
|  |  |  |  title ASC
|  |  |  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  |  |  With LRU Buffer Replacement Strategy for index leaf pages.
|  |  |  |  Using I/O Size 2 Kbytes for data pages.
|  |  |  |  With LRU Buffer Replacement Strategy for data pages.
|  |  |
|  |  |SCAN Operator (VA = 2)
|  |  |  FROM TABLE
|  |  |  titleauthor
|  |  |  Index : titleidind
```

```
| | | Forward Scan.  
| | | Positioning by key.  
| | | Index contains all needed columns. Base table will not be read.  
| | | Keys are:  
| | | title_id ASC  
| | | Using I/O Size 2 Kbytes for index leaf pages.  
| | | With LRU Buffer Replacement Strategy for index leaf pages.
```

Distinct operators

There are three unary operators you can use to enforce distinctness: `GROUP SORTED Distinct`, `SORT Distinct`, and `HASH Distinct`. Each has advantages and disadvantages. The optimizer chooses an efficient distinct operator with respect to its use within the entire query plan's context.

See Table 1-3 on page 23 for a list and description of all query processor operators.

GROUP SORTED Distinct operator

You can use the `GROUP SORTED Distinct` operator to apply distinctness. `GROUP SORTED Distinct` requires that the input stream is already sorted on the distinct columns. It reads a row from its child operator and initializes the current distinct columns' values to be filtered.

The row is returned to the parent operator. When the `GROUP SORTED` operator is called again to fetch another row, it fetches another row from its child and compares the values to the current cached values. If the value is a duplicate, the row is discarded and the child is called again to fetch a new row.

This process continues until a new distinct row is found. The distinct columns' values for this row are cached and are used later to eliminate nondistinct rows. The current row is returned to the parent operator for further processing.

The `GROUP SORTED Distinct` operator returns a sorted stream. The fact that it returns a sorted and distinct data stream are properties that the optimizer can use to improve performance in additional upstream processing. The `GROUP SORTED Distinct` operator is a nonblocking operator. It returns a distinct row to its parent as soon as it is fetched. It does not require the entire input stream to be processed before it can start returning rows. The following query collects distinct last and first author's names:

```
select distinct au_lname, au_fname  
from authors  
where au_lname = "Bloom"
```

QUERY PLAN FOR STATEMENT 1 (at line 2).

STEP 1

The type of query is SELECT.

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

```
|GROUP SORTED Operator (VA = 1)
|Distinct
|
|  |SCAN Operator (VA = 0)
|  |  FROM TABLE
|  |  authors
|  |  Index : aunmind
|  |  Forward Scan.
|  |  Positioning by key.
|  |  Index contains all needed columns. Base table will not be read.
|  |  Keys are:
|  |    au_lname ASC
|  |  Using I/O Size 2 Kbytes for index leaf pages.
|  |  With LRU Buffer Replacement Strategy for index leaf pages.
```

The GROUP SORTED Distinct operator is chosen in this query plan to apply the distinct property because the scan operator is returning rows in sorted order for the distinct columns au_lname and au_fname. GROUP SORTED incurs no I/O and minimal CPU overhead.

You can use the GROUP SORTED Distinct operator to implement vector aggregation. See “Vector aggregation operators” on page 84. The showplan output prints the line Distinct to indicate that this GROUP SORTED Distinct operator is implementing the distinct property.

SORT Distinct operator

The SORT Distinct operator does not require that its input stream is already sorted on the distinct key columns. It is a blocking operator that drains its child operator’s stream and sorts the rows as they are read. A distinct row is returned to the parent operator after all rows have been sorted. Rows are returned sorted on the distinct key columns. An internal worktable is used as a backing store in case the input set does not fit entirely in memory.

QUERY PLAN FOR STATEMENT 1 (at line 1)

STEP 1

The type of query is SELECT.

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

```
|SORT Operator
| Using Worktable1 for internal storage.
|
|   |SCAN Operator
|   | FROM TABLE
|   | authors
|   | Table Scan.
|   | Forward Scan.
|   | Positioning at start of table.
|   | Using I/O Size 2 Kbytes for data pages.
|   | With LRU Buffer Replacement Strategy for data pages.
```

The scan of the authors table does not return rows sorted on the distinct key columns. This requires that a SORT Distinct operator be used rather than a GROUP SORTED Distinct operator. The SORT operator's distinct key columns are au_lname and au_fname. The showplan output indicates that Worktable1 is used for disk storage in case the input set does not fit entirely in memory.

HASH Distinct operator

The HASH Distinct operator does not require that its input set be sorted on the distinct key columns. It is a nonblocking operator. Rows are read from the child operator and are hashed on the distinct key columns. This determines the row's bucket position. The corresponding bucket is searched to see if the key already exists. The row is discarded if it contains a duplicate key, and another row is fetched from the child operator. The row is added to the bucket if no duplicate distinct key already exists and the row is passed up to the parent operator for further processing. Rows are not returned sorted on the distinct key columns.

The HASH Distinct operator is generally used when the input set is not already sorted on the distinct key columns or when the optimizer cannot use the ordering coming out of the distinct processing later in the plan.

```
select distinct au_lname, au_fname
from authors
where city = "Oakland"
go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

```
| HASH DISTINCT Operator (VA = 1)
|   Using Worktable1 for internal storage.
|
|   | SCAN Operator (VA = 0)
|   |   FROM TABLE
|   |   authors
|   |   Table Scan.
|   |   Forward Scan.
|   |   Positioning at start of table.
|   |   Using I/O Size 2 Kbytes for data pages.
|   |   With LRU Buffer Replacement Strategy for data pages.
```

In this example, the output of the authors table scan is not sorted. The optimizer can choose either a SORT Distinct or HASH Distinct operator strategy. The ordering provided by a SORT Distinct strategy is not useful anywhere else in the plan, so the optimizer will probably choose a HASH Distinct strategy. The optimizer's decision is ultimately based on cost estimates. The HASH Distinct is typically less expensive for unsorted input streams can eliminate rows on the fly for resident partitions. The SORT Distinct operator cannot eliminate any rows until the entire data set has been sorted.

The showplan output for the HASH Distinct operator reports that Worktable1 will be used. A worktable is needed in case the distinct row result set cannot fit in memory. In that case, partially processed groups are written to disk.

Vector aggregation operators

There are three unary operators used for vector aggregation. They are the GROUP SORTED COUNT AGGREGATE, the HASH VECTOR AGGREGATE, and the GROUP INSERTING operators.

See Table 1-3 on page 23 for a list and description of all query processor operators.

GROUP SORTED COUNT AGGREGATE operator

The GROUP SORTED COUNT AGGREGATE nonblocking operator is a variant of the GROUP SORTED Distinct operator described in “GROUP SORTED Distinct operator” on page 81. The GROUP SORTED COUNT AGGREGATE operator requires that input set to be sorted on the group by columns. The algorithm is very similar to that of GROUP SORTED Distinct.

A row is read from the child operator. If the row is the start of a new vector, its grouping columns are cached and the aggregation results are initialized.

If the row belongs to the current group being processed, the aggregate functions are applied to the aggregate results. When the child operator returns a row that starts a new group or End Of Scan, the current vector and its aggregated values are returned to the parent operator.

The first row in the GROUP SORTED COUNT AGGREGATE operator is returned after an entire group is processed, where the first row in the GROUP SORTED Distinct operator is returned at the start of a new group. This example collects a list of all cities with the number of authors that live in each city.

```
select city, total_authors = count(*)
from authors
group by city
plan
"(group_sorted
(sort (scan authors))
)"
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.

STEP 1
The type of query is SELECT.

3 operator(s) under root

ROOT:EMIT Operator (VA = 3)

```
|GROUP SORTED Operator (VA = 2)
|  Evaluate Grouped COUNT AGGREGATE.
|
|      |SORT Operator (VA = 1)
|      |  Using Worktable1 for internal storage.
|      |
|      |      |SCAN Operator (VA = 0)
|      |      |  FROM TABLE
```

```

|      |      | authors
|      |      | Table Scan.
|      |      | Forward Scan.
|      |      | Positioning at start of table.
|      |      | Using I/O Size 2 Kbytes for data pages.
|      |      | With LRU Buffer Replacement Strategy for data pages.

```

In this query plan, the scan of authors does not return rows in grouping order. A SORT operator is applied to order the stream based on the grouping column city. At this point, a GROUP SORTED COUNT AGGREGATE operator can be applied to evaluate the count aggregate.

The GROUP SORTED COUNT AGGREGATE operator showplan output reports the aggregate functions being applied as:

```

| Evaluate Grouped COUNT AGGREGATE.

```

HASH VECTOR AGGREGATE operator

The HASH VECTOR AGGREGATE operator is a blocking operator. All rows from the child operator must be processed before the first row from the HASH VECTOR AGGREGATE operator can be returned to its parent operator. Other than this, the algorithm is similar to the HASH Distinct operator's algorithm.

Rows are fetched from the child operator. Each row is hashed on the query's grouping columns. The bucket that is hashed is searched to see if the vector already exists.

If the group by values do not exist, the vector is added and the aggregate values are initialized using this first row. If the group by values do exist, the current row is aggregated to the existing values. This example collects a list of all cities with the number of authors that live in each city.

```

select city, total_authors = count(*)
from authors
group by city

```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1
The type of query is SELECT.

2 operator(s) under root

ROOT:EMIT Operator (VA = 2)

```

| HASH VECTOR AGGREGATE Operator (VA = 1)
| GROUP BY

```



```

| Evaluate Grouped COUNT AGGREGATE.
| Using Worktable1 for internal storage.
| Key Count: 1
|
|   | SCAN Operator (VA = 0)
|   | FROM TABLE
|   | authors
|   | Table Scan.
|   | Forward Scan.
|   | Using I/O Size 2 Kbytes for data pages.
|   | With LRU Buffer Replacement Strategy for data pages.

```

In this query plan, the `HASH VECTOR AGGREGATE` operator reads all of the rows from its child operator, which is scanning the authors table. Each row is checked to see if there is already an entry bucket entry for the current city value. If there is not, a hash entry row is added with the new city grouping value and the count result is initialized to 1. If there is already a hash entry for the new row's city value, the aggregation function is applied. In this case, the count result is incremented.

The showplan output prints a group by message specifically for the `HASH VECTOR AGGREGATE` operator, then prints the grouped aggregation messages:

```

| Evaluate Grouped COUNT AGGREGATE.

```

The showplan output reports used to store spilled groups and unprocessed rows:

```

| Using Worktable1 for internal storage.

```

GROUP INSERTING

`GROUP INSERTING` is a blocking operator. All rows from the child operator must be processed before the first row can be returned from the `GROUP INSERTING`.

`GROUP INSERTING` is limited to 31 or fewer columns in the group by clause. The operator starts by creating a worktable with a clustered index of the grouping columns. As each row is fetched from the child, a lookup into the work table is done based on the grouping columns. If no row is found, then the row is inserted. This effectively creates a new group and initializes its aggregate values. If a row is found, the new aggregate values are updated based on evaluating the new values. The `GROUP INSERTING` operator returns rows ordered by the grouping columns.

```

select city, total_authors = count(*)
from authors
group by city

```

```
plan
'(group_inserting (i_scan auidind authors ))'

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.
```

```
STEP 1
The type of query is SELECT.
```

```
2 operator(s) under root
|ROOT:EMIT Operator (VA = 2)
|
|GROUP INSERTING Operator (VA = 1)
|GROUP BY
|Evaluate Grouped COUNT AGGREGATE
|Using Worktable1 for internal storage.
|
|SCAN Operator (VA = 0)
|FROM TABLE
|authors
|Table Scan.
|Forward Scan.
|Positioning at start of table.
|Using I/O Size 2 Kbytes for data pages.
|With LRU Buffer Replacement Strategy for data pages.
```

In this example, the group inserting operator starts by building a worktable with a clustered index keyed on the city column. The group inserting operator proceeds to drain the authors table. For each row, a lookup is done on the city value. If there is no row in the aggregation worktable with the current city value, then the row is inserted. This creates a new group for the current city value with an initialized count value. If the row for the current city value is found, then an evaluation is done to increment the COUNT AGGREGATE value.

compute by message

Processing is done in the EMIT operator, and requires that the EMIT operator's input stream be sorted according to any order by requirements in the query. The processing is similar to what is done in the GROUP SORTED AGGREGATE operator.

Each row read from the child is checked to see if it starts a new group. If it does not, aggregate functions are applied as appropriate to the query's requested groups. If a new group is started, the current group and its aggregated values are returned to the user. A new group is then started and its aggregate values are initialized from the new row's values. This example collects an ordered list of all cities and reports a count of the number of entries for each city after the city list.

```
select city
from authors
order by city
compute count(city) by city
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

2 operator(s) under root

Emit with Compute semantics

ROOT:EMIT Operator (VA = 2)

```
| SORT Operator (VA = 1)
|   Using Worktable1 for internal storage.
|
|   | SCAN Operator (VA = 0)
|   |   FROM TABLE
|   |   authors
|   |   Table Scan.
|   |   Forward Scan.
|   |   Positioning at start of table.
|   |   Using I/O Size 2 Kbytes for data pages.
|   |   With LRU Buffer Replacement Strategy for data pages.
```

In this example, the EMIT operator's input stream is sorted on the city attribute. For each row, the compute by count value is incremented. When a new city value is fetched, the current city's values and associated count value is returned to the user. The new city value becomes the new compute by grouping value and its count is initialized to one.

Union operators

UNION ALL operator

The UNION ALL operator merges several compatible input streams without performing any duplicate elimination. Every data row that enters the UNION ALL operator is included in the operator's output stream.

The UNION ALL operator is a nary operator that displays this message:

UNION ALL OPERATOR has N children.

N is the number of input streams into the operator.

This example demonstrates the use of UNION ALL:

```
select * from sysindexes where id < 100
union all
select * from sysindexes where id > 200
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

3 operator(s) under root

```
| ROOT:EMIT Operator (VA = 3)
|
| | UNION ALL Operator (VA = 2) has 2 children.
| |
|
| | | SCAN Operator (VA = 0)
| | | FROM TABLE
| | | sysindexes
| | | Using Clustered Index.
| | | Index : csysindexes
| | | Forward Scan.
| | | Positioning by key.
| | | Keys are:
| | | id ASC
| | | Using I/O Size 2 Kbytes for index leaf pages.
| | | With LRU Buffer Replacement Strategy for index leaf pages.
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.
```

```
SCAN Operator (VA = 1)
FROM TABLE
sysindexes
Using Clustered Index
Index : csysindexes
Forward scan.
Positioning by key.
Keys are:
    id ASC
Using I/O Size 2 Kbytes for index leaf pages.
With LRU Buffer Replacement Strategy for index leaf pages.
Using I/O Size 2 Kbytes for data pages.
With LRU Buffer Replacement Strategy for data pages.
```

The UNION ALL operator starts by fetching all rows from its leftmost child. In this example, it returns all of the sysindexes rows with an ID less than 100. As each child operator's datastream is emptied, the UNION ALL operator moves on to the child operator immediately to its right. This stream is opened and emptied. This continues until the last (the *N*th) child operator is emptied.

MERGE UNION operator

The `MERGE UNION` operator performs a `UNION ALL` operation on several sorted compatible data streams and eliminates duplicates within these streams.

The MERGE UNION operator is a nary operator that displays this message:

MERGE UNION OPERATOR has $\langle N \rangle$ children.

$\langle N \rangle$ is the number of input streams into the operator.

HASH UNION

The `HASH UNION` operator uses Adaptive Server hashing algorithms to simultaneously perform a `UNION ALL` operation on several data streams and hash-based duplicate elimination.

The HASH UNION operator is a nary operator that displays this message:

HASH UNION OPERATOR has <N> children.

$\langle N \rangle$ is the number of input streams into the operator.

HASH UNION also displays the name of the worktable it uses, in this format:

```
HASH UNION OPERATOR Using Worktable <X> for internal
storage.
```

This worktable is used by the HASH UNION operator to temporarily store data for the current iteration that cannot be processed in the memory currently available.

This example demonstrates the use of HASH UNION:

```
select * from sysindexes
union
select * from sysindexes
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

3 operator(s) under root

```
| ROOT:EMIT Operator (VA = 3)
|
|   | HASH UNION Operator has 2 children.
|   |   Using Worktable1 for internal storage.
|   |
|   |   | SCAN Operator
|   |   |   FROM TABLE
|   |   |   sysindexes
|   |   |   Table Scan.
|   |   |   Forward Scan.
|   |   |   Positioning at start of table.
|   |   |   Using I/O Size 2 Kbytes for data pages.
|   |   |   With LRU Buffer Replacement Strategy for data pages.
|   |
|   |   | SCAN Operator (VA = 1)
|   |   |   FROM TABLE
|   |   |   sysindexes
|   |   |   Table Scan.
|   |   |   Forward Scan.
|   |   |   Positioning at start of table.
|   |   |   Using I/O size 2 Kbytes for data pages.
|   |   |   With LRU Buffer Replacement Strategy for data pages.
```

SCALAR AGGREGATE operator

The SCALAR AGGREGATE operator keeps track of running information about an input data stream, such as the number of rows in the stream, or the maximum value of a given column in the stream.

The SCALAR AGGREGATE operator prints a list of up to 10 messages describing the scalar aggregation operations it executes. The message has the following format:

Evaluate Ungrouped <Type of Aggregate> Aggregate

<Type of Aggregate> can be any of the following: count, sum, average, min, max, any, once-unique, count-unique, sum-unique, average-unique, or once.

The following query performs a SCALAR AGGREGATE (that is, unwrapped) aggregation on the authors table in the pubs2 database:

```
select count(*) from authors
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
```

```
The type of query is SELECT.
```

```
2 operator(s) under root
```

```
ROOT:EMIT Operator (VA = 2)
```

```

| SCALAR AGGREGATE Operator (VA = 1)
|   Evaluate Ungrouped COUNT AGGREGATE.
|
|   | SCAN Operator (VA =0)
|   |   FROM TABLE
|   |   authors
|   |   Index : aunmind
|   |   Forward Scan.
|   |   Positioning at index start.
|   |   Index contains all needed columns. Base table will not be read.
|   |   Using I/O Size 4 Kbytes for index leaf pages.
|   |   With LRU Buffer Replacement Strategy for index leaf pages.
```

The SCALAR AGGREGATE message indicates that the query to be executed is an ungrouped count aggregation.

RESTRICT operator

The **RESTRICT** operator is a unary operator that evaluates expressions based on column values. The **RESTRICT** operator is associated with multiple column evaluations lists that can be processed before fetching a row from the child operator, after fetching a row from the child operator, or to compute the value of virtual columns after fetching a row from the child operator.

SORT operator

The **SORT** operator has only one child operator within the query plan. Its role is to generate an output data stream from the input stream, using a specified sorting key.

The **SORT** operator may execute a streaming sort when possible, but may also have to store results temporarily into a worktable. The **SORT** operator displays the worktable's name in this format:

Using Worktable<N> for internal storage.

where <N> is a numeric identifier for the worktable within the showplan output.

Here is an example of a simple query plan using a **SORT** operator and a worktable:

```
select au_id from authors order by postalcode
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
```

```
The type of query is SELECT.
```

```
2 operator(s) under root
```

```
ROOT:EMIT Operator (VA = 2)
```

```
|SORT Operator (VA = 1)
|  Using Worktable1 for internal storage.
|
|  |SCAN Operator (VA = 0)
|  |  FROM TABLE
|  |  authors
|  |  Table Scan.
|  |  Forward Scan.
|  |  Positioning at start of table.
|  |  Using I/O Size 4 Kbytes for data pages.
```


| | With LRU Buffer Replacement Strategy for data pages.

The SORT operator drains its child operator and sorts the rows. In this case, it sorts each row fetched from the authors table using the postalcode attribute. If all of the rows fit into memory, then no data is spilled to disk. But, if the input data's size exceeds the available buffer space, then sorted runs are spilled to disk. These runs are recursively merged into larger sorted runs until there are fewer runs than there are available buffers to read and merge the runs with.

STORE operator

The STORE operator is used to create a worktable, fill it, and possibly create an index on it. As part of the execution of a query plan, the worktable is used by other operators in the plan. A SEQUENCER operator guarantees that the plan fragment corresponding to the worktable and potential index creation is executed before other plan fragments that use the worktable. This is important when a plan is executed in parallel, because execution processes operate asynchronously.

Reformatting strategies use the STORE operator to create a worktable with a clustered index on it.

If the STORE operator is used for a reformatting operation, it prints this message:

```
Worktable <X> created, in <L> locking mode for
reformatting.
```

The locking mode <L> has to be one of “allpages,” “datapages,” or “datarows.”

The STORE operator also prints this message:

```
Creating clustered index.
```

If the STORE operator is not used for a reformatting operation, it prints this message:

```
Worktable <X> created, in <L> locking mode.
```

The following example applies to the STORE operator, as well as to the SEQUENCER operator.

```
select * from bigun a, bigun b where a.c4 = b.c4 and a.c2 < 10
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.
```

```
STEP 1
```

The type of query is SELECT.

7 operator(s) under root

```

|ROOT:EMIT Operator (VA = 7)
|
|  |SEQUENCER Operator (VA = 6) has 2 children.
|  |
|  |  |STORE Operator (VA = 5)
|  |  |  Worktable1 created, in allpages locking mode, for REFORMATTING.
|  |  |  Creating clustered index.
|  |  |
|  |  |  |INSERT Operator(VA = 4)
|  |  |  |  The update mode is direct.
|  |  |  |
|  |  |  |  |SCAN Operator(VA = 0)
|  |  |  |  |  FROM TABLE
|  |  |  |  |  bigun
|  |  |  |  |  b
|  |  |  |  |  Table Scan.
|  |  |  |  |  Forward Scan.
|  |  |  |  |  Positioning at start of table.
|  |  |  |  |  Using I/O Size 2 Kbytes for data pages.
|  |  |  |  |  With LRU Buffer Replacement Strategy for data pages.
|  |  |  |
|  |  |  |  TO TABLE (VA = 3)
|  |  |  |  Worktable1.
|  |  |
|  |  |NESTED LOOP JOIN (Join Type: Inner Join)(VA = 7)
|  |  |
|  |  |  |SCAN Operator (VA = 2)
|  |  |  |  FROM TABLE
|  |  |  |  bigun
|  |  |  |  a
|  |  |  |  Table Scan.
|  |  |  |  Forward Scan.
|  |  |  |  Positioning at start of table.
|  |  |  |  Using I/O Size 2 Kbytes for data pages.
|  |  |  |  With LRU Buffer Replacement Strategy for data pages.
|  |  |  |
|  |  |  |  |SCAN Operator (VA = 1)
|  |  |  |  |  FROM TABLE
|  |  |  |  |  Worktable1.
|  |  |  |  |  Using Clustered Index.
|  |  |  |  |  Forward Scan.
|  |  |  |  |  Positioning key.

```

```

| | | | Using I/O Size 2 Kbytes for data pages.
| | | | With LRU Buffer Replacement Strategy for data pages.

```

In the example plan shown above, the `STORE` operator is used in a reformatting strategy. It is located directly below the `SEQUENCER` operator in the leftmost child of the `SEQUENCER` operator.

The `STORE` operator creates `Worktable1`, which is filled by the `INSERT` operator below it. The `STORE` operator then creates a clustered index on `Worktable1`. The index is built on the join key `b.c4`.

SEQUENCER operator

The `SEQUENCER` operator is a nary operator used to sequentially execute each the child plans below it. The `SEQUENCER` operator is used in reformatting plans, and certain aggregate processing plans.

The `SEQUENCER` operator executes each of its child subplans, except for the rightmost one. Once all the left child subplans are executed, the rightmost subplan is executed.

The `SEQUENCER` operator displays this message:

```
SEQUENCER operator has N children.
```

```
select * from tab1 a, tab2 b where a.c4 = b.c4 and a.c2 < 10
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Optimized using the Abstract Plan in the PLAN clause.
```

```
STEP 1
```

```
The type of query is SELECT.
```

```
7 operator(s) under root
```

```

| ROOT:EMIT Operator (VA = 7)
|
| | SEQUENCER Operator (VA = 6) has 2 children.
| |
| | | STORE Operator (VA = 5)
| | |   Worktable1 created, in allpages locking mode, for REFORMATTING.
| | |   Creating clustered index.
| | |
| | | | INSERT Operator (VA = 4)
| | | |   The update mode is direct.
| | | |
| | | | SCAN Operator (VA = 0)

```

```
| | | | FROM TABLE
| | | | tab2
| | | | b
| | | | Table Scan.
| | | | Forward Scan.
| | | | Positioning at start of table.
| | | | Using I/O Size 2 Kbytes for data pages.
| | | | With LRU Buffer Replacement Strategy for data pages.
|
| | TO TABLE
| | | Worktable1.
|
| NESTED LOOP JOIN Operator (Join Type: Inner Join) (VA = 3)
|
| | SCAN Operator (VA = 2)
| | | FROM TABLE
| | | tab1
| | | a
| | | Table Scan.
| | | Forward Scan.
| | | Positioning at start of table.
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.
|
| | SCAN Operator (VA = 1)
| | | FROM TABLE
| | | Worktable1.
| | | Using Clustered Index.
| | | Forward Scan.
| | | Positioning by key.
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.
```

In this example, the SEQUENCER operator implements a reformatting strategy. The leftmost branch of the SEQUENCER operator creates a clustered index on Worktable1. This branch is executed and closed before the SEQUENCER operator proceeds to the next child operator. The SEQUENCER operator arrives at the rightmost child, opens, and begins to drain it, returning rows back to its parent operator. The design intent of the SEQUENCER operator is for operators in the rightmost branch to use the worktables created in the preceding outer branches of the SEQUENCER operator. In this example, Worktable1 is used in a nested-loop join strategy. The scan of Worktable1 is positioned by a key on its clustered index for each row that comes from the outer scan of tab1.

REMOTE SCAN operator

The `REMOTE SCAN` operator sends a SQL query to a remote server for execution. It then processes the results returned by the remote server, if any. `REMOTE SCAN` displays the formatted text of the SQL query it handles.

`REMOTE SCAN` has 0 or 1 child operators.

SCROLL operator

The `SCROLL` operator encapsulates the functionality of scrollable cursors in Adaptive Server. Scrollable cursors may be insensitive, meaning that they display a snapshot of their associated data, taken when the cursor is opened, or semi-sensitive, meaning that the next rows to be fetched are retrieved from the live data.

The `SCROLL` operator is a unary operator that displays this message:

```
SCROLL OPERATOR ( Sensitive Type: <T>)
```

The type may be insensitive or semi-sensitive.

This is an example of a plan featuring an insensitive scrollable cursor:

```
declare CI insensitive scroll cursor for
select au_lname, au_id from authors
go
set showplan on
go
open CI
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
STEP 1
```

```
The type of query is OPEN CURSOR CI.
```

```
QUERY PLAN FOR STATEMENT 1 (at line 2).
```

```
STEP 1
```

```
The type of query is DECLARE CURSOR.
```

```
2 operator(s) under root
```

```
ROOT:EMIT Operator (VA = 2)
```

```
|SCROLL Operator (Sensitive Type: Insensitive) (VA = 1)
```

```

| Using Worktable1 for internal storage.
|
| | SCAN Operator (VA = 0)
| | FROM TABLE
| | authors
| | Table Scan.
| | Forward Scan.
| | Positioning at start of table.
| | Using I/O Size 4 Kbytes for data pages.
| | With LRU Buffer Replacement Strategy for data pages.

```

The `SCROLL` operator is the child operator of the root `EMIT` operator, and its only child is the `SCAN` operator on the `authors` table. `SCROLL` message specifies that the CI cursor is insensitive.

Scrollable cursor rows are initially cached in memory. `Worktable1` is used as a backing store for this cache when the amount of data processed exceeds the cache's physical memory limits.

RID JOIN operator

The `RID JOIN` operator is a binary operator that joins two data streams, based on row IDs generated for the same source table. Each data row in a SQL table is associated with a unique row ID (RID). Think of a rid-join as a special case of a self-join query. The left child fills a worktable with the set of uniquely qualifying RIDs. The RIDs are the result of applying a distinct filter to the RIDs returned from two or more disparate index cases of the same source table.

The `RID JOIN` operator is used to implement the general or strategy. The general-or strategy is often used when a query's predicate contains a collection of disjunctions that can be qualified by different indexes on the same table. In this case, each index is scanned based on the predicates that can be qualified by that index. For each index row that qualifies, a RID is returned.

The returned RIDs are processed for uniqueness so that the same row is not returned twice, which might happen if two or more of the disjunctions qualify the same row.

The `RID JOIN` operator inserts the unique RIDs into a worktable. The worktable of unique RIDs is passed to the scan operator in the rid-join's right branch. The access methods can iteratively fetch the next RID to be processed directly from the worktable, and look up the associated row. This row is then returned to the `RID JOIN` parent operator.

The `RID JOIN` operator displays this message:

Using Worktable <N> for internal storage.

This worktable is used to store the unique RIDs generated from the left child.

The following example demonstrates the showplan output for the RID JOIN operator.

```
select * from tab1 a where a.c1 = 10 or a.c3 = 10
```

```
QUERY PLAN FOR STATEMENT 1 (at line 2).
```

STEP 1

The type of query is SELECT.

6 operator(s) under root.

```
| ROOT:EMIT Operator (VA = 6)
|
| | RID JOIN Operator (VA = 5)
| |   Using Worktable2 for internal storage.
| |
| | | HASH UNION Operator (VA = 6) has 2 children.
| | |   Key Count: 1
| | |
| | | | SCAN Operator (VA = 0)
| | | |   FROM TABLE
| | | |   tab1
| | | |   a
| | | |   Index:tablidx
| | | |   Forward Scan.
| | | |   Positioning by key.
| | | |   Index contains all needed columns. Base table will not be read.
| | | |   Keys are:
| | | |   c1 ASC
| | | |   Using I/O Size 2 Kbytes for index leaf pages.
| | | |   With LRU Buffer Replacement Strategy for index leaf pages.
| | |
| | | | SCAN Operator (VA = 4)
| | | |   FROM TABLE
| | | |   tab1
| | | |   a
| | | |   Index:tablidx2
| | | |   Forward Scan.
| | | |   Positioning by key.
| | | |   Index contains all needed columns. Base table will not be read.
| | | |   Keys are:
| | | |   c3 ASC
```

```

| | | Using I/O Size 2 Kbytes for index leaf pages.
| | | With LRU Buffer Replacement Strategy for index leaf pages.
|
| | RESTRUCT Operator (VA = 3)
|
| | | SCAN Operator (VA = 2)
| | | FROM TABLE
| | | tab1
| | | a
| | | Using Dynamic Index.
| | | Forward Scan.
| | | Positioning by Row IDentifier (RID).
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.

```

In this example, the index `tab1idx` is scanned to get all RIDs from `tab1` that have a `c1` value of 10. Adaptive Server scans `tab1idx2` to get all RIDs from `tab1` that have a `c3` value of 10.

The `HASH UNION` operator is used to eliminate duplicate RIDs. There are duplicate RIDs for any `tab1` rows where both `c1` and `c3` rows have a value of 10.

The `RID JOIN` operator inserts all of the returned rows into `Worktable2`. `Worktable2` is passed to the scan of `tab1` after it has been completely filled. The access methods fetch the first RID, look up the associated row, and return it to the `RID JOIN` operator. On subsequent calls to the `tab1`'s scan operator, the access methods fetch the next RID to be processed and return its associated row.

SQLFILTER operator

The `SQLFILTER` operator is a nary operator that executes subqueries. Its leftmost child represents the outer query, and the other children represent query plan fragments associated with one or more subqueries.

The leftmost child generates correlation values that are substituted into the other child plans.

The `SQLFILTER` operator displays this message:

```
SQLFILTER Operator has <N> children.
```

This example illustrates the use of `SQLFILTER`:

```

select pub_name from publishers
where pub_id =
(select distinct titles.pub_id from titles

```



```

        where publishers.pub_id = titles.pub_id
        and price > $1000)
QUERY PLAN FOR STATEMENT 1 (at line 1).
4 operator(s) under root

STEP 1
The type of query is SELECT.

4 operator(s) under root

ROOT:EMIT Operator (VA = 4)

|SQFILTER Operator (VA = 3) has 2 children.
|
| |SCAN Operator (VA = 0)
| |  FROM TABLE
| |  publishers
| |  Table Scan.
| |  Forward Scan.
| |  Positioning at start of table.
| |  Using I/O Size 8 Kbytes for data pages.
| |  With LRU Buffer Replacement Strategy for data pages.
|
| Run subquery 1 (at nesting level 1)
|
|   QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 3)
|
|   Correlated Subquery
|   Subquery under an EXPRESSION predicate.
|
|   |SCALAR AGGREGATE Operator (VA = 2)
|   |  Evaluate Ungrouped ONCE-UNIQUE AGGREGATE
|   |
|   | |SCAN Operator (VA = 1)
|   | |  FROM TABLE
|   | |  titles
|   | |  Table Scan.
|   | |  Forward Scan.
|   | |  Positioning at start of table.
|   | |  Using I/O Size 8 Kbytes for data pages.
|   | |  With LRU Buffer Replacement Strategy for data pages.
|
| END OF QUERY PLAN FOR SUBQUERY 1

```

The `SQLFILTER` operator has two children in this example. The leftmost child is the query's outer block. It is a simple scan of the publishers table. The right child is used to evaluate the query's subquery. `SQLFILTER` fetch rows from the outer block. For every row from the outer block, `SQLFILTER` invokes the right child to evaluate the subquery. If the subquery evaluates to `TRUE`, a row is returned to the `SQLFILTER`'s parent operator.

EXCHANGE operator

The `EXCHANGE` operator is a unary operator that encapsulates parallel processing of Adaptive Server SQL queries. `EXCHANGE` can be located almost anywhere in a query plan and divides the query plan into plan fragments. A plan fragment is a query plan tree that is rooted at an `EMIT` or `EXCHANGE:EMIT` operator and has leaves that are `SCAN` or `EXCHANGE` operators. A serial plan is a plan fragment that is executed by a single process.

An `EXCHANGE` operator's child operator is always an `EXCHANGE:EMIT` operator. `EXCHANGE:EMIT` is the root of a new plan fragment. An `EXCHANGE` operator has an associated server process called the Beta process that acts as a local execution coordinator for the `EXCHANGE` operator's worker processes. Worker processes execute the plan fragment as directed by the parent `EXCHANGE` operator and its Beta process. The plan fragment is often executed in a parallel fashion, using two or more processes. The `EXCHANGE` operator and Beta process coordinate activities, including the exchange of data between the fragment boundaries.

The topmost plan fragment, rooted at an `EMIT` operator rather than an `EXCHANGE:EMIT` operator, is executed by the Alpha process. The Alpha process is a consumer process associated with the user connection. The Alpha process is the global coordinator of all of the query plan's worker processes. It is responsible for initially setting up all of the plan fragment's worker processes and eventually freeing them. It manages and coordinates all of the fragment's worker processes in the case of an exception.

The `EXCHANGE` operator displays this message:

```
Executed in parallel by N producer and P consumer processes.
```

The number of producers refers to the number of worker processes that execute the plan fragment located beneath the EXCHANGE operator. The number of consumers refers to the number of worker processes that execute the plan fragment that contains the EXCHANGE operator. The consumers process the data passed to them by the producers. Data is exchanged between the producer and consumer processes through a pipe set up in the EXCHANGE operator. The producer's EXCHANGE:EMIT operator writes rows into the pipe while consumers read rows from this pipe. The pipe mechanism synchronizes producer writes and consumer reads such that no data is lost.

This example illustrates a parallel query in the master database against the system table sysmessages:

```
use master
go
set showplan on
go
select count(*) from sysmessages t1 plan '(t_scan t1) (prop t1 (parallel 4))
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Optimized using the forced options (internally generated Abstract Plan).

Executed in parallel by coordinating process and 4 worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
| SCALAR AGGREGATE Operator
|   Evaluate Ungrouped COUNT AGGREGATE.
|
|   | EXCHANGE Operator
|   | Executed in parallel by 4 Producer and 1 Consumer processes.
|
|   |   | EXCHANGE:EMIT Operator
|   |   |   | SCAN Operator
|   |   |   |   FROM TABLE
|   |   |   |   sysmessages
|   |   |   |   Table Scan.
|   |   |   |   Forward Scan.
|   |   |   |   Positioning at start of table.
|   |   |   |   Executed in parallel with a 4-way hash scan.
|   |   |   |   Using I/O Size 4 Kbytes for data pages.
|   |   |   |   With LRU Buffer Replacement Strategy for data pages.
```

There are two plan fragments in this example. The first fragment in any plan, parallel or not, is always rooted by an `EMIT` operator. The first fragment in this example consists of the `EMIT`, `SCALAR AGGREGATE`, and `EXCHANGE` operators. This first fragment is always executed by the single Alpha process. In this example, it also acts as the Beta process responsible for managing the `EXCHANGE` operator's worker processes.

The second plan fragment is rooted at the `EXCHANGE:EMIT` operator. Its only child operator is the `SCAN` operator. The `SCAN` operator is responsible for scanning the sysmessages table. The scan is executed in parallel:

`Executed in parallel with a 4-way hash scan`

This indicates that each worker process is responsible for approximately a quarter of the table. Pages are assigned to the worker processes based on having the data page ID.

The `EXCHANGE:EMIT` operator writes data rows to the consumers by writing to a pipe created by its parent `EXCHANGE` operator. In this example, the pipe is a four-to-one demultiplexer, and include several pipe types that perform quite different behaviors.

INSTEAD-OF TRIGGER operators

There are two operators associated with the instead-of triggers feature: `INSTEAD-OF TRIGGER` and `CURSOR SCAN`. The instead-of trigger feature is available as of Adaptive Server version 15.0.2. The instead-of trigger feature uses pseudotables, which allow the user to apply specific actions for inserts, deletes, and updates on views, when these actions would otherwise have been ambiguous.

INSTEAD-OF TRIGGER operator

The `INSTEAD-OF TRIGGER` operator appears only in query plans for insert, update, or delete statements on a view that has an instead-of trigger created upon it. Its function is to create and fill the inserted and deleted pseudotables that are used in the trigger to examine the rows that would have been modified by the original insert, update, or delete query. The only purpose of the query plan that contains an `INSTEAD-OF TRIGGER` operator is to fill the inserted and deleted tables—the actual operation of the original SQL statement is never attempted on the view referenced in the statement. Rather, it is up to the trigger to perform the updates to the view's underlying tables based on the data available in the inserted and deleted pseudo tables.

The following is an example of the `INSTEAD-OF TRIGGER` operator's showplan output:

```
create table t12 (c0 int primary key, c1 int null, c2 int null)
go
. . .
create view t12view as select c1,c2 from t12
go
create trigger v12updtrg on t12view
instead of update as
select * from deleted
go
update t12view set c1 = 3
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

2 operator(s) under root

```
| ROOT:EMIT Operator (VA = 1)
|
| | INSTEAD-OF TRIGGER Operator
| | Using Worktable1 for internal storage.
| | Using Worktable2 for internal storage.
| |
| | | SCAN Operator (VA = 0)
| | | FROM TABLE
| | | t12
| | | Table Scan.
| | | Forward Scan.
| | | Positioning at start of table.
```

```
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.
```

In this example, the `v12updttrig` instead-of trigger is defined on the `t12view`. The update to the `t12view` results in the creation of the `INSTEAD-OF TRIGGER` operator. The `INSTEAD-OF TRIGGER` operator creates two worktables. `Worktable1` and `Worktable2` are used to hold the inserted and deleted rows, respectively. These worktables are unique in that they persist across statements. Trigger execution results in the following showplan lines getting printed.

QUERY PLAN FOR STATEMENT 1 (at line 3).

```
STEP 1
  The type of query is SELECT.
```

1 operator(s) under root

```
| ROOT:EMIT Operator (VA = 1)
|
| | SCAN Operator (VA = 0)
| | FROM CACHE
```

The showplan statement output above is for the trigger's statement, `select * from deleted`. The rows to be deleted from the view were inserted into the "deleted" cache when the initial update statement was executed. Then, the trigger scans the table to report what rows would have been deleted from the `t12view` view.

CURSOR SCAN operator

The `CURSOR SCAN` operator only appears in positioned delete or update (that is, `delete view-name where current of cursor_name`) statements on a view that has an instead-of trigger created upon it. As such, it appears only as a child operator of the `INSTEAD-OF TRIGGER` operator. A positioned delete or update accesses only the row on which the cursor is currently positioned. The `CURSOR SCAN` operator reads the current row of the cursor directly from the `EMIT` operator of the query plan for the fetch cursor statement. These values are passed to the `INSTEAD-OF TRIGGER` operator to be inserted into the inserted or deleted pseudo tables (this example uses the same table as the previous example).

```
declare curs1 cursor for select * from t12view
go
```

```
open curs1
go
fetch curs1
```

c1	c2
1	2

```
(1 row affected)
set showplan on
go
update t12view set c1 = 3
where current of curs1
```

QUERY PLAN FOR STATEMENT (at line 1).

```
STEP 1
    The type of query is SELECT.
```

2 operator(s) under root

```
|ROOT:EMIT Operator (VA = 2)
|
| |INSTEAD-OF TRIGGER Operator (VA = 1)
| |  Using Worktable1 for internal storage.
| |  Using Worktable2 for internal storage.
| |
| | |CURSOR SCAN Operator (VA = 0)
| | |  FROM EMIT OPERATOR
```

The showplan output in this example is identical to that from the previous INSTEAD-OF TRIGGER operator example, with one exception. A CURSOR SCAN operator appears as the child operator of the INSTEAD-OF TRIGGER operator rather than a scan of the view's underlying tables.

The CURSOR SCAN gets the values to be inserted into the pseudo tables by accessing the result of the cursor fetch. This is conveyed by the FROM EMIT OPERATOR message.

QUERY PLAN FOR STATEMENT 1 (at line 3).

```
STEP 1
    The type of query is SELECT.
```

1 operator(s) under root

```
|ROOT:EMIT Operator (VA = 1)
```

```
|  
| SCAN Operator (VA = 0)  
| FROM CACHE
```

The showplan statement above is for the trigger's statement. It is identical to the output in the INSTEAD-OF TRIGGER example.

deferred_index and deferred_varcol messages

The update mode is deferred_varcol.

The update mode is deferred_index.

These showplan messages indicate that Adaptive Server may process an update command as a deferred index update.

Adaptive Server uses deferred_varcol mode when updating one or more variable-length columns. This update may be done in deferred or direct mode, depending on information that is available only at runtime.

Adaptive Server uses deferred_index mode when the index is unique or may change as part of the update. In this mode, Adaptive Server deletes the index entries in direct mode but inserts them in deferred mode.

Displaying Query Optimization Strategies and Estimates

This chapter describes the messages printed by the query optimization options of the `set` command.

Topic	Page
set commands for text format messages	111
set commands for XML format messages	112
Diagnostic usage scenarios	119
Permissions for set commands	122

set commands for text format messages

Either the query optimizer or the query execution layer can generate diagnostic output. To generate diagnostic output in text format, use:

```
set option
{ {show | show_lop | show_managers | show_log_props |
  show_parallel | show_histograms | show_abstract_plan |
  show_search_engine | show_counters | show_best_plan |
  show_code_gen | show_pio_costing | show_ljo_costing |
  show_pll_costing | show_elimination | show_missing_stats}
{normal | brief | long | on | off} }...
```

Note Each option specified must be followed by one of `normal`, `brief`, `long`, `on`, or `off`. `on` and `normal` are equivalent. Each `show` option must include one of these choices (`normal`, `brief`, and so on); specify more than one option in a single `set` option command by separating each option or choice pair with commas.

See “Diagnostic usage scenarios” on page 119 for examples of using the `set` options.

Table 3-1: Optimizer set commands for text format messages

Option	Definition
show	Shows a reasonable collection of details, where the collection depends on the choice of {normal brief long on off}
show_lob	Shows the logical operators used
show_managers	Shows the data structure managers used during optimization
show_log_props	Shows the logical properties evaluated
show_parallel	Shows details of parallel query optimization
show_histograms	Shows the processing of histograms associated with SARG/join columns
show_abstract_plan	Shows the details of an abstract plan
show_search_engine	Shows the details of the join-ordering algorithm
show_counters	Shows the optimization counters
show_best_plan	Shows the details of the best query plan selected by the optimizer
show_code_gen	Shows details of code generation
show_pio_costing	Shows estimates of physical input/output (reads/writes from/to the disk)
show_ljo_costing	Shows estimates of logical input/output (reads/writes from/to memory)
show_pll_costing	Shows estimates relating to costing for parallel execution
show_elimination	Shows partition elimination
show_missing_stats	Shows details of useful statistics missing from SARG/join columns

set commands for XML format messages

You can regenerate diagnostics as an XML document. This makes it easier for front-end tools to interpret a document. You can use the native XPath query processor inside Adaptive Server to query this output if the XML option is enabled.

Either the query optimizer or the query execution layer can generate diagnostics output. To generate an XML document for the diagnostic output, use this set plan command:

```
set plan for
{show_exec_xml, show_opt_xml, show_execio_xml,
show_lob_xml, show_managers_xml, show_log_props_xml,
show_parallel_xml, show_histograms_xml, show_final_plan_xml,
show_abstract_plan_xml, show_search_engine_xml,
show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
show_ljo_costing_xml, show_elimination_xml}
to {client | message} on
```

Option	Definition
show_exec_xml	Gets the compiled plan output in XML, showing each of the query plan operators.
show_opt_xml	Gets optimizer diagnostic output, which shows the different components such as logical operators, output from the managers, some of the search engine diagnostics, and the best query plan.
show_execio_xml	Gets the plan output along with estimated and actual I/Os. show_execio_xml also includes the query text.
show_lop_xml	Gets the output logical operator tree in XML.
show_managers_xml	Shows the output of the different component managers during the preparation phase of the query optimizer.
show_log_props_xml	Shows the logical properties for a given equivalence class (one or more groups of relations in the query).
show_parallel_xml	Shows the diagnostics related to the optimizer while generating parallel query plans.
show_histograms_xml	Shows diagnostics related to histograms and the merging of histograms.
show_final_plan_xml	Gets the plan output. Does not include the estimated and actual I/Os. show_final_plan_xml includes the query text.
show_abstract_plan_xml	Shows the generated abstract plan.
show_search_engine_xml	Shows diagnostics related to the search engine.
show_counters_xml	Shows plan object construction/destruction counters.
show_best_plan_xml	Shows the best plan in XML.
show_pio_costing_xml	Shows actual physical input/output costing in XML.
show_ljo_costing_xml	Shows actual logical input/output costing in XML.
show_elimination_xml	Shows partition elimination in XML.
client	When specified, output is sent to the client. By default, this is the error log. When trace flag 3604 is active, however, output is sent to the client connection.
message	When specified, output is sent to an internal message buffer.

To turn an option off, specify:

```
set plan for
{show_exec_xml, show_opt_xml, show_execio_xml, show_lop_xml,
show_managers_xml, show_log_props_xml, show_parallel_xml,
show_histograms_xml, show_final_plan_xml
show_abstract_plan_xml, show_search_engine_xml,
show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
show_ljo_costing_xml, show_elimination_xml} off
```

You need not specify the destination stream when turning the option off.

When message is specified, the client application must get the diagnostics from the buffer using a built-in function called showplan_in_xml(query_num).

query_num refers to the number of queries that are cached in the buffer. Currently, a maximum of 20 queries are cached in the buffer. The cache stops collecting query plans when it reaches 20 queries; it ignores the rest of the query plans. However, the message buffer continues to collect query plans. After 20 queries, you can display the message buffer only in its entirety by using a value of 0.

Valid values for *query_num* are 1 – 20, -1, and 0 (zero). A value of -1 refers to the last XML doc in the cache; a value of 0 refers to the entire message buffer.

The message buffer may overflow. If this occurs, there is no way to log all of the XML document, which may result in a partial and invalid XML document.

When the message buffer is accessed using *showplan_in_xml*, the buffer is emptied after execution.

You may want to use *set textsize* to set the maximum text size, as the XML document is printed as a text column and the document is truncated if the column is not large enough. For example, set the text size to 100000 bytes using:

```
set textsize 100000
```

When *set plan* is issued with *off*, all XML tracing is turned off if all of the trace options have been turned off. Otherwise, only specified options are turned off. Other options previously turned on are still valid and tracing continues on the specified destination stream. When you issue another *set plan* option, the previous options are joined with the current options, but the destination stream is switched unconditionally to a new one.

Using *show_execio_xml* to diagnose query plans

show_execio_xml includes diagnostic information that you may find can be helpful for investigating problematic queries. Information from *show_execio_xml* includes:

- The version level of the query plan. Each version of the plan is uniquely identified. This is the first version of the plan:

```
<planVersion>1.0</planVersion>
```

- The statement number in a batch or stored procedure, along with the line number of the statement in the original text. This is statement number 2, but line number 6, in the query:

```
<statementNum>2</statementNum>
```

```
<lineNum>6</lineNum>
```

- The abstract plan for the query. For example, this is the abstract plan for the query `select * from titles`:

```
<abstractPlan>
  <![CDATA[>
    ( i_scan titleidind titles ) ( prop titles ( parallel 1
  ) ( prefetch 8 ) ( lru ) )
]]>
</abstractPlan>
```

- The logical I/O, physical I/O, and CPU costs:

```
<costs>
  <lio> 2 </lio>
  <pio> 2 </pio>
  <cpu> 18 </cpu>
</costs>
```

You can estimate the total costs with this formula (the 25, 2, and 0.1 are constants):

$$25 \times pio + 2 \times lio + 0.1 \times cpu$$

- The estimated execution resource usage, including the number of threads and auxiliary scan descriptors used by the query plan.
- The number of plans the query engine viewed and the plans it determined were valid, the total time the query spent in the query engine (in milliseconds), the time the query engine took to determine the first legal plan, and the amount of procedure cache used during the optimization process.

```
<optimizerMetrics>
  <optTimeMs>6</optTimeMs>
  <optTimeToFirstPlanMs>3</optTimeToFirstPlanMs>
  <plansEvaluated>1</plansEvaluated>
  <plansValid>1</plansValid>
  <procCacheBytes>140231</procCacheBytes>
</optimizerMetrics>
```

- The last time update statistics was run on the current table and whether the query engine used an estimation constant for a given column that it could have estimated better if statistics were available. This section includes information about columns with missing statistics:

```
<optimizerStatistics>
  <statInfo>
    <objName>titles</objName>
```

```

    <columnStats>
      <column>title_id</column>
      <updateTime>Oct  5 2006  4:40:14:730PM</updateTime>
    </columnStats>
    <columnStats>
      <column>title</column>
      <updateTime>Oct  5 2006  4:40:14:730PM</updateTime>
    </columnStats>
  </statInfo>
</optimizerStatistics>

```

- An operator tree that includes table and index scans with information about cache strategies and I/O sizes (inserts, updates, and deletes have the same information for the target table). The operator tree also shows whether updates are performed in “direct” or “deferred” mode. The exchange operator includes information about the number of producer and consumer processes the query used.

```

<TableScan>
  <VA>0</VA>
  <est>
    <rowCnt>18</rowCnt>
    <lio>2</lio>
    <pio>2</pio>
    <rowSz>218.5555</rowSz>
  </est>
  <varNo>0</varNo>
  <objName>titles</objName>
  <scanType>TableScan</scanType>
  <partitionInfo>
    <partitionCount>1</partitionCount>
  </partitionInfo>
  <scanOrder> ForwardScan </scanOrder>
  <positioning> StartOfTable </positioning>
  <dataIOSizeInKB>8</dataIOSizeInKB>
  <dataBufReplStrategy> LRU </dataBufReplStrategy>
</TableScan>

```

Showing cached plans in XML

`show_cached_plan_in_xml` helps track the query performance in the statement cache. For a given query, `show_cached_plan_in_xml`, identified by its object ID or SSQID and PlanID, returns:

- The header section, which contains information about the cache statement, such as the statement ID, object ID and the text:

```
<?xml version="1.0" encoding="UTF-8"?>
<query>
  <statementId>1328134997</statementId>
  <text>
    <![CDATA[SQL Text: select name from sysobjects where id = 10]]>
  </text>
```

If PlanID is set to 0, show_cached_plan_in_xml displays output for all available plans associated with the cached statement.

- The plan section which contains the plan ID and these subsections:
 - Parameter – returns the plan status, the parameters used to compile the query, and the parameter values that caused the slowest performance.

```
<planId>11</planId>
<planStatus> available </planStatus>
<execCount>1371</execCount>
<maxTime>3</maxTime>
<avgTime>0</avgTime>
<compileParameters/>
<execParameters/>
```

- opTree – returns the operators tree, row count, and logical I/O (lio) and physical I/O (pio) estimates for every operator. the opTree subsection returns query plan and optimizer estimates such as lio, pio and row count

This is an example of an output for the Emit operator.

```
<opTree>
  <Emit>
    <VA>1</VA>
    <est>
      <rowCnt>10</rowCnt>
      <lio>0</lio>
      <pio>0</pio>
      <rowSz>22.54878</rowSz>
    </est>
  </act>
  <act>
    <rowCnt>1</rowCnt>
  </act>
  <arity>1</arity>
  <IndexScan>
    <VA>0</VA>
    <est>
```

```

        <rowCnt>10</rowCnt>
        <lio>0</lio>
        <pio>0</pio>
        <rowSz>22.54878</rowSz>
    </est>
    <act>
        <rowCnt>1</rowCnt>
        <lio>3</lio>
        <pio>0</pio>
    </act>
    <varNo>0</varNo>
    <objName>sysobjects</objName>
    <scanType>IndexScan</scanType>
    <indName>csysobjects</indName>
    <indId>3</indId>
    <scanOrder> ForwardScan </scanOrder>
    <positioning> ByKey </positioning>
    <perKey>
        <keyCol>id</keyCol>
        <keyOrder> Ascending </keyOrder>
    </perKey>
    <indexIOSizeInKB>2</indexIOSizeInKB>
    <indexBufReplStrategy> LRU </indexBufReplStrategy>
    <dataIOSizeInKB>2</dataIOSizeInKB>
    <dataBufReplStrategy> LRU </dataBufReplStrategy>
</IndexScan>
</Emit>
<opTree>

```

- execTree – returns the query plan with the operator internal details. Details vary, depending on the operator. This is an example of an output for the Emit operator.

```

<Emit>
<Details>
<VA>5</VA>
    <Vtuple Label="Output Vtuple">
        <collection Label="Columns (#2)">
            <Column>
                <0x0x1462d2838) type:GENERIC_TOKEN len:0 offset:0
valuebuf:0x(nil) status:(0x00000008 (STATNULL))
(constant:0x0x1462d24c0 type:INT4 len:4 maxlen:4 constat: (0x0004
(VARIABLE), 0x0002 (PARAM)))
            </Column>
            <Column>
                <0x0x1462d2878) type:GENERIC_TOKEN len:0 offset:0
valuebuf:0x(nil) status:(0x00000008 (STATNULL))

```



```
(constant:0x0x1462d26e8 type:INT4 len:4 maxlen:4 constat: (0x0004
(VARIABLE), 0x0002 (PARAM))
  </Column>
  <Collection>
    <Collection Label="Evals">
      <EVAL>
        constp: 0x0x1462d2290 status: 0 E_ASSIGN
      </EVAL>
      <EVAL>
        constp: 0x0x1462d2348 status: 0 E_ASSIGN
      </EVAL>
      <EVAL>
        constp: 0x(nil) status: 0 E_END
      </EVAL>
    </Collection>
  </Vtuple>
</Details>
```

Diagnostic usage scenarios

For the following examples, if `dbcc traceon(3604)` is set, trace information is sent to the client's connection. If `dbcc traceon (3605)` is set, trace information is sent to the error log. For Adaptive Server versions 15.0.2 and later, you can use the `set switch on`. For example:

```
set switch on 3604
set switch on 3605
```

Optimization tracing options (`dbcc traceon/off(302,310,317)`) from versions of Adaptive Server earlier than 15.0 are no longer supported.

Use `dbcc traceon(3604)` or `set switch on print_output_to_client` to direct trace output to the client process that would otherwise go to the error log. Use `dbcc traceon(3605)` or `set switch on print_output_to_errorlog` to direct output to the error log as well as to the client process.

Scenario A

To send the execution plan XML to the client as trace output, use:

```
set plan for show_exec_xml to client on
```

Then run the queries for which the plan is wanted:

```
select id from sysindexes where id < 0
```

Scenario B

To get the execution plan, use the `showplan_in_xml` function. You can get the output from the last query, or from any of the first 20 queries in a batch or stored procedure.

```
set plan for show_opt_xml to message on
```

Run the query as:

```
select id from sysindexes where id < 0
select name from sysobjects where id > 0
go
```

```
select showplan_in_xml(0)
go
```

The example generates two XML documents as text streams. You can run an XPath query over this built-in as long as the XML option is enabled in Adaptive Server.

```
select xmlextract("/", showplan_in_xml(-1))
go
```

This allows the XPath query “/” to be run over the XML doc produced by the last query.

Scenario C

To set multiple options:

```
set plan for show_exec_xml, show_opt_xml to client on
go
```

```
select name from sysobjects where id > 0
go
```

This sets up the output from the optimizer and the query execution engine to send the result to the client, as is done in normal tracing.

```
set plan for show_exec_xml off
go
select name from sysobjects where id > 0
go
```

The optimizer’s diagnostics are still available, as `show_opt_xml` is left on.

Scenario D

When running a set of queries in a batch, you can ask for the optimizer plan for the last query.

```
set plan for show_opt_xml to message on
go
declare @v int
select @v = 1
select name from sysobjects where id = @v
```

```
go

select showplan_in_xml(-1)
go
```

`showplan_in_xml()` can also be part of the same batch as it works the same way. Any message for the `showplan_in_xml()` function is ignored for logging.

To create a stored procedure:

```
create proc PP as
declare @v int
select @v = 1
select name from sysobjects where id = @v
go

exec PP
go

select showplan_in_xml(-1)
go
```

If the stored procedure calls another stored procedure, and the called stored procedure compiles, and optimizer diagnostics are turned on, you get the optimizer diagnostics for the new set of statements as well. The same is true if `show_execio_xml` is turned on and only the called stored procedure is executed.

Scenario E

To query the output of the `showplan_in_xml()` function for the query execution plan, which is an XML doc:

```
set plan for show_exec_xml to message on
go

select name from sysobjects
go

select case when
'/Emit/Scan[@Label="Scan:myobjectss"]' xmltest
showplan_in_xml(-1)
then "PASSED" else "FAILED" end
go

set plan for show_exec_xml off
go
```

Scenario F

Use `show_final_plan_xml` to configure Adaptive Server to display the query plan as XML output. This output does not include the actual LIO costs, PIO costs, or the row counts. Once `show_final_plan_xml` is enabled, you can select the query plan from the last run query (which has a query ID of -1). To enable `show_final_plan_xml`:

```
set plan for show_final_plan_xml to message on
```

Run your query, for example:

```
use pubs2
go
select * from titles
go
```

Select the query plan for the last query run using the `showplan_in_xml` parameter:

```
select showplan_in_xml(-1)
```

Permissions for set commands

The `sa_role` has full access to the set commands described above.

For other users, the system administrator must grant and revoke new set tracing permissions to allow set option and set plan for XML, as well as `dbcc traceon/off` (3604,3605).

For more information, see the grant command description in *Adaptive Server Reference Manual: Commands*.

Analyzing dynamic parameters

Adaptive Server lets you analyze dynamic parameters (which are indicated by question marks) before running a query, helping you avoid inefficient query plans.

Analyze dynamic parameters using:

- `@@lwpid` global variable – returns the object ID of the most recently prepared lightweight procedure that corresponds to a dynamic SQL prepare statement.

- `@@plwpid` global variable – returns the object ID of the next most recently prepared lightweight procedure that corresponds to a dynamic SQL prepare statement.
- `show_dynamic_params_in_xml` – displays information about parameters in dynamic SQL statements. See the *Reference Manual: Blocks*.

Using the value provided by `@@plwpid` as the value for the `show_dynamic_params_in_xml object_id` parameter, Adaptive Server displays information about the dynamic parameters in the query. Continue refining the parameters until you find the ones that provide you with the best query plan.

Disable the statement cache before you analyze dynamic parameters for a query. The statement cache reuses query plans to avoid compilation.

The output of `show_dynamic_params_in_xml` is similar to:

```
<!ELEMENT query (parameter*)>
<!ELEMENT parameter (number, type, column?)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT column (#PCDATA)>
```

The root element of the document is `<query>`, which can have zero or more `<parameter>` elements. Each `<parameter>` element includes:

- `number` – the dynamic parameter's position in the statement (starting at 1).
- `type` – the datatype.
- `column` – the name of a table and column (in the format *table.column*) associated with the dynamic parameter, if such an association exists. If no association exists, there is no column information in the output.

Dynamic parameter example analysis

A typical use case for analyzing dynamic parameters is:

- 1 Disable statement cache:

```
set statement_cache off
```
- 2 Prepare the statement to be examined that contains the dynamic parameters.
- 3 Select `@@plwpid` to determine the object ID of the most recent lightweight procedure.

- 4 Run `show_dynamic_params_in_xml` using the value of `@@plwpid` as the value for `object_id`, and displaying the result in the local variable named `xml doc1`.
- 5 Close the statement.
- 6 Enable the statement cache:

```
set statement_cache on
```
- 7 Analyze the results in `xml doc1`, and select values for the parameters.

Finding Slow Running Queries

Topic	Page
Saving diagnostics to a trace file	125
Displaying SQL text	129
Retaining session settings	132

Adaptive Server includes the `set show_sqltext`, `set tracefile`, and `set export_options` parameters that enable you to collect diagnostic information about poorly-running queries without having to previously enable `showplan` or other investigatory parameters.

Saving diagnostics to a trace file

Once enabled, `set tracefile` saves all SQL text for the current session to the specified file, each SQL text batch appending to the previous batch.

The syntax to enable tracing is:

```
set tracefile file_name [off] [for spid]
```

The syntax to disable tracing is:

```
set tracefile off [for spid]
```

Where:

- *file_name* – is the full path to the file in which you are saving the SQL text. If you do not specify a directory path, Adaptive Server creates the file in *\$SYBASE*.

Note If *file_name* contains special characters (":", "/", and so on) other than numbers and letters, you must include *file_name* in quotes. For example, this *file_name* must be in quotes because of the "/" for the directory structure:

```
set tracefile '/tmp/mytracefile.txt' for 25
```

If *file_name* does not contain special characters and you want to save it to *\$SYBASE*, it does not require quotes. For example, this *file_name* does not need to be in quotes:

```
set tracefile mytracefile.txt
```

-
- *off* – disables the tracing for this session or *spid*.
 - *spid* – server process ID whose SQL text you want saved to a trace file. Only the users with the SA or SSO role can enable tracing for other *spids*. You cannot save the SQL text for system tasks (such as the housekeeper or the port manager).

Examples

- This example opens a trace file named *sql_text_file* for the the current session:

```
set tracefile '/var/sybase/REL1502/text_dir/sql_text_file'
```

Subsequent outputs from *set showplan*, *set statistics io*, and *dbcc traceon(100)* are saved in *sql_text_file*.

- This example does not specify a directory path, so the trace file is saved in *\$SYBASE/sql_text_file*:

```
set tracefile 'sql_text_file' for 11
```

Any SQL run on *spid* 11 is saved to this tracefile.

- This example saves the SQL text for *spid* 86:

```
set tracefile  
'/var/sybase/REL1502/text_dir/sql_text_file' for 86
```

- This example disables *set tracefile*:

```
set tracefile off
```

These are the restrictions for *set tracefile*:

- You cannot save the SQL text for system tasks (such as the housekeeper or the port manager).
- You must have the sa or sso roles, or be granted set tracing permission, to run enable or disable tracing.
- set tracefile is not allowed to open an existing file as a tracefile.
- During an SA or SSO session, if you enable set tracefile for a specific spid, all subsequent tracing commands executed take effect on that spid, not the SA or SSO spid.
- If Adaptive Server runs out of file space while writing the tracefile, it closes the file and disables the tracing.
- If an isql session starts tracing for a spid, but the isql session quits without disabling the tracing, another isql session can begin tracing this spid.
- Tracing occurs for the session for which it is enabled only, not for the session that enabled it.
- You cannot trace more than one session at a time from a single sa or sso session. If you attempt to open a tracefile for a session for which there is already a trace file open, Adaptive Server issues this error message:
`tracefile is already open for this session.`
- You cannot trace the same session from multiple sa or sso sessions.
- The file storing the trace output is closed when the session being traced quits or when you disable tracing.
- Before you allocate resources for tracing, keep in mind that each tracing requires one file descriptor per engine.

Set options that save diagnostic information to a trace file

You can use set tracefile in combination with other set commands and options that provide diagnostic information for a better understanding of slow-running queries. These are the set commands and options that save diagnostic information to a file:

- set show_sqltext [on | off]
- set showplan [on | off]
- set statistics io [on | off]
- set statistics time [on | off]

- `set statistics plancost [on | off]`

These are the set options:

- `set option show [normal | brief | long | on | off]`
- `set option show_lop [normal | brief | long | on | off]`
- `set option show_parallel [normal | brief | long | on | off]`
- `set option show_search_engine [normal | brief | long | on | off]`
- `set option show_counters [normal | brief | long | on | off]`
- `set option show_managers [normal | brief | long | on | off]`
- `set option show_histograms [normal | brief | long | on | off]`
- `set option show_abstract_plan [normal | brief | long | on | off]`
- `set option show_best_plan [normal | brief | long | on | off]`
- `set option show_code_gen [normal | brief | long | on | off]`
- `set option show_pio_costing [normal | brief | long | on | off]`
- `set option show_ljo_costing [normal | brief | long | on | off]`
- `set option show_log_props [normal | brief | long | on | off]`
- `set option show_elimination [normal | brief | long | on | off]`

Which sessions are being traced?

Use `sp_helpapptrace` to determine which sessions Adaptive Server is tracing. `sp_helpapptrace` returns the server process IDs (spids) for all the sessions Adaptive Server is tracing, the spids of the sessions tracing them, and the name of the tracefile.

The syntax for `sp_helpapptrace` is:

`sp_helpapptrace`

`sp_helpapptrace` returns these columns:

- `traced_spid` – spid of the session you are tracing.
- `tracer_spid` – spid of the session that `traced_spid` is tracing. Prints “exited” if the `tracer_spid` session has exited.
- `trace_file` – full path to the tracefile.

For example:

sp_helpapptrace		
traced_spid	tracer_spid	trace_file
-----	-----	-----
11	exited	/tmp/myfile1
13	14	/tpcc/sybase.15_0/myfile2

Rebinding a trace

If a session is tracing another session, but quits without disabling the tracing, Adaptive Server allows a new session to rebind with the earlier trace. This means that a sa or sso is not required to finish every trace they start, but can start a trace session, quit, and then rebind to this trace session

Displaying SQL text

set show_sqltext allows you to print the SQL text for ad-hoc queries, stored procedures, cursors, and dynamic prepared statements. You do not need to enable the set show_sqltext before you execute the query (as you do with commands like set showplan on) to collect diagnostic information for a SQL session. Instead you can enable it while the commands are running to help determine which query is performing poorly and diagnose their problems.

Before you enable show_sqltext, you must first enable dbcc traceon to display the output to standard out:

```
dbcc traceon(3604)
```

The syntax for set show_sqltext is:

```
set show_sqltext {on | off}
```

For example, this enables show_sqltext:

```
set show_sqltext on
```

Once set show_sqltext is enabled, Adaptive Server prints all SQL text to standard out for each command or system procedure you enter. Depending on the command or system procedure you run, this output can be extensive.

For example, if you run sp_who, Adaptive Server prints all SQL text associated with this system procedure (the output is abbreviated for space purposes):

```
sp_who
2007/02/23 02:18:25.77
SQL Text: sp_who
Sproc: sp_who, Line: 0
Sproc: sp_who, Line: 20
Sproc: sp_who, Line: 22
Sproc: sp_who, Line: 25
Sproc: sp_who, Line: 27
Sproc: sp_who, Line: 30
Sproc: sp_who, Line: 55
Sproc: sp_who, Line: 64
Sproc: sp_autoformat, Line: 0
Sproc: sp_autoformat, Line: 165
Sproc: sp_autoformat, Line: 167
Sproc: sp_autoformat, Line: 177
Sproc: sp_autoformat, Line: 188
. . .
Sproc: sp_autoformat, Line: 326
Sproc: sp_autoformat, Line: 332
SQL Text: INSERT
#colinfo_af(colid,colname,usertype,type,typename,collength,maxlength,autoformat,selected,selectorder,asname,mbyte) SELECT
c.colid,c.name,t.usertype,t.type,t.name,case when c.length < 80 then 80 else
c.length end,0,0,0,0,c.name,0 FROM tempdb.dbo.syscolumns c,tempdb.dbo.systypes
t WHERE c.id=1949946031 AND c.usertype=t.usertype
Sproc: sp_autoformat, Line: 333
Sproc: sp_autoformat, Line: 334
. . .
Sproc: sp_autoformat, Line: 535
Sproc: sp_autoformat, Line: 0
Sproc: sp_autoformat, Line: 393
Sproc: sp_autoformat, Line: 395
. . .
Sproc: sp_autoformat, Line: 686
Sproc: sp_autoformat, Line: 688
SQL Text: UPDATE #colinfo_af SET maxlength=(SELECT
isnull(max(isnull(char_length(convert(varchar(80),fid)),4)),1) FROM
#whoiresult ), autoformat = 1, mbyte=case when usertype in (24, 25, 34, 35) then
1 else 0 end WHERE colname='fid'
Sproc: sp_autoformat, Line: 689
Sproc: sp_autoformat, Line: 690
. . .
Sproc: sp_autoformat, Line: 815
Sproc: sp_autoformat, Line: 818
SQL Text: SELECT
fid=right(space(80)+isnull(convert(varchar(80),fid),'NULL'),3),
```

```

spid=right(space(80)+isnull(convert(varchar(80),spid),'NULL'),4),
status=SUBSTRING(convert(varchar(80),status),1,8),
loginame=SUBSTRING(convert(varchar(80),loginame),1,8),
origname=SUBSTRING(convert(varchar(80),origname),1,8),
hostname=SUBSTRING(convert(varchar(80),hostname),1,8),
blk_spid=right(space(80)+isnull(convert(varchar(80),blk_spid),'NULL'),8),
dbname=SUBSTRING(convert(varchar(80),dbname),1,6),
tempdbname=SUBSTRING(convert(varchar(80),tempdbname),1,10),
cmd=SUBSTRING(convert(varchar(80),cmd),1,17),
block_xloid=right(space(80)+isnull(convert(varchar(80),block_xloid),'NULL'),1
1) FROM #whoresult order by fid, spid, dbname

```

Sproc: sp_autoformat, Line: 819

Sproc: sp_autoformat, Line: 820

Sproc: sp_autoformat, Line: 826

Sproc: sp_who, Line: 68

Sproc: sp_who, Line: 70

fid	spid	status	loginame	origname	hostname	blk_spid	dbname
tempdbname	cmd	block_xloid					
0	2	sleeping	NULL	NULL	NULL	0	master tempdb
DEADLOCK TUNE		0					
0	3	sleeping	NULL	NULL	NULL	0	master tempdb
ASTC HANDLER		0					
0	4	sleeping	NULL	NULL	NULL	0	master tempdb
CHECKPOINT SLEEP		0					
0	5	sleeping	NULL	NULL	NULL	0	master tempdb
HK WASH		0					
0	6	sleeping	NULL	NULL	NULL	0	master tempdb
HK GC		0					
0	7	sleeping	NULL	NULL	NULL	0	master tempdb
HK CHORES		0					
0	8	sleeping	NULL	NULL	NULL	0	master tempdb
PORT MANAGER		0					
0	9	sleeping	NULL	NULL	NULL	0	master tempdb
NETWORK HANDLER		0					
0	10	sleeping	NULL	NULL	NULL	0	master tempdb
LICENSE HEARTBEAT		0					
0	1	running	sa	sa	echo	0	master tempdb
INSERT		0					

(10 rows affected)

(return status = 0)

To disable `show_sqltext`, enter:

```
set show_sqltext off
```

Restrictions for
`show_sqltext`

- You must have the `sa` or `sso` roles to run `show_sqltext`.
- You cannot use `show_sqltext` to print the SQL text for triggers.
- You cannot use `show_sqltext` to show a binding variable or a view name.

Retaining session settings

Adaptive Server's default behavior is to reset any set parameter changes that are set by a trigger or system procedure after they finish running. Enabling `set export_options` allows you to export the session settings set by a system procedure or trigger to the parent (or issuer) of the stored procedure or trigger.

In this example, Adaptive Server exports `set showplan on` to `outer_proc`, but does not export it to a parent stored procedure:

```
create proc inner_proc
as
    set showplan on
    select * from titles
    set export_options on
go

create proc outer_proc
as
    exec inner_proc
go
```

Parallel Query Processing

This chapter provides an in-depth description of parallel query processing.

Topic	Page
Vertical, horizontal, and pipelined parallelism	133
Queries that benefit from parallel processing	134
Enabling parallelism	135
Controlling parallelism at the session level	139
Controlling query parallelism	141
Using parallelism selectively	141
Using parallelism with large numbers of partitions	143
When parallel query results differ	144
Understanding parallel query plans	146
Adaptive Server parallel query execution model	148

Vertical, horizontal, and pipelined parallelism

Adaptive Server supports horizontal and vertical parallelism for query execution. Vertical parallelism is the ability to run multiple operators simultaneously by employing different system resources such as CPUs, disks, and so on. Horizontal parallelism is the ability to run multiple instances of an operator on the specified portion of the data.

The way you partition your data greatly affects the efficiency horizontal parallelism. The logical partitioning of data is useful in operational decision-support systems (DSS) queries where large volumes of data are being processed.

See Chapter 10, “Partitioning Tables and Indexes,” in the *Transact-SQL User’s Guide* and the *Performance and Tuning Series: Physical Database Tuning* guide for a more detailed discussion of partitioning on Adaptive Server. Understanding different types of partitioning is a prerequisite to understanding this chapter.

Adaptive Server also supports pipelined parallelism. Pipelining is a form of vertical parallelism in which intermediate results are piped to higher operators in a query tree. The output of one operator is used as input for another operator. The operator used as input can run simultaneously with the operator feeding the data, which is an essential element in pipelined parallelism. Use parallelism only when multiple resources like disks and CPUs are available. Using parallelism can be detrimental if your system is not configured for resources that can work in tandem. In addition, data must be spread across disk resources in a way that closely ties the logical partitioning of the data with the physical partitioning on parallel devices. The biggest challenge for a parallel system is to control the correct granularity of parallelism. If parallelism is too finely grained, communication and synchronization overhead can offset any benefit that is obtained from parallel operations. Making parallelism too coarse does not permit proper scaling.

Queries that benefit from parallel processing

When Adaptive Server is configured for parallel query processing, the query optimizer evaluates each query to determine whether it is eligible for parallel execution. If it is eligible, and if the optimizer determines that a parallel query plan can deliver results faster than a serial plan, the query is divided into plan fragments that are processed simultaneously. The results are combined and delivered to the client in a shorter period of time than it takes to process the query serially as a single fragment.

Parallel query processing can improve the performance of:

- select statements that scan large numbers of pages but return relatively few rows, such as table scans or clustered index scans with grouped or ungrouped aggregates.
- Table scans or clustered index scans that scan a large number of pages, but have where clauses that return only a small percentage of rows.
- select statements that include union, order by, or distinct, since these query operations can make use of parallel sorting or parallel hashing.
- select statements where a reformatting strategy is chosen by the optimizer, since these can populate worktables in parallel and can make use of parallel sorting.
- join queries.

Commands that return large, unsorted result sets are unlikely to benefit from parallel processing due to network constraints. In most cases, results can be returned from the database faster than they can be merged and returned to the client over the network.

Parallel DMLs like insert, delete, and update are not supported and so do not benefit from parallelism.

Enabling parallelism

To configure Adaptive Server for parallelism, enable the number of worker processes and max parallel degree parameters.

To gain optimal performance, be aware of other configuration parameters that affect the quality of plans generated by Adaptive Server.

number of worker processes

Before you enable parallelism, configure the number of worker processes (also referred to as threads) available for Adaptive Server by setting the configuration parameter number of worker processes. Sybase recommends that you set the value for number of worker processes to one and a half times the total number required at peak load. You can calculate an approximate number using the max parallel degree configuration parameter, which indicates the total number of worker processes that can be used for any query. Depending on the number of connections to the Adaptive Server and the approximate number of queries that are run simultaneously, you can use this rule to roughly estimate the value for the number of worker processes that may be needed at any time:

$$[\text{number of worker processes}] = [\text{max parallel degree}] \times [\text{the number of concurrent connections wanting to run queries in parallel}] \times [1.5]$$

For example, to set the number of worker processes to 40:

```
sp_configure "number of worker processes", 40
```

Any runtime adjustment for the number of threads may have a negative effect on query performance. Adaptive Server always tries to optimize thread usage, but it may have already committed to a plan that needs increased resources, and therefore does not guarantee a linear scaledown when it runs with fewer threads.

If the query processor has insufficient worker processes, the processor tries to adjust the query plan during runtime. If a minimal number of worker processes are required but unavailable, the query aborts with this error message:

```
Insufficient number of worker processes to execute the
parallel query. Increase the value of the configuration
parameter 'number of worker processes'
```

max parallel degree

Use the max parallel degree configuration parameter to configure the maximum amount of parallelism for a query. This parameter determines the maximum number of threads Adaptive Server uses when processing a given query. For example, to set max parallel degree to 10, enter:

```
sp_configure "max parallel degree", 10
```

Unlike versions of Adaptive Server earlier than 15.0, this parameter's value is not entirely enforced by the query optimizer. A complete enforcement process is expensive in terms of optimization time. Adaptive Server comes close to the desired setting of max parallel degree and exceeds it only for semantic reasons.

max resource granularity

The value of max resource granularity configures the maximum percentage of system resources a query can use. In Adaptive Server version 15.0 and later, max resource granularity affects only procedure cache. By default, max resource granularity is 10%. However, this value is not enforced at execution time; it is only a guide for the query optimizer. The query engine can avoid memory-intensive strategies, such as hash-based algorithms, when max resource granularity is set to a low value.

To set max resource granularity to 5%, enter:

```
sp_configure "max resource granularity", 5
```

If the the query processor's search engine has consumed more than the configured percentage of procedure cache, and if it has found at least one full plan, the search engine times out and uses the current best plan for the query.

If the query processor does not find a full plan before reaching the value for max resource granularity as a percentage of procedure cache, the search engine continues to search until it finds the next full plan. However, if the search engine reaches 50% of the complete procedure cache and finds no plan, it aborts the query compilation to avoid shutting down the server.

max repartition degree

Adaptive Server must dynamically repartition intermediate data to match the partitioning scheme of another operand or to perform an efficient partition elimination. max repartition degree controls the amount of dynamic repartitioning Adaptive Server can do. If the value of max repartition degree is too high, the number of intermediate partitions becomes too large and the system becomes flooded with worker processes that compete for resources, which eventually degrades performance. The value for max repartition degree enforces the maximum number of partitions created for any intermediate data. Repartitioning is a CPU-intensive operation. The value of max repartition degree should not exceed the total number of Adaptive Server engines.

If all tables and indexes are unpartitioned, Adaptive Server uses the value for max repartition degree to provide the number of partitions to create as a result of repartitioning the data. When the value is set to 1, which is the default case, the value of max repartition degree is set to the number of online engines.

Use max repartition degree when using the force option to perform a parallel scan on a table or index.

```
select * from customers (parallel)
```

For example, if the customers table is unpartitioned and the force option is used, Adaptive Server tries to find the inherent partitioning degree of that table or index, which in this case is 1. It uses the number of engines configured for the server, or whatever degree is best based on the number of pages in the table or index that does not exceed the value of max repartition degree.

To set max repartition degree to 5:

```
sp_configure "max repartition degree", 5
```

max scan parallel degree

The max scan parallel degree configuration parameter is used only for backward compatibility, when the data in a partitioned table or index is highly skewed. If the value of this parameter is greater than 1, Adaptive Server uses this value to do a hash-based scan. The value of max scan parallel degree cannot exceed the value of max parallel degree.

prod-consumer overlap factor

prod-consumer overlap factor affects how much pipelined parallelism can be created in a query plan. The default value is 20%, which means that if two operators in a parent-child relationship are run by separate worker processes, there is a 20% overlap. The remaining 80% of the operation is sequential. This affects the way in which Adaptive Server costs two plan fragments. Consider the example of a scan operator under a grouping operation. In such a case, if the scan operator takes $N1$ seconds and grouping operations take $N2$ seconds, the response time of the two operators is:

$$0.2 * \max(N1, N2) + 0.8 * (N1 + N2)$$

In setting this parameter, consider the number of online engines on which Adaptive Server is running and the complexity of the queries to be run. As a general rule, use thread resources to scan on multiple partitions first. Then, if there are unused thread resources, use them to speed up vertical pipelined parallelism. Do not exceed a value of 50.

min pages for parallel scan

max pages for parallel scan controls the tables and indexes that can be accessed in parallel. If the number of pages in a table is below this value, the table is accessed serially. The default value is 200 pages; page size is not relevant. Although the tables and indexes of the table are accessed serially, Adaptive Server tries to repartition the data, if that is appropriate, and to use parallelism above the scans, if that is appropriate.

max query parallel degree

max query parallel degree defines the number of worker processes to use for a given query. This parameter is relevant only if you do not want to enable parallelism globally. You must configure the number of worker processes to a value greater than zero, but max query parallel degree must be set to 1.

When max query parallel degree is set to a value greater than 1, queries are not compiled to use parallelism. Instead, you can specify parallel hints, using abstract plans to compile one or more queries using parallelism.

Use use parallel *N* to define how much parallelism is to be used for a given query. Alternatively, use create plan to specify the query and the number of worker processes to use for it.

Controlling parallelism at the session level

The set options let you restrict the degree of parallelism on a session basis, in stored procedures, or in triggers. These options are useful for tuning experiments with parallel queries and can also be used to restrict noncritical queries to run in serial, so that worker processes remain available for other tasks.

Table 5-1: Session-level parallelism control parameters

Parameter	Function
parallel_degree	Sets the maximum number of worker processes for a query in a session, stored procedure, or trigger. Overrides the max parallel degree configuration parameter, but must be less than or equal to the value of max parallel degree.
scan_parallel_degree	Sets the maximum number of worker processes for a hash-based scan during a specific session, stored procedure, or trigger. Overrides the max scan parallel degree configuration parameter and must be less than or equal to the value of max scan parallel degree.
resource_granularity	Overrides the global value max resource granularity and sets it to a session-specific value, which influences whether Adaptive Server uses memory-intensive operations.
repartition_degree	Sets the value of max repartition degree for a session. This is the maximum degree to which any intermediate data stream is be repartitioned for semantic purposes.

If you specify a value that is too large for any of the set options, the value of the corresponding configuration parameter is used, and a message reports the value that is in effect. While `set parallel_degree`, `set scan_parallel_degree`, `set repartition_degree`, or `set resource_granularity` is in effect during a session, the plans for any stored procedures that you execute are not placed in the procedure cache. Procedures executed with these set options in effect may produce less than optimal plans.

set command examples

This example restricts all queries started in the current session to 5 worker processes:

```
set parallel_degree 5
```

While this command is in effect, any query on a table with more than 5 partitions cannot use a partition-based scan.

To remove the session limit, use:

```
set parallel_degree 0
```

or:

```
set scan_parallel_degree 0
```

To run subsequent queries in serial mode, use:

```
set parallel_degree 1
```

or:

```
set scan_parallel_degree 1
```

To set resource granularity to 25% of the total resources available in the system, use:

```
set resource_granularity 25
```

The same is true for repartition degree as well; you can set it to a value of 5. It cannot, however, exceed the value of max parallel degree.

```
set repartition_degree 5
```

Controlling query parallelism

The parallel extension to the from clause of a select command allows users to suggest the number of worker processes used in a select statement. The degree of parallelism that you specify cannot be more than the value set with `sp_configure` or the session limit controlled by a set command. If you specify a higher value, the specification is ignored, and the optimizer uses the set or `sp_configure` limit.

The syntax for the select statement is:

```
select ...  
from tablename [( [index index_name]  
[parallel [degree_of_parallelism | 1 ]]  
[prefetch size] [lru|mru] ) ] ,  
tablename [( [index index_name]  
[parallel [degree_of_parallelism | 1]  
[prefetch size] [lru|mru] ) ] ...
```

Query-level parallel clause examples

To specify the degree of parallelism for a single query, include `parallel` after the table name. This example executes in serial:

```
select * from huge_table (parallel 1)
```

This example specifies the index to use in the query, and sets the degree of parallelism to 2:

```
select * from huge_table (index ncix parallel 2)
```

Using parallelism selectively

Not all queries benefit from parallelism. In general, the optimizer determines which queries will not benefit from parallelism and attempts to run them serially. When the query processor makes errors in such cases, it is usually because of skewed statistics or incorrect costing as a result of imperfect modeling. Experience will show you whether queries are running better or worse, and you can decide to keep parallel on or off.

If you keep parallel on, and have identified the queries you want to run in serial mode, you can attach an abstract plan hint, as follows:

```
select count(*) from sysobjects
plan "(use parallel 1)"
```

The same effect is achieved by creating a query plan:

```
create plan "select count(*) from sysobjects"
"use parallel 1"
```

If, however, you notice that parallelism is resource-intensive or that it does not generate query plans that perform well, use it selectively. To enable parallelism for selected complex queries:

- 1 Set the number of worker processes to a number greater than zero, based on the guidelines in “number of worker processes” on page 135. For example, to configure 10 worker processes, execute:

```
sp_configure "number of worker processes", 10
```

- 2 Set max query parallel degree to a value greater than 1. As a starting point, set it to what you would have used for max parallel degree:

```
sp_configure "max query parallel degree", 10
```

- 3 The preferred way to force a query to use a parallel plan is to use the abstract plan syntax:

```
use parallel N
```

where *N* is less than the value of max query parallel degree.

To write a query that uses a maximum of 5 threads, use:

```
select count (*), S1.id from sysobjects S1, sysindexes S2
where S1.id = S2.id
group by S1.id
plan
"(use parallel 5)"
```

This query tells the optimizer to use 5 worker processes, if it can. the only drawback to this approach is that you must alter the actual queries in the application. To avoid this, use create plan:

```
create plan
"select count(*), S1.id from sysobjects S1, sysindexes S2
where S1.id = S2.id
group by S1.id"
"(use parallel 5)"
```

To turn the abstract plan load option on globally, enter:

```
sp_configure "abstract plan load", 1
```


See “Creating and Using Abstract Plans” on page 321 for more information about using abstract plans.

Using parallelism with large numbers of partitions

The information in this section also applies when partitioning is configured for manageability, and when partitions are created on physical or logical devices that exhibit little or no parallelism.

For the purpose of this discussion, you have decided to partition a table using range partitioning that represents each week of a year. The issue here is that the query optimizer does not know how the underlying disk system will respond to a 52-way parallel scan. The optimizer must determine the best way to scan the table. If there are enough worker processes configured, the optimizer uses 52 threads to scan the table, which may well cause serious performance issues and be even slower than a serial scan.

To prevent this, first find out exactly how much parallelism is supported. If you know the devices that are used for this table, you can use the following command on a UNIX system, where the underlying device is called */dev/xx*:

```
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &
```

Assume that time records as *x*.

Now run two of the same commands concurrently:

```
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &  
time dd if=/dev/xx of=/dev/null bs=2k skip=8 count = 102400 &
```

This time, assume that time is *y*. In a linear scale-up, *x* is the same as *y*, which is probably impossible to achieve. The following identity may suffice:

$$x \leq y \leq (N \cdot x) / k$$

Where *N* is the number of simultaneous sessions started and *k* is a constant that identifies an acceptable improvement level. A good approximation of *k* might be 1.4, which says that parallel scan is allowed as long as it delivers 40% better metrics than a serial scan.

Table 5-2: Parallel scan metrics

Number of threads	Performance metrics	Acceptable for $k=1.4$
1	200s	
2	245s	$245 \leq (200*2)/1.4$; i.e. $245 \leq 285.71$
4	560s	$560 \leq (200*4)/1.4$; i.e. $560 \leq 571.42$
5	725s	$725 \leq (200*5)/1.4$; i.e. $725 \leq 714.28$

Table 5-2 shows that the disk subsystem did not perform well after four concurrent accesses; the performance numbers went below the acceptable limit established by k . In general, read enough data blocks to allow for any skewed readings.

Having established that 4 threads is optimal, provide this hint by binding it to the object using `sp_chgattribute` in this way:

```
sp_chgattribute <tablename>, "plldegree", 4
```

This tells the query optimizer to use a maximum of four threads. It may choose fewer than four threads if it does not find enough resources. The same mechanism can be applied to an index. For example, if an index called `auth_ind` exists on `authors` and you want to use two threads to access it, use:

```
sp_chgattribute "authors.auth_ind", "plldegree", 4
```

You must run `sp_chgattribute` from the current database.

When parallel query results differ

When a query does not include scalar aggregates or require a final sorting step, a parallel query might return results in a different order from the same query run in serial, and subsequent executions of the same query in parallel might return results in different orders. The relative speed of the different worker processes leads to differences in result-set ordering. Each parallel scan behaves differently, due to pages already in cache, lock contention, and so forth. Parallel queries always return the same set of results, just not in the same order.

Note If you need a dependable ordering of results, use `order by`, or run the query in serial mode.

In addition, due to the pacing effects of multiple worker processes reading data pages, two types of queries accessing the same data may return different results when an aggregate or a final sort is not done. They are:

- Queries that use `set rowcount`
- Queries that select a column into a local variable without sufficiently restrictive query clauses

Queries that use ***set rowcount***

The `set rowcount` option stops processing from continuing after a certain number of rows are returned to the client. With serial processing, the results are consistent in repeated executions as long as the query plans are the same. In serial mode, given the same query plan, the same rows are returned in the same order for a given rowcount value, because a single process reads the data pages in the same order every time. With parallel queries, the order of the results and the set of rows returned can differ, because worker processes may access pages sooner or later than other processes. To get consistent results, either use a clause that performs a final sort step, or run the query in serial.

Queries that set local variables

This query sets the value of a local variable in a `select` statement:

```
select @tid = title_id from titles
where type = "business"
```

The `where` clause matches multiple rows in the `titles` table, so the local variable is always set to the value from the last matching row returned by the query. The value is always the same in serial processing, but for parallel query processing, the results depend on which worker process finishes last. To achieve a consistent result, use a clause that performs a final sort step, execute the query in serial mode, or add clauses so that the query arguments select only single rows.

Understanding parallel query plans

The key to understanding parallel query processing in Adaptive Server is to understand the basic building blocks in a parallel query plan.

Note See Chapter 2, “Using showplan,” which explains how to display a query plan in a text-based format for each SQL statement in a batch or stored procedure.

A compiled query plan contains a tree of execution operators that closely resembles the relational semantics of the query. Each query operator implements a relational operation using a specific algorithm. For example, a query operator called the nested-loop join implements the relational join operation. In Adaptive Server, the primary operator for parallelism is the exchange operator, which is a control operator that does not implement any relational operation. An exchange operator is to create new worker processes that can handle a fragment of the data. During optimization, Adaptive Server strategically places the exchange operator to create operator tree fragments that can run in parallel. All operators found below the exchange operator (down to the next exchange operator) are executed by worker threads that clone the fragment of the operator tree to produce data in parallel. The exchange operator can then redistribute this data to the parent operator above it in the query plan. The exchange operator handles the pipelining and rerouting of data.

In the following sections, the word “degree” is used in two different contexts. When “degree N” of a table or index is referred to, it references the number of partitions contained in a table or index. When the “degree of an operation” or “the degree of a configuration parameter” is referred to, it references the number of partitions generated in the intermediate data stream.

The following example shows how operators in the query processor work in serial with the following query run in the pubs2 database. The table titles is hash-partitioned three ways on the column pub_id.

```
select * from titles
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
1 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```

|SCAN Operator
|  FROM TABLE
|  titles
|  Table Scan.
|  Forward Scan.
|  Positioning at start of table.
|  Using I/O Size 2 Kbytes for data pages.
|  With LRU Buffer Replacement Strategy for data
|  pages.

```

As this example illustrates, the titles table is being scanned by the scan operator, the details of which appear in the showplan output. The emit operator reads the data from the scan operator and sends it to the client application. A given query can create an arbitrarily complex tree of such operators.

When parallelism turned on, Adaptive Server can perform a simple scan in parallel using the exchange operator above the scan operator. exchange produces three worker processes (based on the three partitions), each of which scans the three disjointed parts of the table and sends the output to the consumer process. The emit operator at the top of the tree does not know that the scans are done in parallel.

Example A:

```
select * from titles
```

Executed in parallel by coordinating process and 3 worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```

|EXCHANGE Operator (Merged)
|Executed in parallel by 3 Producer and 1 Consumer processes.

```

```

|
| |EXCHANGE:EMIT Operator
| |
| | |RESTRICT Operator
| | |
| | | |SCAN Operator
| | | |  FROM TABLE
| | | |  titles
| | | |  Table Scan.
| | | |  Forward Scan.
| | | |  Positioning at start of table.

```

				Executed in parallel with a 3-way partition scan.
				Using I/O Size 2 Kbytes for data pages.
				With LRU Buffer Replacement Strategy for data pages.

The operator called EXCHANGE: EMIT is placed under an EXCHANGE operator to funnel data. See “EXCHANGE operator” on page 148.

Adaptive Server parallel query execution model

One of the key components of the parallel query execution model is the EXCHANGE operator. You can see it in the showplan output of a query.

EXCHANGE operator

The EXCHANGE operator marks the boundary between a producer and a consumer operator (the operators below the EXCHANGE operator produce data and those above it consume data). Example A, which showed parallel scan of the titles table (select * from titles), the EXCHANGE: EMIT and the SCAN operator produce data. This is shown briefly.

```
select * from titles
```

The type of query is SELECT.

ROOT:EMIT Operator

```
|EXCHANGE Operator (Merged)
|Executed in parallel by 3 Producer and 1 Consumer
  processes.
|
|  |EXCHANGE:EMIT Operator
|  |
|  |  |RESTRICT Operator
|  |  |
|  |  |  |SCAN Operator
|  |  |  |  FROM TABLE
|  |  |  |  titles
|  |  |  |  Table Scan.
```

In this example, one consumer process reads data from a pipe (which is used as a medium to transfer data across process boundaries) and passes the data to the emit operator, which in turn routes the result to the client. The exchange operator also spawns worker processes, which are called producer threads. The exchange:emit operator writes the data into a pipe managed by the exchange operator.

Figure 5-1: Binding of thread to plan fragments in query plan

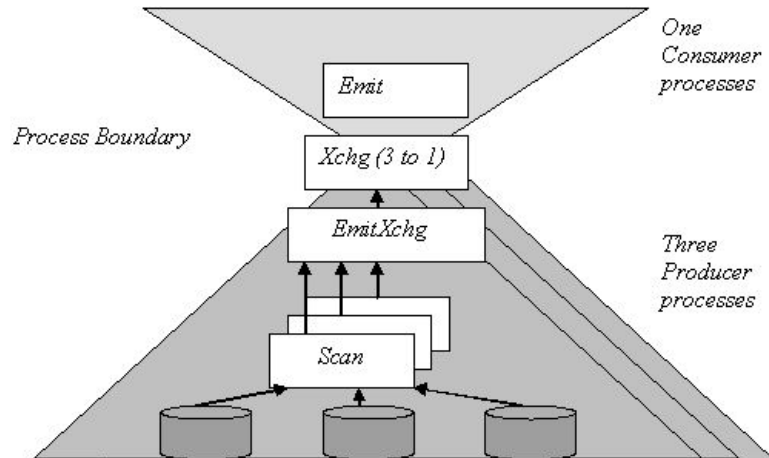


Figure 5-1 shows the process boundary between a producer and a consumer process. There are two plan fragments in this query plan. The plan fragment with the scan and the exchange:emit operators are cloned three ways and then a three-to-one exchange operator writes it into a pipe. The emit operator and the exchange operator are run by a single process, which means there is a single clone of that plan fragment.

Pipe management

The four types of pipes managed by the exchange operator are distinguished by how they split and merge data streams. You can determine which type of pipe is being managed by the exchange operator by looking at its description in the showplan output, where the number of producers and consumers are shown. The four pipe types are described below.

Many-to-one In this case, the exchange operator spawns multiple producer threads and has one consumer task that reads the data from a pipe, to which multiple producer threads write. The exchange operator in the previous example implements a many-to-one exchange. A many-to-one exchange operator can be order-preserving and this technique is employed particularly when doing a parallel sort for an order by clause and the resultant data stream merged to generate the final ordering. The showplan output shows more than one producer process and one consumer process.

```
| EXCHANGE Operator (Merged)
      | Executed in parallel by 3 Producer and 1
      | Consumer processes
```

One-to-many In this case, there is one producer and multiple consumer threads. The producer thread writes data to multiple pipes according to a partitioning scheme devised at query optimization, and then routes data to each of these pipes. Each consumer thread reads data from one of the assigned pipes. This kind of data split can preserve the ordering of the data. The showplan output shows one producer process and more than one consumer processes:

```
| EXCHANGE Operator (Repartitioned)
      | Executed in parallel by 1 Producer
      | and 4 Consumer processes
```

Many-to-many Many-to-many means there are multiple producers and multiple consumers. Each producer writes to multiple pipes, and each pipe has multiple consumers. Each stream is written to a pipe. Each consumer thread reads data from one of the assigned pipes.

```
| EXCHANGE Operator (Repartitioned)
      | Executed in parallel by 3 Producer and 4
      | Consumer processes
```

Replicated exchange operators In this case, the producer thread writes all of its data to each pipe that the exchange operator configures. The producer thread makes a number of copies of the source data (the number is specified by the query optimizer) equal to the number of pipes in the exchange operator. Each consumer thread reads data from one of the assigned pipes. The showplan output shows this as follows:

```
| EXCHANGE (Replicated)
      | Executed in parallel by 3 Producers and 4
      | Consumer processes
```


Worker process model

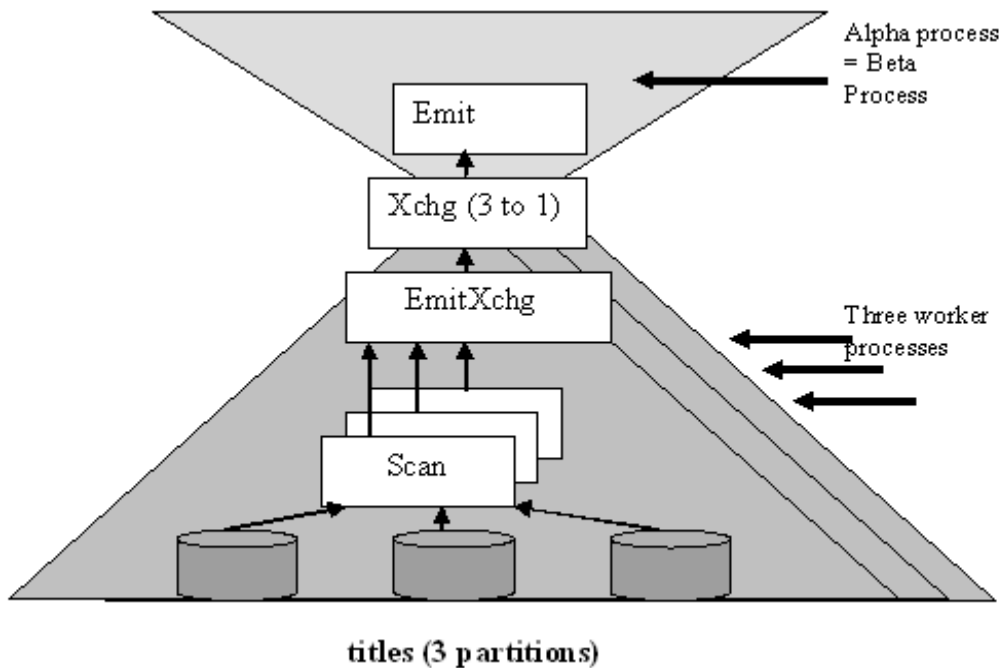
A parallel query plan is composed of different operators, at least one of which is an exchange operator. At runtime, a parallel query plan is bound to a set of server processes that, together, execute the query plan in a parallel fashion.

The server process associated with the user connection is called the *alpha process* because it is the source process from which parallel execution is initiated. In particular, each worker process involved in the execution of the parallel query plan is spawned by the alpha process.

In addition to spawning worker processes, the alpha process initializes all the worker processes involved in the execution of the plan, and creates and destroys the pipes necessary for worker processes to exchange data. The alpha process is, in effect, the global coordinator for the execution of a parallel query plan.

At runtime, Adaptive Server associates each exchange operator in the plan with a set of worker processes. The worker processes execute the query plan fragment located immediately below the exchange operator.

For the query in Example A, represented in “EXCHANGE operator” on page 148, the exchange operator is associated with three worker processes. Each worker process executes the plan fragment made of the exchange:emit operator and of the scan operator.

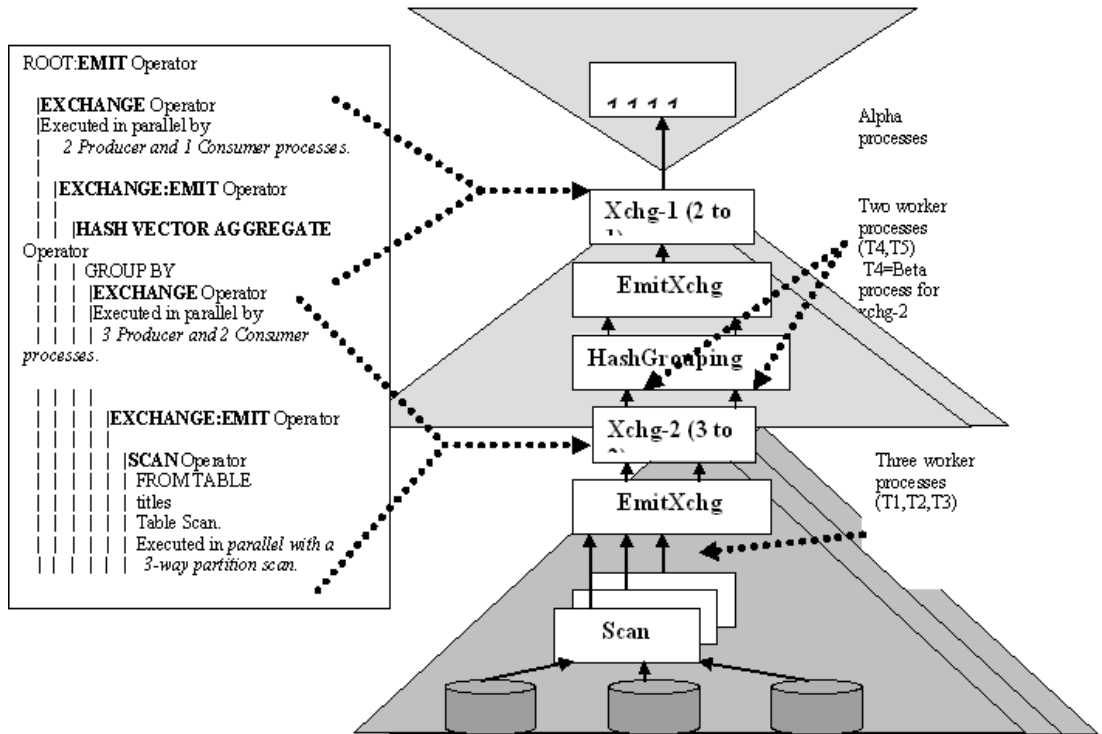
Figure 5-2: Query execution plan with one exchange operator

Each exchange operator is also associated with a server process named the *beta process*, which can be either the alpha process or a worker process. The beta process associated with a given exchange operator is the local coordinator for the execution of the plan fragment below the exchange operator. In the example above, the beta process is the same process as the alpha process, because the plan to be executed has only one level of exchange operators.

Next, use this query to illustrate what happens when the query plan contains multiple exchange operators:

```
select count(*), pub_id, pub_date
from titles
group by pub_id, pub_date
```

Figure 5-3: Query execution plan with two exchange operators



There are two levels of exchange operators marked as EXCHANGE-1 and EXCHANGE-2 in Figure 5-3. Worker process T4 is the beta process associated with the exchange operator EXCHANGE-2.

The beta process locally orchestrates execution of the plan fragment below the exchange operator; it dispatches query plan information that is needed by the worker processes, and synchronizes the execution of the plan fragment.

A process involved in the execution of a parallel query plan that is neither the alpha process nor a beta process is called a *gamma process*.

A given parallel query plan is bound at runtime to a unique alpha process, to one or more beta processes, and to at least one gamma process. Any Adaptive Server parallel plan needs at least two different processes (alpha and gamma) to be executed in parallel.

To find out the mapping between exchange operators and worker processes, as well as to figure out which process is the alpha process, and which processes are the beta processes, use `dbcc traceon(516)`:

```
=====Thread to XCHg Map BEGINS=====
ALFA thread spid: 17
XCHG = 2                                <- refers to Xchg-2
Comp Count = 2 Exec Count = 2
  Range Adjustable
    Consumer XCHG = 5
    Parent thread spid: 34              <- refers to T4
      Child thread 0: spid:37          <- refers to T1
      Child thread 1: spid: 38         <- refers to T2
      Child thread 2: spid: 36         <- refers to T3
    Scheduling level: 0
  XCHG = 5                              <- refers to Xchg-1
  Comp Count = 3 Exec Count = 3
  Bounds Adjustable
    Consumer XCHG -1
    Parent thread spid: 17              <- refers to Alpha
      Child thread 0: spid: 34          <- refers to T4
      Child thread 1: spid: 35         <- refers to T5

Scheduling level: 0
=====Thread to XCHg Map BEGINS=====
```

Using parallelism in SQL operations

Partition tables or indexes in any way that best reflects the needs of your application. Sybase recommends that you put partitions on segments that use different physical disks so that enough I/O parallelism is present. For example, you can have a well-defined partition based on hashing of certain columns of a table, or certain ranges, or a list of values ascribed to a partition. Hash, range, and list partitions belong to the category of “semantic-based” partitioning—given a row, you can determine to which partition the row belongs.

Round-robin partitioning has no semantics associated with its partitioning. A row can occur in any of its partitions. The choice of columns to partition and the type of partitioning used can have a significant impact on the performance of the application. Think of partitions as a low-cardinality index; the columns on which partitioning must be defined are based on the queries in the application.

The query processing engine and its operators take advantage of the Adaptive Server partitioning strategy. Partitioning defined on table and indexes is called static partitioning. In addition, Adaptive Server *dynamically* repartitions data to match the needs for relational operations like joins, vector aggregation, distincts, unions, and so on. Repartitioning is done in streaming mode and no storage is associated with it. Repartitioning is not the same as issuing the alter table repartition command, where static repartitioning is done.

A query plan consists of query execution operators. In Adaptive Server, operators belong to one of two categories:

- Attribute-insensitive operators include scans, union alls, and scalar aggregation. Underlying partitions do not affect attribute-insensitive operators.
- Attribute-sensitive operators (for example, join, distinct, union, and vector aggregation) allow for an operation on a given amount of data to be broken into a smaller number of operations on smaller fragments of the data using semantics-based partitioning. Afterwards, a simple union all provides the final result set. The union all is implemented using a many-to-one exchange operator.

The following sections discuss these two classes of operators. The examples in these sections use the following table with enough data to trigger parallel processing.

```
create table RA2(a1 int, a2 int, a3 int)
```

Parallelism of attribute-insensitive operation

This section discusses the attribute-insensitive operations, which include scans (serial and parallel), scalar aggregations, and union alls.

Table scan

For horizontal parallelism, either at least one of the tables in the query must be partitioned, or the configuration parameter max repartition degree must be greater than 1. If max repartition degree is set to 1, Adaptive Server uses the number of online engines as a hint. When Adaptive Server runs horizontal parallelism, it runs multiple versions of one or more operators in parallel. Each clone of an operator works on its partition, which can be statically created or dynamically built at execution.

Serial table scan

The following example below shows the serial execution of a query where the table RA2 is scanned using the table scan operator. The result of this operation is routed to the emit operator, which forwards the result to the client.

```
select * from RA2
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
1 operator(s) under root
```

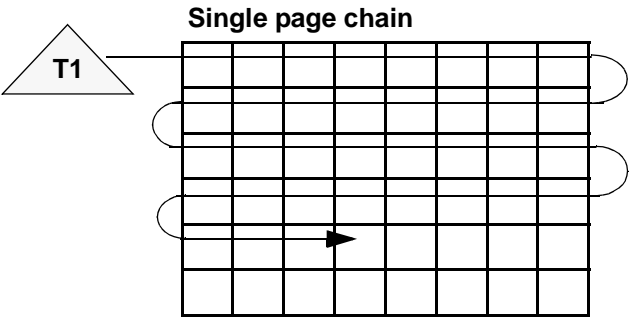
```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```
|SCAN Operator
|  FROM TABLE
|  RA2
|  Table Scan.
|  Forward Scan.
|  Positioning at start of table.
|  Using I/O Size 2 Kbytes for data pages.
|  With LRU Buffer Replacement Strategy for data
|  pages.
```

In versions earlier than 15.0, Adaptive Server did not try to scan an unpartitioned table in parallel using a hash-based scan unless a force option was used. Figure 5-4 shows a scan of an allpages-locked table executed in serial mode by a single task T1. The task follows the page chain of the table to read each page, while doing physical I/O if the needed pages are not in the cache.

Figure 5-4: Serial task scans data pages



Parallel table scan

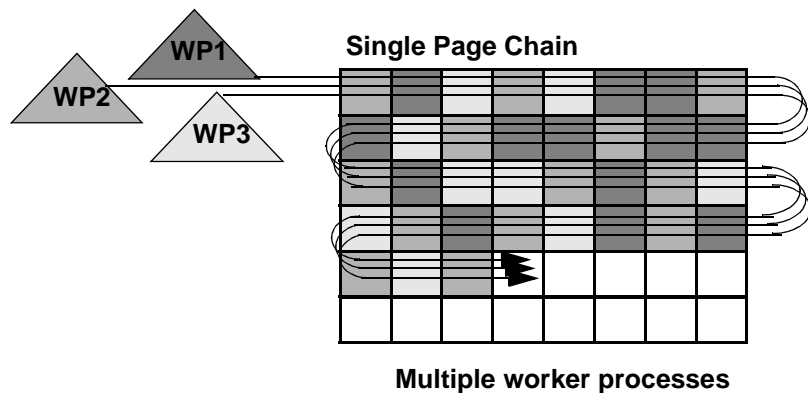
You can force a parallel table scan of an unpartitioned table using the Adaptive Server force option. In this case, Adaptive Server uses a hash-based scan.

Hash-based table scans

Figure 5-5 shows how three worker processes divide the work of accessing data pages from an allpages-locked table during a hash-based table scan. Each worker process performs a logical I/O on every page, but each process examines rows on one-third of the pages, as indicated by differently shaded lines. Hash-based table scans are used only if the user forces a parallel degree. See “Partition skew” on page 198.

With one engine, the query still benefits from parallel access because one work process can execute while others wait for I/O. If there are multiple engines, some of the worker processes can be running simultaneously.

Figure 5-5: Multiple worker processes scans unpartitioned table



Hash-based scans increase the logical I/O for the scan, since each worker process must access each page to hash on the page ID. For a data-only-locked table, hash-based scans hash either on the extent ID or the allocation page ID, so that only a single worker process scans a page and logical I/O does not increase.

Partitioned-based table scans

If you partition this table as follows:

```
alter table RA2 partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

With the following query, Adaptive Server may choose a parallel scan of the table. Parallel scan is chosen only if there are sufficient pages to scan and the partition sizes are similar enough that the query will benefit from parallelism.

```
select * from RA2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 2
```

```
worker processes.

3 operator(s) under root

The type of query is SELECT.

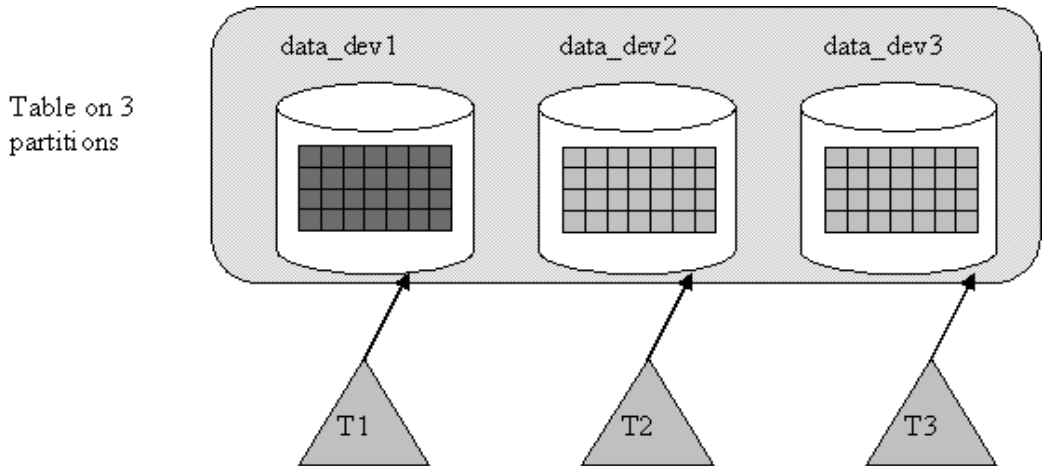
ROOT:EMIT Operator

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1 Consumer
|   processes.

|
| | EXCHANGE:EMIT Operator
| |
| | | SCAN Operator
| | |   FROM TABLE
| | |   RA2
| | |   Table Scan.
| | |   Forward Scan.
| | |   Positioning at start of table.
| | |   Executed in parallel with a 2-way
| | |     partition scan.
| | |   Using I/O Size 2 Kbytes for data pages.
| | |   With LRU Buffer Replacement Strategy
| | |     for data pages.
```

After partitioning the table, showplan output includes two additional operators, exchange and exchange:emit. This query includes two worker processes, each of which scans a given partition and passes the data to the exchange:emit operator, as illustrated in Figure 5-1 on page 149.

Figure 5-6 shows how a query scans a table that has three partitions on three physical disks. With a single engine, this query can benefit from parallel processing because one worker process can execute while others sleep, waiting for I/O or waiting for locks held by other processes to be released. If multiple engines are available, the worker processes can run simultaneously on multiple engines. Such a configuration can perform extremely well.

Figure 5-6: Multiple worker processes access multiple partitions**Index scan**

Indexes, like tables, can be partitioned or unpartitioned. Local indexes inherit the partitioning strategy of the table. Each local index partition scans data in only one partition. Global indexes have a different partitioning strategy from the base table; they reference one or more partitions.

Global nonclustered indexes

Adaptive Server supports global indexes that are nonclustered and unpartitioned for all table partitioning strategies. Global indexes are supported for compatibility with Adaptive Server versions earlier than 15.0; they are also useful in OLTP environments. The index and the data partitions can reside on the same or different storage areas.

Noncovered scan of global nonclustered index using hashing

To create an unpartitioned global nonclustered index on table RA2, which is partitioned by range, enter:

```
create index RA2_NC1 on RA2(a3)
```

This query has a predicate that uses the index key of a3:

```
select * from RA2 where a3 > 300
QUERY PLAN FOR STATEMENT 1 (at line 1).
. . . . .
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

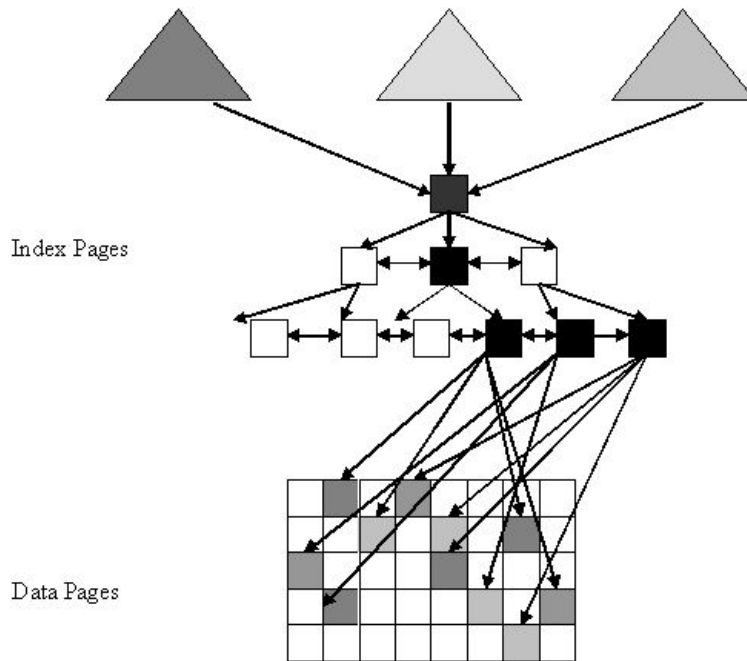
```
| EXCHANGE Operator (Merged)
```

```

| Executed in parallel by 3 Producer and 1
| Consumer processes.
|
| | EXCHANGE:EMIT Operator
| |
| | | SCAN Operator
| | | FROM TABLE
| | | RA2
| | | Index : RA2_NC1
| | | Forward Scan.
| | | Positioning by key.
| | | Keys are:
| | | a3 ASC
| | | Executed in parallel with a 3-way
| | | hash scan.
| | | Using I/O Size 2 Kbytes for index
| | | leaf pages.
| | | With LRU Buffer Replacement Strategy
| | | for index leaf pages.
| | | Using I/O Size 2 Kbytes for data
| | | pages.
| | | With LRU Buffer Replacement Strategy
| | | for data pages.

```

Adaptive Server uses an index scan using the index RA2_NC1 using three producer threads spawned by the exchange operator. Each producer thread scans all qualifying leaf pages and uses a hashing algorithm on the row ID of the qualifying data and accesses the data pages to which it belongs. The parallelism in this case is exhibited at the data page level.

Figure 5-7: Hash-based parallel scan of global nonclustered index

Legend for Figure 5-7:

- Pages read by worker process T1, T2, T3
- Pages read by worker process T1
- Pages read by worker process T2
- Pages read by worker process T3

If the query does not need to access the data page, then it does not execute in parallel. However, the partitioning columns must be added to the query; therefore, it becomes a noncovered scan:

```
select a3 from RA2 where a3 > 300
```

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.
3 operator(s) under root

```

The type of query is SELECT.

ROOT:EMIT Operator

```

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1 Consumer
| processes.

```

```

| | | EXCHANGE:EMIT Operator
| | | |
| | | | SCAN Operator
| | | | FROM TABLE
| | | | RA2
| | | | Index : RA2_NC1
| | | | Forward Scan.
| | | | Positioning by key.
| | | | Keys are:
| | | | a3 ASC
| | | | Executed in parallel with a 2-way hash
| | | | scan.
| | | | Using I/O Size 2 Kbytes for index leaf
| | | | pages.
| | | | With LRU Buffer Replacement Strategy for
| | | | index leaf pages.
| | | | Using I/O Size 2 Kbytes for data pages.
| | | | With LRU Buffer Replacement Strategy for
| | | | data pages.

```

Covered scan using
nonclustered global
index

If there is a nonclustered index that includes the partitioning column, there is no reason for Adaptive Server to access the data pages and the query executes in serial:

```

create index RA2_NC2 on RA2(a3,a1,a2)

select a3 from RA2 where a3 > 300

QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root

The type of query is SELECT.

```

ROOT:EMIT Operator

```

| SCAN Operator
| FROM TABLE
| RA2
| Index : RA2_NC2
| Forward Scan.
| Positioning by key.
| Index contains all needed columns. Base table
| will not be read.
| Keys are:
|   a3 ASC
| Using I/O Size 2 Kbytes for index leaf pages.
| With LRU Buffer Replacement Strategy for index
| leaf pages.

```

Clustered index scans With a clustered index on an all-pages-locked table, a hash-based scan strategy is not permitted. The only allowable strategy is a partitioned scan. Adaptive Server uses a partitioned scan if that is necessary. For a data-only-locked table, a clustered index is usually a placement index, which behaves as a nonclustered index. All discussions pertaining to a nonclustered index on an all-pages-locked table apply to a clustered index on a data-only-locked table as well.

Local indexes Adaptive Server supports clustered and nonclustered local indexes.

Clustered indexes on partitioned tables Local clustered indexes allow multiple threads to scan each data partition in parallel, which can greatly improve performance. To take advantage of this parallelism, use a partitioned clustered index. On a local index, data is sorted separately within each partition. The information in each data partition conforms to the boundaries established when the partitions were created, which makes it possible to enforce unique index keys across the entire table.

Unique, clustered local indexes have the following restrictions:

- Index columns must include all partition columns.
- Partition columns must have the same order as the index definition's partition key.
- Unique, clustered local indexes cannot be included on a round-robin table with more than one partition.

Nonclustered indexes on partitioned tables Adaptive Server supports local, nonclustered indexes on partitioned tables.

There is, however, a slight difference when using local indexes. When doing a covered index scan of a local nonclustered index, Adaptive Server can still use a parallel scan because the index pages are partitioned as well.

To illustrate the difference, this example creates a local nonclustered index:

```
create index RA2_NC2L on RA2(a3,a1,a2) local index
```

```
select a3 from RA2 where a3 > 300
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 2 worker processes.

3 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1 Consumer
| processes.
```

```
|
| | EXCHANGE:EMIT Operator
| |
| | | SCAN Operator
| | | FROM TABLE
| | | RA2
| | | Index : RA2_NC2L
| | | Forward Scan.
| | | Positioning by key.
| | | Index contains all needed columns. Base
| | | table will not be read.
| | | Keys are:
| | | a3 ASC
| | | Executed in parallel with a 2-way
| | | partition scan.
| | | Using I/O Size 2 Kbytes for index leaf
| | | pages.
| | | With LRU Buffer Replacement Strategy
| | | for index leaf pages.
```

Sometimes, Adaptive Server chooses a hash-based scan on a local index. This occurs when a different parallel degree is needed or when the data in the partition is skewed such that a hash-based parallel scan is preferred.

Scalar aggregation

The Transact-SQL scalar aggregation operation can be done in serial or in parallel.

Two-phased scalar aggregation

In a parallel scalar aggregation, the aggregation operation is performed in two phases, using two scalar aggregate operators. In the first phase, the lower scalar aggregation operator performs aggregation on the data stream. The result of scalar aggregation from the first phase is merged using a many-to-one exchange operator, and this stream is aggregated a second time.

In case of a `count(*)` aggregation, the second phase aggregation performs a scalar sum. This is highlighted in the showplan output of the next example.

```
select count(*) from RA2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 2
worker processes.
```

```
5 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```

| SCALAR AGGREGATE Operator
|   Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.
|
|   EXCHANGE Operator (Merged)
|   Executed in parallel by 2 Producer and 1
|   Consumer processes.
|
|   |
|   |   EXCHANGE:EMIT Operator
|   |   |
|   |   |   SCALAR AGGREGATE Operator
|   |   |   Evaluate Ungrouped COUNT AGGREGATE.
|   |   |   |
|   |   |   |   SCAN Operator
|   |   |   |   FROM TABLE
|   |   |   |   RA2
|   |   |   |   Table Scan.
|   |   |   |   Forward Scan.
|   |   |   |   Positioning at start of table.
```

```

| | | | | Executed in parallel with a
| | | | | 2-way partition scan.
| | | | | Using I/O Size 2 Kbytes for data
| | | | | pages.
| | | | | With LRU Buffer Replacement
| | | | | Strategy for data pages.

```

Serial aggregation

Adaptive Server may also choose to do the aggregation in serial. If the amount of data to be aggregated is not enough to guarantee a performance advantage, a serial aggregation may be the preferred technique. In case of a serial aggregation, the result of the scan is merged using a many-to-one exchange operator. This is shown in the example below, where a selective predicate has been added to minimize the amount of data flowing into the scalar aggregate operator. In such a case, it probably does not make sense to do the aggregation in parallel.

```

select count(*) from RA2 where a2 = 10
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

```

```

4 operator(s) under root

```

```

The type of query is SELECT.

```

```

ROOT:EMIT Operator

```

```

| SCALAR AGGREGATE Operator
|   Evaluate Ungrouped COUNT AGGREGATE.
|
|   | EXCHANGE Operator (Merged)
|   | Executed in parallel by 2 Producer
|   | and 1 Consumer processes.

```

```

| | | | | EXCHANGE:EMIT Operator
| | | | |
| | | | |   | SCAN Operator
| | | | |   |   FROM TABLE
| | | | |   |   RA2
| | | | |   |   Table Scan.
| | | | |   |   Forward Scan.
| | | | |   |   Positioning at start of table.
| | | | |   |   Executed in parallel with a 2-way

```



```

                                partition scan.
|   |   |   |   Using I/O Size 2 Kbytes for data
                                pages.
|   |   |   |   With LRU Buffer Replacement
                                Strategy for data pages.

```

union all

union all operators are implemented using a physical operator by the same name. union all is a fairly simple operation and should be used in parallel only when the query is moving a lot of data.

Parallel union all

The only condition to generating a parallel union all is that each of its operands must be of the same degree, irrespective of the type of partitioning they have. The following example (using table HA2) shows a union all operator being processed in parallel. The position of the exchange operator above the union all operator signifies that it is being processed by multiple threads:

```

create table HA2(a1 int, a2 int, a3 int)
partition by hash(a1, a2) (p1, p2)

```

```

select * from RA2
union all
select * from HA2

```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 2 worker processes.

The type of query is SELECT.

ROOT:EMIT Operator

```

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1 Consumer
  processes.

```

```

|
| | EXCHANGE:EMIT Operator
| |
| | | UNION ALL Operator has 2 children.
| | |
| | | | SCAN Operator
| | | | FROM TABLE

```

```

      |   |   |   |   |   RA2
      |   |   |   |   |   Table Scan.
. . . . . |   |   |   |   |   . . . . .
      |   |   |   |   |   Executed in parallel with a 2-way
      |   |   |   |   |   partition scan.
. . . . . |   |   |   |   |   . . . . .
      |   |   |   |   |   |SCAN Operator
      |   |   |   |   |   |FROM TABLE
      |   |   |   |   |   |HA2
      |   |   |   |   |   |Table Scan.
. . . . . |   |   |   |   |   . . . . .
      |   |   |   |   |   |   Executed in parallel with a 2-way
      |   |   |   |   |   |   partition scan.

```

Serial union all

In the next example, the data from each side of the union operator is restricted by selective predicates on either side. The amount of data being sent through the union all operator is small enough that Adaptive Server decides not to run the unions in parallel. Instead, each scan of the tables RA2 and HA2 are organized by putting 2-to-1 exchange operators on each side of the union. The resultant operands are then processed in parallel by the union all operator:

```

select * from RA2
where a2 > 2400
union all
select * from HA2
where a3 in (10,20)
Executed in parallel by coordinating process and 4
worker processes.

```

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```

|UNION ALL Operator has 2 children.
|
|  |EXCHANGE Operator (Merged)
|  |Executed in parallel by 2 Producer and 1
|  |Consumer processes.
|
|  |
|  |  |EXCHANGE:EMIT Operator

```



```
create table RB2(b1 int, b2 int, b3 int)
partition by range(b1,b2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

Then join RB2 with RA2; the scans and the join can be done in parallel without additional repartitioning. Adaptive Server can join the first partition of RA2 with the first partition of RB2, then join the second partition of RA2 with the second partition of RB2. This is called an equipartitioned join and is possible only if the two tables join on columns a1, b1 and a2, b2 as shown below:

```
select * from RA2, RB2
where a1 = b1 and a2 = b2 and a3 < 0

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.
```

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer
|   and 1 Consumer processes.
```

```
|
| | EXCHANGE:EMIT Operator
| |
| | | NESTED LOOP JOIN Operator
| | |   (Join Type: Inner Join)
| | |
| | | | RESTRICT Operator
| | | |
| | | | | SCAN Operator
| | | | |   FROM TABLE
| | | | |   RB2
| | | | |   Table Scan.
| | | | |   Forward Scan.
| | | | |   Positioning at start of table.
| | | | |   Executed in parallel with a
| | | | |     2-way partition scan.
| | |
| | | | RESTRICT Operator
| | | |
| | | | | SCAN Operator
```

					FROM TABLE
					RA2
					Table Scan.
					Forward Scan.
					Positioning at start of table.
					Executed in parallel with a
					2-way partition scan.

The exchange operator is shown above the nested-loop join. This implies that exchange spawns two producer threads: the first scans the first partition of RA2 and RB2 and performs the nested-loop join; the second scans the second partition of RA2 and RB2 to do the nested-loop join. The two threads merge the results using a many-to-one (in this case, two-to-one) exchange operator.

One of the tables with useful partitioning

In this example, the table RB2 is repartitioned to a three-way hash partitioning on column b1 using the alter table command.

```
alter table RB2 partition by hash(b1) (p1, p2, p3)
```

Now, take a slightly modified join query as shown below:

```
select * from RA2, RB2 where a1 = b1
```

The partitioning on table RA2 is not useful because the partitioned columns are not a subset of the joining columns (that is, given a value for the joining column a1, you cannot specify the partition to which it belongs). However, the partitioning on RB2 is helpful because it matches the joining column b1 of RB2. In this case, the query optimizer repartitions table RA2 to match the partitioning of RB2 by using hash partitioning on column a1 of RA2 (the joining column, which is followed by a three-way merge join). The many-to-many (two-to-three) exchange operator above the scan of RA2 does this dynamic repartitioning. The exchange operator above the merge join operator merges the result using a many-to-one (three-to-one, in this case) exchange operator. The showplan output for this query is shown in the following example:

```
select * from RA2, RB2 where a1 = b1
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 5
worker processes.
```

```
10 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```
|EXCHANGE Operator (Merged)
```

```
|Executed in parallel by 3 Producer and 1 Consumer
```

processes.

```

|
| | EXCHANGE:EMIT Operator
|
| | MERGE JOIN Operator (Join Type: Inner
| | Join)
| | Using Worktable3 for internal storage.
| | Key Count: 1
| | Key Ordering: ASC
|
| | SORT Operator
| | Using Worktable1 for internal storage.
|
| | EXCHANGE Operator (Repartitioned)
| | Executed in parallel by 2 Producer
| | and 3 Consumer processes.

```

```

| | | | | | | | | | EXCHANGE:EMIT Operator
| | | | | | | | | | | RESTRICT Operator
| | | | | | | | | | | | SCAN Operator
| | | | | | | | | | | | FROM TABLE
| | | | | | | | | | | | RA2
| | | | | | | | | | | | Table Scan.
| | | | | | | | | | | | Forward Scan.
| | | | | | | | | | | | Positioning at start
| | | | | | | | | | | | of table.
| | | | | | | | | | | | Executed in parallel
| | | | | | | | | | | | with a 2-way
| | | | | | | | | | | | partition scan.

```

```

| | | | | | | | | | SORT Operator
| | | | | | | | | | Using Worktable2 for internal storage.
|
| | | | | | | | | | SCAN Operator
| | | | | | | | | | FROM TABLE
| | | | | | | | | | RB2
| | | | | | | | | | Table Scan.
| | | | | | | | | | Forward Scan.
| | | | | | | | | | Positioning at start of table.
| | | | | | | | | | Executed in parallel with a
| | | | | | | | | | 3-way partition scan.

```

Both tables with
useless partitioning

The next example uses a join where the native partitioning of the tables on both sides is useless. The partitioning on table RA2 is on columns (a1,a2) and that of RB2 is on (b1). The join predicate is on different sets of columns, and the partitioning for both tables does not help at all. One option is to dynamically repartition both sides of the join. By repartitioning table RA2 using a M-to-N (two-to-three) exchange operator, Adaptive Server chooses column a3 of table RA2 for repartitioning, as it is involved in the join with table RB2. For identical reasons, table RB2 is also repartitioned three ways on column b3. The repartitioned operands of the join are equipartitioned with respect to the join predicate, which means that the corresponding partitions from each side will join. In general, when repartitioning needs to be done on both sides of the join operator, Adaptive Server employs a hash-based partitioning scheme.

```
select * from RA2, RB2 where a3 = b3
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 8  
worker processes.
```

```
12 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```
|EXCHANGE Operator (Merged)
```

```
|Executed in parallel by 3 Producer and 1 Consumer  
processes.
```

```
|
| |EXCHANGE:EMIT Operator
| | |MERGE JOIN Operator
| | | (Join Type: Inner Join)
| | | Using Worktable3 for internal storage.
| | | Key Count: 1
| | | Key Ordering: ASC
| | | |SORT Operator
| | | |Using Worktable1 for internal
| | | |storage.
| | | |EXCHANGE Operator (Repartitioned)
| | | |Executed in parallel by 2
| | | |Producer and 3 Consumer
| | | |processes.
```


Replicated join

A replicated join is useful when an index nested-loop join needs to be used. Consider the case where a large table has a useful index on the joining column, but useless partitioning, and joins to a small table that is either partitioned or not partitioned. The small table can be replicated N ways to that of the inner table, where N is the number of partitions of the large table. Each partition of the large table is joined with the small table and, because no exchange operator is needed on the inner side of the join, an index nested-loop join is allowed.

```
create table big_table(b1 int, b2 int, b3 int)
partition by hash(b3) (p1, p2)
```

```
create index big_table_nc1 on big_table(b1)
```

```
create table small_table(s1 int, a2 int, s3 int)
```

```
select * from small_table, big_table
where small_table.s1 = big_table.b1
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 3 worker processes.

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
|EXCHANGE Operator (Merged)
|Executed in parallel by 2 Producer and 1
  Consumer processes.
```

```
|
|  |EXCHANGE:EMIT Operator
|  |
|  |  |NESTED LOOP JOIN Operator (Join Type:
|  |  |  Inner Join)
|  |  |
|  |  |  |EXCHANGE Operator (Replicated)
|  |  |  |Executed in parallel by 1 Producer
|  |  |  |and 2 Consumer processes.
```

```
|  |  |  |
|  |  |  |  |EXCHANGE:EMIT Operator
|  |  |  |
|  |  |  |  |SCAN Operator
|  |  |  |  |  FROM TABLE
```

```

| | | | | small_table
| | | | | Table Scan.
|
|
| | SCAN Operator
| | FROM TABLE
| | big_table
| | Index : big_table_nc1
| | Forward Scan.
| | Positioning by key.
| | Keys are:
| |     b1 ASC
| | Executed in parallel with a
| |     2-way hash scan.

```

Parallel reformatting

Parallel reformatting is especially useful when you are working with a nested-loop join. Usually, reformatting refers to materializing the inner side of a nested join into a worktable, then creating an index on the joining predicate. With parallel queries and nested-loop join, reformatting is also helpful when there is no useful index on the joining column or nested-loop join is the only viable option for a query because of the server/session/query level settings. This is an important option for Adaptive Server. The outer side may have useful partitioning and, if not, it can be repartitioned to create that useful partitioning. But for the inner side of a nested-loop join, any repartitioning means that the table must be reformatted into a worktable that uses the new partitioning strategy. The inner scan of a nested-loop join must then access the worktable.

In this next example, partitioning for tables RA2 and RB2 is on columns (a1, a2) and (b1, b2) respectively. The query is run with merge and hash join turned off for the session.

```
select * from RA2, RB2 where a1 = b1 and a2 = b3
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 12 worker processes.

17 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
|SEQUENCER Operator has 2 children.
|
|   |EXCHANGE Operator (Merged)
```

```

| | Executed in parallel by 4 Producer
| | and 1 Consumer processes.
| |
| | | EXCHANGE:EMIT Operator
| | |
| | | | STORE Operator
| | | | Worktable1 created, in allpages
| | | | locking mode, for REFORMATTING.
| | | | Creating clustered index.
| | | |
| | | | | INSERT Operator
| | | | | The update mode is direct.
| | | | |
| | | | | EXCHANGE Operator
| | | | | (Repartitioned)
| | | | | Executed in parallel by
| | | | | 2 Producer and 4
| | | | | Consumer processes.
| | | | |
| | | | | | EXCHANGE:EMIT Operator
| | | | | |
| | | | | | | RESTRICT Operator
| | | | | | |
| | | | | | | | SCAN Operator
| | | | | | | | FROM TABLE
| | | | | | | | RB2
| | | | | | | | Table Scan.
| | | | | | | | Executed in
| | | | | | | | parallel
| | | | | | | | with a
| | | | | | | | 2-way
| | | | | | | | partition
| | | | | | | | scan.
| | | | | | |
| | | | | | | TO TABLE
| | | | | | | Worktable1.
| | | | | | |
| | | | | | | EXCHANGE Operator (Merged)
| | | | | | | Executed in parallel by 4 Producer
| | | | | | | and 1 Consumer processes.
| | | | | | |
| | | | | | | EXCHANGE:EMIT Operator
| | | | | | |
| | | | | | | | NESTED LOOP JOIN Operator

```

```

      (Join Type: Inner Join)

      |EXCHANGE Operator (Repartitioned)
      |Executed in parallel by 2
      |   Producer and 4 Consumer
      |   processes.

      |EXCHANGE:EMIT Operator

      |RESTRICT Operator

      |SCAN Operator
      |   FROM TABLE
      |   RA2
      |   Table Scan.
      |   Executed in
      |       parallel with
      |       a 2-way
      |       partition scan.

      |SCAN Operator
      |   FROM TABLE
      |   Worktable1.
      |   Using Clustered Index.
      |   Forward Scan.
      |   Positioning by key.

```

The sequence operator executes all of its child operators but the last, before executing the last child operator. In this case, the sequence operator executes the first child operator, which reformats table RB2 into a worktable using a four-way hash partitioning on columns b1 and b3. The table RA2 is also repartitioned four ways to match the stored partitioning of the worktable.

Serial join

Sometimes, it may not make sense to run a join in parallel because of the amount of data that needs to be joined. If you run a query similar to that of the earlier join queries, but now have predicates on each of the tables (RA2 and RB2) such that the amount of data to be joined is not enough, the join may be done in serial mode. In such a case, it does not matter how these tables are partitioned. The query still benefits from scanning the tables in parallel.

```
select * from RA2, RB2 where a1=b1 and a2 = b2
and a3 = 0 and b2 = 20
```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4

worker processes.

11 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```

|MERGE JOIN Operator (Join Type: Inner Join)
|  Using Worktable3 for internal storage.
|    Key Count: 1
|    Key Ordering: ASC
|
|  |SORT Operator
|  |  Using Worktable1 for internal storage.
|  |
|  |  |EXCHANGE Operator (Merged)
|  |  |  Executed in parallel by 2 Producer and
|  |  |    1 Consumer processes.
|  |
|  |  |EXCHANGE:EMIT Operator
|  |  |  |RESTRICT Operator
|  |  |  |  |SCAN Operator
|  |  |  |  |    FROM TABLE
|  |  |  |  |    RA2
|  |  |  |  |    Table Scan.
|  |  |  |  |    Executed in parallel with
|  |  |  |  |      a 2-way partition scan.
|  |  |
|  |  |SORT Operator
|  |  |  Using Worktable2 for internal storage.
|  |  |
|  |  |  |EXCHANGE Operator (Merged)
|  |  |  |  Executed in parallel by 2 Producer and
|  |  |  |    1 Consumer processes.
|  |  |
|  |  |  |EXCHANGE:EMIT Operator
|  |  |  |  |RESTRICT Operator
|  |  |  |  |  |SCAN Operator
|  |  |  |  |  |    FROM TABLE
|  |  |  |  |  |    RB2

```

```

| | | | | | | Table Scan.
| | | | | | | Executed in parallel with
| | | | | | | a 2-way partition scan.

```

Semijoins Semijoins, which result from flattening of in/exist subqueries, behave the same way as regular inner joins. However, replicated joins are not used for semijoins, because an outer row can match more than one time in such a situation.

Outer joins In terms of parallel processing for outer joins, replicated joins are not considered. Everything else behaves in a similar way as regular inner joins. One other point of difference is that no partition elimination is done for any table in an outer join that belongs to the outer group.

Vector aggregation

Vector aggregation refers to queries with group-bys. There are different ways Adaptive Server can perform vector aggregation. The actual algorithms are not described here; only the technique for parallel evaluation is shown in the following sections.

In-partitioned vector aggregation If any base or intermediate relation requires a grouping and is partitioned on a subset, or the same columns as that of the columns in the group by clause, the grouping operation can be done in parallel on each of the partitions and the resultant grouped streams merged using a simple N-to-1 exchange. This is because a given group cannot appear in more than one stream. The same restriction applies to grouping over any SQL query as long as you use semantics-based partitioning on the grouping columns or a subset of them. This method of parallel vector aggregation is called in-partitioned aggregation.

The following query uses a parallel in-partitioned vector aggregation since range partitioning is defined on the columns a1 and a2, which also happens to be the column on which the aggregation is needed.

```

select count(*), a1, a2 from RA2 group by a1,a2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

```

```

4 operator(s) under root

```

```

The type of query is SELECT.

```

```

ROOT:EMIT Operator

```

```

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and

```

1 Consumer processes.

```

|
| | EXCHANGE:EMIT Operator
| |
| | | HASH VECTOR AGGREGATE Operator
| | | GROUP BY
| | | Evaluate Grouped COUNT AGGREGATE.
| | | Using Worktable1 for internal storage.
| | |
| | | | SCAN Operator
| | | | FROM TABLE
| | | | RA2
| | | | Table Scan.
| | | | Forward Scan.
| | | | Positioning at start of table.
| | | | Executed in parallel with a 2-way
| | | | partition scan.
| | | | Using I/O Size 2 Kbytes for data
| | | | pages.
| | | | With LRU Buffer Replacement
| | | | Strategy for data pages.

```

Repartitioned vector aggregation

Sometimes, the partitioning of the table or the intermediate results may not be useful for the grouping operation. It may still be worthwhile to do the grouping operation in parallel by repartitioning the source data to match the grouping columns, then applying the parallel vector aggregation. Such a scenario is shown below, where the partitioning is on columns (a1, a2), but the query requires a vector aggregation on column a1.

```
select count(*), a1 from RA2 group by a1
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 4 worker processes.

6 operator(s) under root

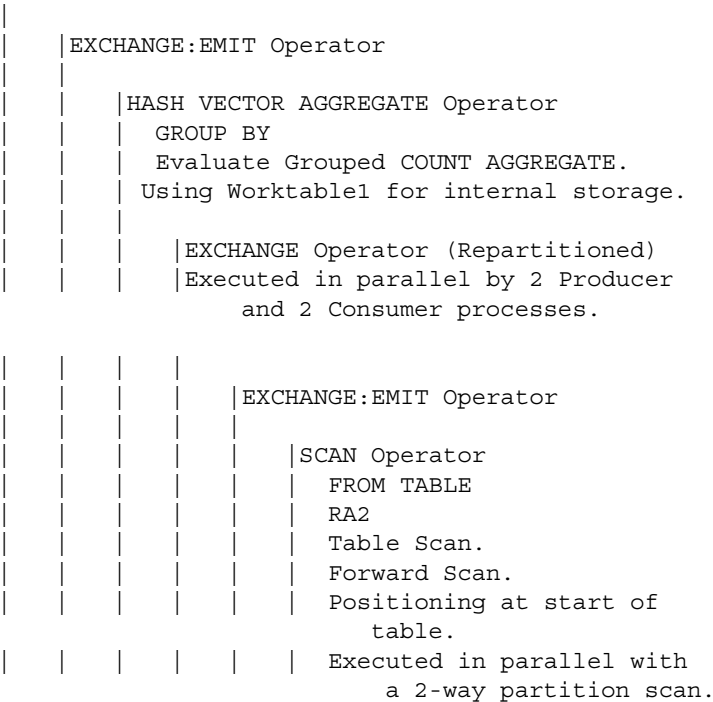
The type of query is SELECT.

ROOT:EMIT Operator

```

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1 Consumer
processes.

```



Two-phased vector aggregation

For the query in the previous example, repartitioning may be expensive. Another possibility is to do a first level of grouping, merge the data using a N-to-1 exchange operator, then do another level of grouping. This is called a *two-phased* vector aggregation. Depending on the number of duplicates for the grouping column, Adaptive Server can reduce the cardinality of the data streaming through the N-to-1 exchange, which reduces the cost of the second level of grouping.

```
select count(*), a1 from RA2 group by a1

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

5 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

|HASH VECTOR AGGREGATE Operator
```



```

| GROUP BY
| Evaluate Grouped SUM OR AVERAGE AGGREGATE.
| Using Worktable2 for internal storage.
|
| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and
|   1 Consumer processes.
|
| EXCHANGE:EMIT Operator
|
| HASH VECTOR AGGREGATE Operator
|   GROUP BY
|   Evaluate Grouped COUNT AGGREGATE.
|   Using Worktable1 for internal
|   storage.
|
| SCAN Operator
|   FROM TABLE
|   RA2
|   Table Scan.
|   Executed in parallel with
|     a 2-way partition scan.

```

Serial vector
aggregation

As with some of the earlier examples, if the amount of data flowing into the grouping operator is restricted by using a predicate, executing that query in parallel may not make much sense. In such a case, the partitions are scanned in parallel and an N-to-1 exchange operator is used to serialize the stream followed by a serial vector aggregation:

```

select count(*), a1, a2 from RA2
where a1 between 100 and 200
group by a1, a2

```

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and
  2 worker processes.

```

```

4 operator(s) under root

```

The type of query is SELECT.

```

ROOT:EMIT Operator

```

```

| HASH VECTOR AGGREGATE Operator
|   GROUP BY
|   Evaluate Grouped COUNT AGGREGATE.

```

```

| Using Worktable1 for internal storage.
|
| | EXCHANGE Operator (Merged)
| | Executed in parallel by 2 Producer and 1
| | Consumer processes.
|
| | | EXCHANGE:EMIT Operator
| | |
| | | | SCAN Operator
| | | | FROM TABLE
| | | | RA2
| | | | Positioning at start of table.
| | | | Executed in parallel with a 2-way
| | | | partition scan.

```

You cannot always group on the partitioning columns, or take advantage of a table that is already partitioned on the grouping columns. The query optimizer determines if it is better to repartition and perform the grouping in parallel, or merge the data stream in a partitioned table and do the grouping in serial or a two-phased aggregation.

distinct

Queries with distinct operations are the same as grouped vector aggregation without the aggregation part. For example:

```
select distinct a1, a2 from RA2
```

is same as:

```
select a1, a2 from RA2 group by a1, a2
```

All of the methodologies that are applicable to vector aggregates are applicable here as well.

Queries with an *in* list

Adaptive Server uses an optimized technique to handle an in list. This is a common SQL construct. So, a construct like:

```
col in (value1, value2,..valuek)
```

is same as:

```
col = value1 OR col = value2 OR .... col = valuek
```

The values in the in list are put into a special in-memory table and sorted for removal of duplicates. The table is then joined back with the base table using an index nested-loop join. The following example illustrates this with two values in the in list that correspond to two values in the or list:

```

SCAN Operator
FROM OR List
OR List has up to 2 rows of OR/IN values.

```

```
select * from RA2 where a3 in (1425, 2940)
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Executed in parallel by coordinating process and 2
worker processes.

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

```
| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1
|   Consumer processes.
```

```
|
| | EXCHANGE:EMIT Operator
| |
| | | NESTED LOOP JOIN Operator (Join Type:
| | |   Inner Join)
```

```
| | | | SCAN Operator
| | | |   FROM OR List
| | | |   OR List has up to 2 rows of OR/IN
| | | |     values.
```

```
| | | | | RESTRIC Operator
| | | | |
| | | | | | SCAN Operator
| | | | | |   FROM TABLE
| | | | | |   RA2
| | | | | |   Index : RA2_NC1
| | | | | |   Forward Scan.
| | | | | |   Positioning by key.
| | | | | |   Keys are:
| | | | | |     a3 ASC
| | | | | |   Executed in parallel with a
| | | | | |     2-way hash scan.
```

Queries with or clauses

Adaptive Server takes a disjunctive predicate like an or clause and applies each side of the disjunction separately to qualify a set of row IDs (RIDs). The set of conjunctive predicates on each side of the disjunction must be indexable. Also, the conjunctive predicates on each side of the disjunction cannot have further disjunction within them; that is, it makes little sense to use an arbitrarily deep nesting of disjunctive and conjunctive clauses. In the next example, a disjunctive predicate is taken on the same column (you can have predicates on different columns as long as you have indexes that can do inexpensive scans), but the predicates may qualify an overlapping set of data rows. Adaptive Server uses the predicates on each side of the disjunction separately and qualifies a set of row IDs. These row IDs are then subjected to duplicate elimination.

```
select a3 from RA2 where a3 = 2955 or a3 > 2990
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).  
Executed in parallel by coordinating process and 2  
worker processes.
```

```
8 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```
| EXCHANGE Operator (Merged)  
| Executed in parallel by 2 Producer and 1  
| Consumer processes.
```

```
|  
| | EXCHANGE:EMIT Operator  
| | | RID JOIN Operator  
| | | Using Worktable2 for internal storage.  
| | | HASH UNION Operator has 2 children.  
| | | Using Worktable1 for internal storage.  
| | | SCAN Operator  
| | | FROM TABLE  
| | | RA2  
| | | Index : RA2_NC1  
| | | Forward Scan.  
| | | Positioning by key.  
| | | Index contains all needed  
| | | columns.Base table will not  
| | | be read.
```

```

| | | | | Keys are:
| | | | | a3 ASC
| | | | | Executed in parallel with a
| | | | | 2-way hash scan.

| | | | | SCAN Operator
| | | | | FROM TABLE
| | | | | RA2
| | | | | Index : RA2_NC1
| | | | | Forward Scan.
| | | | | Positioning by key.
| | | | | Index contains all needed
| | | | | columns. Base table will
| | | | | not be read.

| | | | | Keys are:
| | | | | a3 ASC
| | | | | Executed in parallel with a
| | | | | 2-way hash scan.

| | | | | RESTRICT Operator

| | | | | SCAN Operator
| | | | | FROM TABLE
| | | | | RA2
| | | | | Using Dynamic Index.
| | | | | Forward Scan.
| | | | | Positioning by Row Identifier
| | | | | (RID.)
| | | | | Using I/O Size 2 Kbytes for
| | | | | data pages.
| | | | | With LRU Buffer Replacement
| | | | | Strategy for data pages.

```

Two separate index scans are employed using the index RA2_NC1, which is defined on the column a3. The qualified set of row IDs are then checked for duplicate row IDs, and finally, joined back to the base table. Note the line Positioning by Row Identifier (RID). You can use different indexes for each side of the disjunction, depending on what the predicates are, as long as they are indexable. One way to easily identify this is to run the query separately with each side of the disjunction to make sure that the predicates are indexable. Adaptive Server may not choose an index intersection if it seems more expensive than a single scan of the table.

Queries with an *order by* clause

If a query requires sorted output because of the presence of an *order by* clause, Adaptive Server can apply the sort in parallel. First, Adaptive Server tries to avoid the sort if there is some inherent ordering available. If Adaptive Server is forced to do the sort, it sees if the sort can be done in parallel. To do that, Adaptive Server may repartition an existing data stream or it may use the existing partitioning scheme, then apply the sort to each of the constituent streams. The resultant data is merged using an N-to-1 order, preserving the exchange operator.

```
select * from RA2 order by a1, a2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 2
worker processes.
```

```
4 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```

```
| EXCHANGE Operator (Merged)
```

```
| Executed in parallel by 2 Producer and
    1 Consumer processes.
```

```
|
| | EXCHANGE:EMIT Operator
| | |
| | | SORT Operator
| | | Using Worktable1 for internal storage.
| | |
| | | | SCAN Operator
| | | | FROM TABLE
| | | | RA2
| | | | Index : RA2_NC2L
| | | | Forward Scan.
| | | | Positioning at index start.
| | | | Executed in parallel with a
| | | | 2-way partition scan.
```

Depending upon the volume of data to be sorted, and the available resources, Adaptive Server may repartition the data stream to a higher degree than the current degree of the stream, so that the sort operation is faster. The degrees of sorting depends on whether the benefit obtained from doing the sort in parallel far outweighs the overheads of repartitioning.


```

| | | | | | | SCAN Operator
| | | | | | | FROM TABLE
| | | | | | | RA2
| | | | | | | Index : RA2_NC2L
| | | | | | | Forward Scan.
| | | | | | | Executed in parallel with
| | | | | | | a 2-way partition scan.
| | |
| | | Run subquery 1 (at nesting level 1).
| | |
| | | QUERY PLAN FOR SUBQUERY 1 (at nesting
| | | level 1 and at line 2).
| | |
| | | Correlated Subquery.
| | | Subquery under an EXISTS predicate.
| | |
| | | | SCALAR AGGREGATE Operator
| | | | Evaluate Ungrouped ANY AGGREGATE.
| | | | Scanning only up to the first
| | | | qualifying row.
| | |
| | | | | SCAN Operator
| | | | | FROM TABLE
| | | | | RB2
| | | | | Table Scan.
| | | | | Forward Scan.
| | |
| | | END OF QUERY PLAN FOR SUBQUERY 1.

```

The following example shows an in subquery flattened into a semijoin. Adaptive Server converts this into an inner join to provide greater flexibility in shuffling the tables in the join order. As seen below, the table RB2, which was originally in the subquery, is now being accessed in parallel.

```
select * from RA2 where a1 in (select b1 from RB2)
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
Executed in parallel by coordinating process and 5
worker processes.
```

```
10 operator(s) under root
```

```
The type of query is SELECT.
```

```
ROOT:EMIT Operator
```


| EXCHANGE Operator (Merged)
 | Executed in parallel by 3 Producer and 1 Consumer
 processes.

```

|
| | EXCHANGE:EMIT Operator
| |
| | | MERGE JOIN Operator (Join Type: Inner Join)
| | |   Using Worktable3 for internal storage.
| | |   Key Count: 1
| | |   Key Ordering: ASC
| | |
| | | | SORT Operator
| | | |   Using Worktable1 for internal
| | | |   storage.
| | |
| | | | SCAN Operator
| | | |   FROM TABLE
| | | |   RB2
| | | |   Table Scan.
| | | |   Executed in parallel with a
| | | |   3-way partition scan.
| | |
| | | | SORT Operator
| | | |   Using Worktable2 for internal
| | | |   storage.
| | |
| | | | EXCHANGE Operator (Merged)
| | | |   Executed in parallel by 2
| | | |   Producer and 3 Consumer
| | | |   processes.

```

```

| | | | | EXCHANGE:EMIT Operator
| | | | |
| | | | | | RESTRICT Operator
| | | | | |
| | | | | | | SCAN Operator
| | | | | | |   FROM TABLE
| | | | | | |   RA2
| | | | | | |   Index : RA2_NC2L
| | | | | | |   Forward Scan.
| | | | | | |   Positioning at
| | | | | | |   index start.
| | | | | | |   Executed in
| | | | | | |   parallel with

```

a 2-way
partition scan.

select into clauses

Queries with `select into` clauses create a new table in which to store the query's result set. Adaptive Server optimizes the base query portion of a `select into` command in the same way it does a standard query, considering both parallel and serial access methods. A `select into` statement that is executed in parallel:

- Creates the new table using the columns specified in the `select into` statement.
- Creates N partitions in the new table, where N is the degree of parallelism that the optimizer chooses for the insert operation in the query.
- Populates the new table with query results, using N worker processes.
- Unpartitions the new table, if no specific destination partitioning is required.

Performing a `select into` statement in parallel requires more steps than an equivalent serial query plan. This is a simple `select into` done in parallel:

```
select * into RAT2 from RA2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).  
Executed in parallel by coordinating process and 2  
worker processes.
```

```
4 operator(s) under root
```

```
The type of query is INSERT.
```

```
ROOT:EMIT Operator
```

```
| EXCHANGE Operator (Merged)  
| Executed in parallel by 2 Producer and 1  
| Consumer processes.
```

```
|  
| | EXCHANGE:EMIT Operator  
| | | INSERT Operator  
| | | The update mode is direct.  
| | | SCAN Operator
```

```

      |      |      |      |      FROM TABLE
      |      |      |      |      RA2
      |      |      |      |      Table Scan.
      |      |      |      |      Forward Scan.
      |      |      |      |      Positioning at start of table.
      |      |      |      |      Executed in parallel with a 2-way
      |      |      |      |      partition scan.

      |      |      |      |      TO TABLE
      |      |      |      |      RAT2
      |      |      |      |      Using I/O Size 2 Kbytes for data
      |      |      |      |      pages.

```

Adaptive Server does not try to increase the degree of the stream coming from the scan of table RA2, and uses it to do a parallel insert into the destination table. The destination table is initially created using round-robin partitioning of degree two. After the insert, the table is unpartitioned.

If the data set to be inserted is not big enough, Adaptive Server may choose to insert this data in serial. The scan of the source table can still be done in parallel. The destination table is then created as an unpartitioned table.

The `select into` allows destination partitioning to be specified. In such a case, the destination table is created using that partitioning, and Adaptive Server finds the most optimal way to insert data. If the destination table must be partitioned the same way as the source data, and there is enough data to insert, the insert operator executes in parallel.

The next example shows the same partitioning for source and destination table, and demonstrates that Adaptive Server recognizes this scenario and chooses not to repartition the source data.

```

select * into new_table
partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
from RA2

```

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

```

```

4 operator(s) under root

```

The type of query is INSERT.

ROOT:EMIT Operator

```
|EXCHANGE Operator (Merged)
|Executed in parallel by 2 Producer and 1 Consumer
processes.
```

```
|
|EXCHANGE:EMIT Operator
|
|INSERT Operator
|The update mode is direct.
|
|SCAN Operator
|FROM TABLE
|RA2
|Table Scan.
|Forward Scan.
|Positioning at start of table.
|Executed in parallel with a 2-way
|partition scan.
|
|TO TABLE
|RRA2
|Using I/O Size 16 Kbytes for data
|pages.
```

If the source partitioning does not match that of the destination table, the source data must be repartitioned. This is illustrated in the next example, where the insert is done in parallel using two worker processes after the data is repartitioned using a 2-to-2 exchange operator that converts the data from range partitioning to hash partitioning.

```
select * into HHA2
partition by hash(a1, a2)
(p1, p2)
from RA2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4
worker processes.
```

```
6 operator(s) under root
```

```
The type of query is INSERT.
```

```
ROOT:EMIT Operator
```

```

| EXCHANGE Operator (Merged)
| Executed in parallel by 2 Producer and 1
|   Consumer processes.

|
| | EXCHANGE:EMIT Operator
|
| | | INSERT Operator
| | |   The update mode is direct.
|
| | | | EXCHANGE Operator EXCHANGE Operator (
| | | |   Merged)
| | | | Executed in parallel by 2 Producer
| | | |   and 2 Consumer processes.

| | | | | EXCHANGE:EMIT Operator
| | | | |
| | | | | | SCAN Operator
| | | | | |   FROM TABLE
| | | | | |   RA2
| | | | | |   Table Scan.
| | | | | |   Forward Scan.
| | | | | |   Positioning at start of table.
| | | | | |   Executed in parallel with a
| | | | | |     2-way partition scan.

| | | | TO TABLE
| | | | HHA2
| | | | Using I/O Size 16 Kbytes for data
| | | |   pages.

```

insert/delete/update

insert, delete, and update operations are done in serial in Adaptive Server. However, tables other than the destination table used in the query to qualify rows to be deleted or updated, can be accessed in parallel.

```

delete from RA2
where exists
(select * from RB2
where RA2.a1 = b1 and RA2.a2 = b2)

```

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3
worker processes.

```

9 operator(s) under root

The type of query is DELETE.

ROOT:EMIT Operator

```

|DELETE Operator
|  The update mode is deferred.
|
|  |NESTED LOOP JOIN Operator (Join Type: Inner Join)
|  |
|  |  |SORT Operator
|  |  |  Using Worktable1 for internal storage.
|  |  |
|  |  |  |EXCHANGE Operator (Merged)
|  |  |  |Executed in parallel by 3 Producer
|  |  |  |and 1 Consumer processes.
|  |  |
|  |  |  |EXCHANGE:EMIT Operator
|  |  |  |
|  |  |  |  |RESTRICT Operator
|  |  |  |  |
|  |  |  |  |  |SCAN Operator
|  |  |  |  |  |  FROM TABLE
|  |  |  |  |  |  RB2
|  |  |  |  |  |  Table Scan.
|  |  |  |  |  |  Forward Scan.
|  |  |  |  |  |  Positioning at start of
|  |  |  |  |  |  table.
|  |  |  |  |  |  Executed in parallel with
|  |  |  |  |  |  a 3-way partition scan.
|  |  |  |  |  |  Using I/O Size 2 Kbytes
|  |  |  |  |  |  for data pages.
|  |  |  |  |  |  With LRU Buffer Replacement
|  |  |  |  |  |  Strategy for data pages.
|  |  |
|  |  |  |RESTRICT Operator
|  |  |  |
|  |  |  |  |SCAN Operator
|  |  |  |  |  FROM TABLE
|  |  |  |  |  RA2
|  |  |  |  |  Index : RA2_NC1
|  |  |  |  |  Forward Scan.
|  |  |  |  |  Positioning by key.

```

```

|   |   |   |   Keys are:
|   |   |   |   a3 ASC
|
|   TO TABLE
|   RA2
|   Using I/O Size 2 Kbytes for data pages.

```

The table RB2, which is being deleted, is scanned and deleted in serial. However, table RA2 was scanned in parallel. The same scenario is true for update or insert statements.

Partition elimination

One of the advantages of semantic partitioning is that the query processor may be able to take advantage of this and be able to disqualify range, hash, and list partitions at compile time. With hash partitions, only equality predicates can be used, whereas for range and list partitions, equality and in-equality predicates can be used to eliminate partitions. For example, consider table RA2 with its semantic partitioning defined on columns a1, a2 where (p1 values <= (500,100) and p2 values <= (1000, 2000)). If there are predicates on columns a1 or columns a1, a2, then it would be possible to do some partition elimination. For example, this statement does not qualify any data:

```
select * from RA2 where a1 > 1500
```

You can see this in the showplan output.

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```

.....
|   |   |   |   SCAN Operator
|   |   |   |   FROM TABLE
|   |   |   |   RA2
|   |   |   |   [ Eliminated Partitions : 1 2 ]
|   |   |   |   Index : RA2_NC2L

```

The phrase `Eliminated Partitions` identifies the partition in accordance with how it was created and assigns an ordinal number for identification. For table RA2, the partition represented by p1 where (a1, a2) <= (500, 100) is considered to be partition number one and p2 where (a1, a2) > (500, 100) and <= (1000, 2000) is identified as partition number two.

Consider an equality query on a hash-partitioned table where all keys in the hash partitioning have an equality clause. This can be shown by taking table HA2, which is hash-partitioned two ways on columns (a1, a2). The ordinal numbers refer to the order in which partitions are listed in the output of `sp_help`.

```
select * from HA2 where a1 = 10 and a2 = 20
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
.....
```

```
| SCAN Operator
|   FROM TABLE
|   HA2
|   [ Eliminated Partitions : 1 ]
|   Table Scan.
```

Partition skew

Partition skew plays an important part in determining whether a parallel partitioned scan can be used. Adaptive Server partition skew is defined as the ratio of the size of the largest partition to the average size of a partition. Consider a table with four partitions of sizes 20, 20, 35, and 80 pages. The size of the average partition is $(20 + 20 + 35 + 85)/4 = 40$ pages. The biggest partition has 85 pages so partition skew is calculated as $85/40 = 2.125$. In partitioned scans, the cost of doing a parallel scan is as expensive as doing the scan on the largest partition. Instead, a hash-based partition may turn out to be fast, as each worker process may hash on a page number or an allocation unit and scan its portion of the data. The penalty paid in terms of loss of performance by skewed partitions is not always at the scan level, but rather as more complex operators like several join operations are built over the data. The margin of error increases exponentially in such cases.

Run `sp_help` on a table to see the partition skews:

```
sp_help HA2
```

```
.....
name      type partition_type partitions  partition_keys
-----
HA2       base table          hash           2 a1, a2

partition_name partition_id pages      segment
create_date
-----
-----
HA2_752002679          752002679      324 default
Aug 10 2005  2:05PM
HA2_768002736          768002736      343 default
```



```
Aug 10 2005 2:05PM
```

```
Partition_Conditions
```

```
-----  
NULL
```

```
Avg_pages    Max_pages    Min_pages    Ratio (Max/Avg)
```

```
Ratio (Min/Avg)
```

```
-----  
-----  
-----  
333          343          324          1.030030  
0.972973
```

Alternatively, you can calculate skew by querying the systabstats system catalog, where the number of pages in each partition is listed.

Why queries do not run in parallel

Adaptive Server runs a query in serial when:

- There is not enough data to benefit from parallel access.
- The query contains no equijoin predicates like:

```
select * from RA2, RB2
where a1 > b1
```
- There are not enough resources, such as thread or memory, to run a query in parallel.
- Uses a covered scan of a global nonclustered index.
- Tables and indexes are accessed inside a nested subquery that cannot be flattened.

Runtime adjustment

If there are not enough worker processes available at runtime, the execution engine attempts to reduce the number of worker processes used by the exchange operators present in the plan:

- First, by attempting to reduce the worker process usage of certain exchange operators in the query plan without resorting to serial recompilation of the query. Depending on the semantics of the query plan, certain exchange operators are adjustable and some are not. Some are limited in the way they can be adjusted.
- Parallel query plans need a minimum number of worker processes to run. When enough worker processes are not available, the query is recompiled serially. When recompilation is impossible, the query is aborted and the appropriate error message is generated.

It does so in two ways:

Adaptive Server supports serial recompilation for all:

- Ad hoc select queries, except select into, alter table, and execute immediate queries.
- Stored procedures, except select into and alter table queries.

Recognizing and managing runtime adjustments

Adaptive Server provides two mechanisms to help you observe runtime adjustments of query plans:

- `set process_limit_action` allows you to abort batches or procedures when runtime adjustments take place.
- `showplan` prints an adjusted query plan when runtime adjustments occur, and `showplan` is effect.

Using `set process_limit_action`

Use `process_limit_action` with the `set` command to monitor the use of adjusted query plans at a session or stored procedure level. When you set `process_limit_action` to “abort,” Adaptive Server records error 11015 and aborts the query, if an adjusted query plan is required. When you set `process_limit_action` to “warning,” Adaptive Server records error 11014 but still executes the query. For example, this command aborts the batch when a query is adjusted at runtime:

```
set process_limit_action abort
```

By examining the occurrences of errors 11014 and 11015 in the error log, you can determine the degree to which Adaptive Server uses adjusted query plans instead of optimized query plans. To remove the restriction and allow runtime adjustments, use:

```
set process_limit_action quiet
```

See *set* in the *Reference Manual: Commands*.

Using *showplan*

When you use *showplan*, Adaptive Server displays the optimized plan for a given query before it runs the query. When the query plan involves parallel processing, and a runtime adjustment is made, *showplan* displays this message, followed by the adjusted query plan:

```
AN ADJUSTED QUERY PLAN IS BEING USED FOR STATEMENT 1  
BECAUSE NOT ENOUGH WORKER PROCESSES ARE CURRENTLY  
AVAILABLE.
```

```
ADJUSTED QUERY PLAN:
```

When using *set notexec*, Adaptive Server does not attempt to execute a query, so runtime plans are never displayed.

Reducing the likelihood of runtime adjustments

To reduce the number of runtime adjustments, increase the number of worker processes that are available to parallel queries. You can do this either by adding more total worker processes to the system or by restricting or eliminating parallel execution for noncritical queries, using either:

- *set parallel_degree* to set session-level limits on the degree of parallelism, or
- The query-level *parallel 1* and *parallel N* clauses to limit the worker process usage of individual statements.

To reduce the number of runtime adjustments for system procedures, recompile the procedures after changing the degree of parallelism at the server or session level. See *sp_recompile* in *Adaptive Server Reference Manual: Procedures*.

Eager and Lazy Aggregation

This chapter discusses eager and lazy aggregation in Adaptive Server.

Topic	Page
Overview	203
Aggregation and query processing	205
Examples	208
Using eager aggregation	215

Overview

Aggregate processing is one of the most useful operations in DBMS environments. It summarizes large amounts of data with an aggregated value, including:

- The minimum, maximum, sum, or average value of a column in a specified set of rows
- The count of rows that match a condition
- Other statistical functions

In SQL, aggregate processing is performed using the aggregation functions `min()`, `max()`, `count()`, `sum()`, and `avg()`, and `group by` and `having` clauses. The SQL language implements two aggregate processing types, *vector aggregation* and *scalar aggregation*. A select-project-join (SPJ) query illustrates these two types of aggregate processing:

```
select r1, s1
from r, s
where r2 = s2
```

Vector aggregation

In vector aggregation, the SPJ result set is grouped on the `group by` clause expressions, and then the `select` clause aggregation functions are applied to each group. The query produces one result row per group:

```
select r1, sum (s1)
from r, s
```

```
where r2 = s2
group by r1
```

Scalar aggregation

In scalar aggregation, there is no group by clause and the entire SPJ result set is aggregated, as a single group, by the same select clause aggregate functions. The query produces a single result row:

```
select sum (s1)
from r, s
where r2 = s2
```

Eager aggregation

Eager aggregation transforms the internal representation of queries such as those discussed above, and processes them as if the aggregation is performed incrementally: first locally, over each table, producing intermediate aggregate results over smaller local subgroups, and then globally after the join, thus combining the local aggregation results to produce the final result set.

These queries, which return the same result set over any data set, are *derived table* SQL-level rewrites of the vector and scalar aggregation examples above. They illustrate the eager aggregation transformations that Adaptive Server performs on the internal representation of the queries.

Vector aggregation

```
select r1, sum(sum_s1 * count_r)
from
  (select
    r1,r2,
    count_r = count(*)
  from r
  group by r1,r2
  )gr
,
  (select
    s2,
    sum_s1 = sum(s1)
  from s
  group by s2
  )gs
where r2=s2
group by r1
```

Scalar aggregation

```
select sum(sum_s1 * count_r)
from
  (select
    r2,
```

```
        count_r = count(*)
    from r
    group by r2
)gr
,
(select
    s2,
    sum_s1 = sum(s1)
from s
group by s2
)gs
where r2 = s2
```

Eager aggregation plans are generated and costed by the optimizer, and can be chosen as the best plan. This form of advanced query optimization can result in orders of magnitude of performance gain, relative to the original SQL query.

Aggregation and query processing

From the perspective of query processing (QP), aggregation can be both a costly operation, and an operation whose placement has an important impact on query performance.

- Aggregation generally computes an aggregated value. Vector aggregation is costly, compared to scalar aggregation, as rows must be grouped together to obtain the aggregated result over a group; this implies, in general, a reordering of the rows through sorting or hashing, both costly operations.
- Aggregating after applying cardinality-reducing operators (such as filters) to the input set reduces the cost of aggregation and can thus improve overall query performance.
- Aggregating before applying cardinality-increasing operators (such as joins and unions) to the input set reduces the cost of aggregation and can thus improve overall query performance.
- Aggregating early can reduce the costs of parent operators through reducing their input set cardinality, and can thus improve overall query performance.
- Aggregation can dramatically reduce the cardinality of the input set in the result set, when the grouping columns have relatively few distinct value combinations.

- Some properties of the aggregation's input set, as already grouped (for example, when the aggregation is ordered on the grouping columns), reduce the cost of vector aggregation; in scalar aggregation, rows ordered on the aggregated column allow computing a min or max without accessing each input row.
- Plan fragment physical properties have a big impact on aggregation cost.

The naive QP implementation of aggregation places the scalar or vector aggregate operator, as indicated by the SQL query, over the SPJ part of its query block. However, there are algebraic transformations that preserve the semantics of the query and allow aggregation at other places in the operators tree:

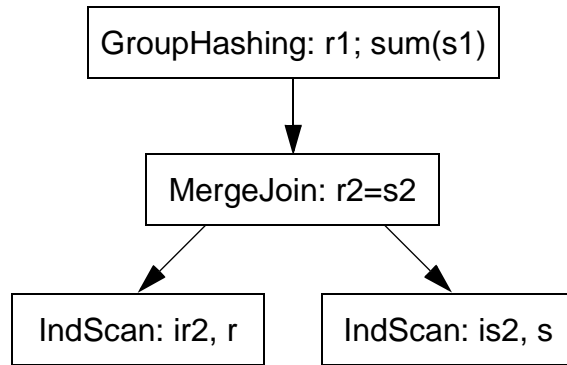
- Pushing the aggregation down toward the leaves, to aggregate early (called eager aggregation).
- Pulling the aggregation up toward the root, to aggregate late (called lazy aggregation).

Plans obtained through such transformations differ greatly in performance. More importantly to distributed query processing (DQP), the cardinality of intermediate results can be greatly reduced by eager aggregation. Such orders-of-magnitude cardinality reduce cross-node data transfer cost, thus removing the main shortcoming of DQP as opposed to traditional QP.

Adaptive Server 15.0.2 and later implements eager aggregation over the leaves of a query plan, which means over the scan operators.

This query illustrates the QP implications of eager aggregation:

```
select r1, sum(s1)
from r,s
where r2 = s2
group by r1
```

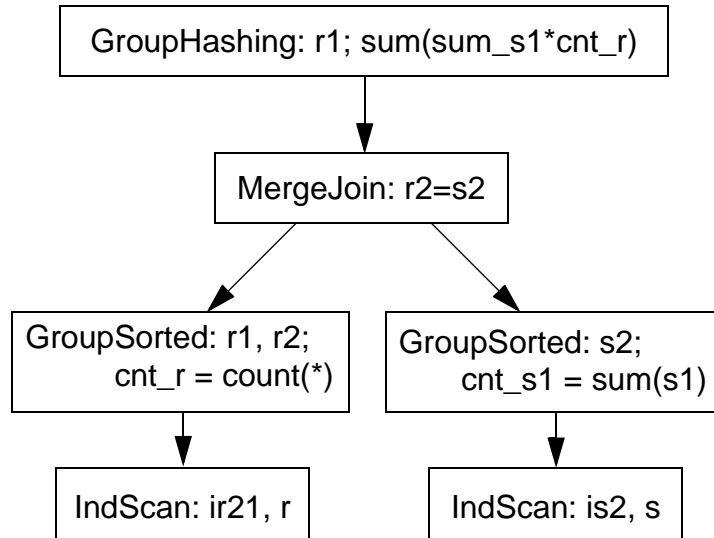

Figure 6-1: Typical query execution plan

The two index scans, on $r(r2)$ and $s(s2)$ provide the orderings needed by the “ $r2=s2$ ” merge join. Hash-based grouping is done over the join, as the query specifies it.

The optimizer also generates query plans that perform eager aggregation, also called the *push-down of grouping*, *early grouping*, or *eager grouping*. The SQL representation of the transform using derived tables is:

```

select r1, sum(sum_s1 * cnt_r)
from
  (select r1, r2, cnt_r = count(*)
   from r
   group by r1, r2
  ) as gr
,
  (select s2, sum_s1 = sum(s1)
   from s
   group by s2
  ) as gs
where r2 = s2
group by r1
  
```

Figure 6-2: Possible eager aggregation plan

The two eager **GroupSorted** operators group on the local grouping columns. **GroupSorted** operators apply to any column projected out for a reason other than that it is an aggregation function argument. These columns include:

- The main grouping columns in the group by clause
- Columns needed by predicates not yet applied

To place the cheap **GroupSorted** operator, the child plan fragment must provide ordering on all the local grouping columns; hence the `ir21` index on `r(r2, r1)`.

Examples

Online data archiving

The most compelling reason to implement eager aggregation is online data archiving, which is a distributed query processing (DQP) installation where recent OLTP read-write data is on an Adaptive Server and historical read-only data is on another server, either Adaptive Server or ASIQ.

The following view, `v`, offers decision support system (DSS) applications transparent access to local Adaptive Server data in `ase_tab` and, through the Component Integration Services (CIS) `proxy_asiq_tab`, to remote historical data on an ASIQ server.

```

create view v(v1, v2)
as
select a1, a2 from ase_tab
union all
select q1, q2 from proxy_aseq_tab

```

The DSS applications ignore the distributed nature of the data and use such union-in-view tables as the base tables of their complex queries, typically using aggregation:

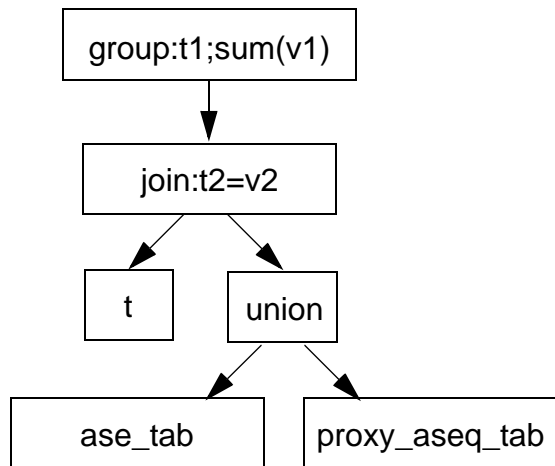
```

select t1, sum(v1)
from t,v
where t2=v2
group by t1

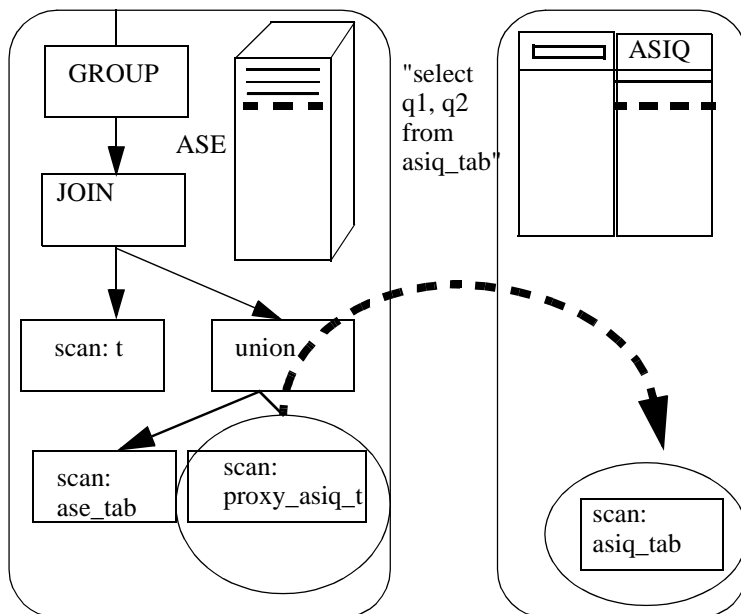
```

After view and union resolution, the following operator tree is obtained:

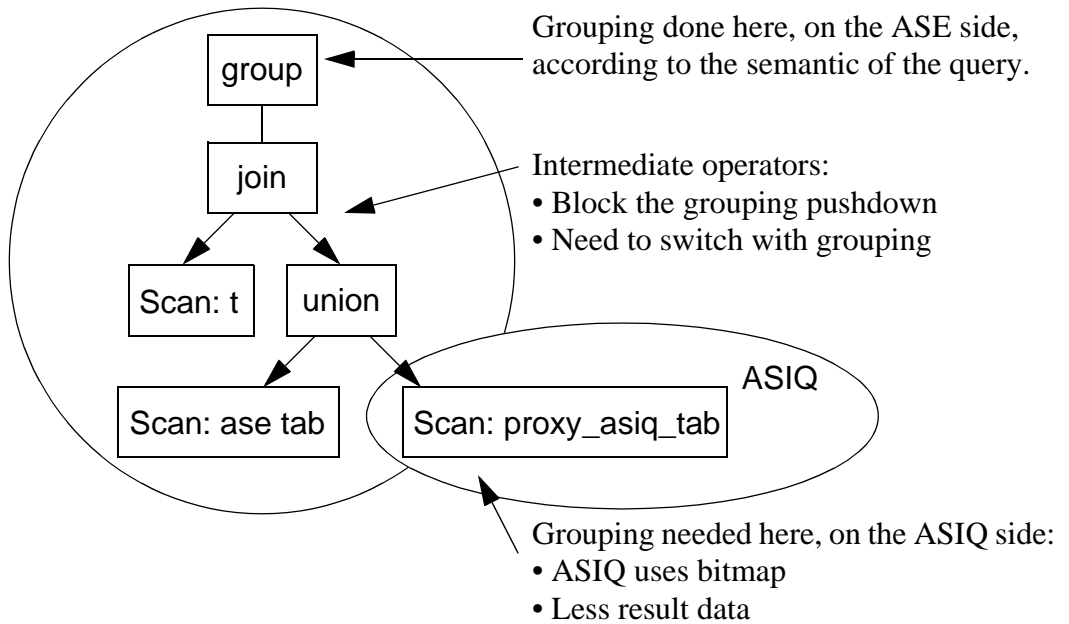
Figure 6-3: SQL query rewrite



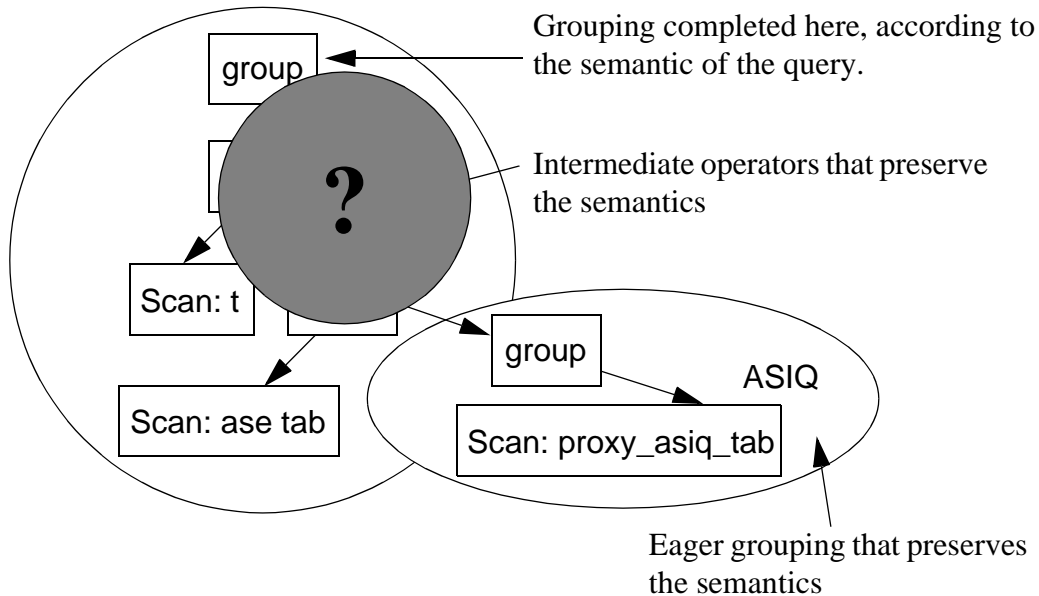
As this tree uses a CIS proxy table, the CIS layer uses a specialized remote scan operator to generate and ship a plan fragment to the remote site.

Figure 6-4: Suboptimal classical CIS behavior

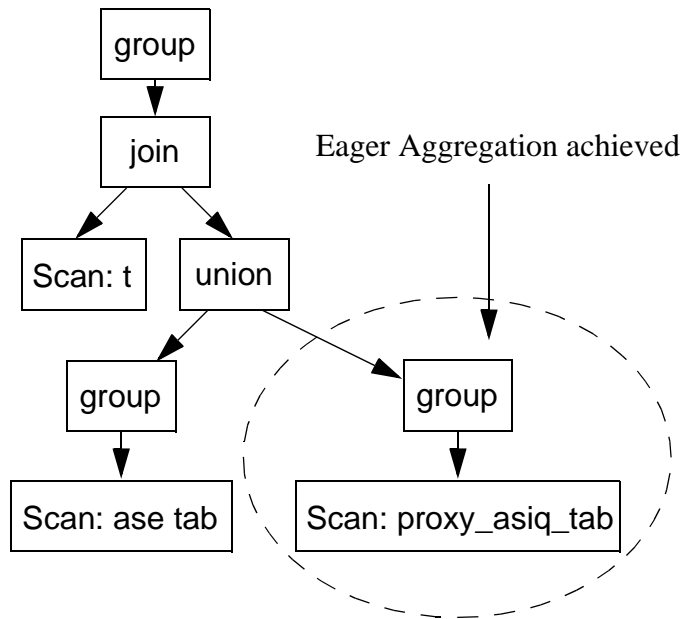
As such, this mechanism is suboptimal: the entire history table is shipped through the CIS layer to the Adaptive Server side, incurring a large network cost; furthermore, the advanced ASIQ bitmap-based grouping algorithms are not used.

Figure 6-5: Classical processing of aggregation

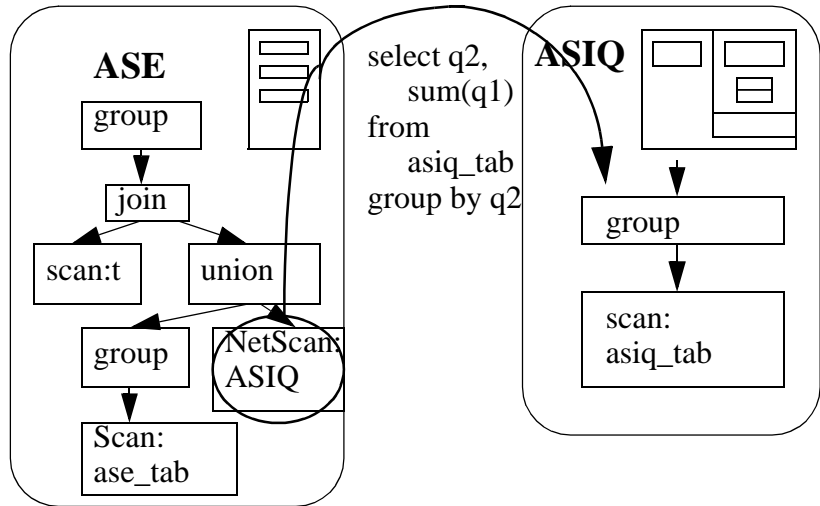
Ideally, transformations are performed on the operators tree and grouping on the ASIQ side, so that only aggregated data is transferred.

Figure 6-6: Desired aggregation processing layout

In this example, there are two operators between the group and the CIS proxy: a join and a union. The next transform pushes grouping below the join and the union, achieving *eager aggregation*:

Figure 6-7: Eager aggregation

Grouping is now adjacent to the CIS proxy. The CIS layer can now send a grouped query to ASIQ and return aggregated data.

Figure 6-8: Optimal CIS behavior with eager aggregation**DSS/DQP**

The efficient execution of complex aggregated DSS queries in a distributed environment is a challenge not only in online data archiving; it is, in general, a generic DSS/DQP problem in online data archiving DSS/DQP:

- A typical query involves complex joins and unions, and aggregation is performed at the top of the query tree.
- The data is distributed across the nodes, and the intermediate results must be shipped from a producer node to a consumer node for further processing.

Single-node DSS

Although the examples above are for DQP in general, and online data archiving in particular, the eager aggregation performance impact goes beyond shipping intermediate results between DQP nodes.

Eager aggregation enhances the performance of aggregated complex queries by reducing intermediate result sets. Since aggregated complex queries are typical, eager aggregation enhances Adaptive Server performance in all DSS applications.

Using eager aggregation

Eager aggregation is an internal query processing feature. You need not change anything at the SQL level when you enable eager aggregation; queries that use aggregation have eager aggregation-based plans automatically enumerated and costed by the optimizer.

Enabling eager aggregation

Eager aggregation is controlled by the `advanced_aggregation` optimizer setting, which is off by default in all optimization goals except `allows_dss`, where it is on. Eager aggregation can be enabled, disabled, or reset to the optimizer goal's default value at either the connection or query level.

For example, to enable at the connection level:

```
set advanced_aggregation on
```

To enable at the query level:

```
select r1, sum (s1)
from r, s
where r2 = s2
group by r1
plan
"(use advanced_aggregation on)"
```

Alternatively, if the optimization goal is set to `allows_dss`, eager aggregation is implicitly enabled. In this example, an abstract plan sets `allows_dss` at the query level:

```
select r1, sum (s1)
from r, s
where r2 = s2
group by r1
plan
"(use optgoal allows_dss)"
```

Checking for eager aggregation

When eager aggregation is enabled, the optimizer determines cost, depending on whether the estimated cheapest plan uses eager aggregation or not.

Output from the showplan aggregation:

```
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

6 operator(s) under root

|ROOT:EMIT Operator

```
|
| |HASH VECTOR AGGREGATE Operator
| |  GROUP BY
| |  Evaluate Grouped SUM OR AVERAGE AGGREGATE.
| |  Using Worktable2 for internal storage.
| |  Key Count: 1
|
| |MERGE JOIN Operator (Join Type: Inner Join)
| |  Using Worktable1 for internal storage.
| |  Key Count: 1
| |  Key Ordering: ASC
|
| |GROUP SORTED Operator
| |  Evaluate Grouped COUNT AGGREGATE.
|
| | |SCAN Operator
| | |  FROM TABLE
| | |  r
| | |  Index : ir21
| | |  Forward Scan.
| | |  Positioning at index start.
| | |  Index contains all needed columns. Base table will not be read.
| | |  Using I/O Size 2 Kbytes for index leaf pages.
| | |  With LRU Buffer Replacement Strategy for index leaf pages.
|
| |GROUP SORTED Operator
| |  Evaluate Grouped SUM OR AVERAGE AGGREGATE.
|
| | |SCAN Operator
| | |  FROM TABLE
| | |  s
| | |  Index : is21
| | |  Forward Scan.
| | |  Positioning at index start.
```

```
| | | | | Index contains all needed columns. Base table will not be read.
| | | | | Using I/O Size 2 Kbytes for index leaf pages.
| | | | | With LRU Buffer Replacement Strategy for index leaf pages.
```

```
r1
```

```
-----
          1          2
          2          4
(2 rows affected)
```

As the query performs vector aggregation over the join of *r* and *s*, the hash vector aggregate operator at the top of the query tree is expected in all cases. However, the group sorted operators over the scans of *r* and of *s* are not part of the query; they perform the eager aggregation.

When `advanced_aggregation` is off, the plan does not contain the eager aggregation operators group sorted:

```
1> set advanced_aggregation off
2> go
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> go
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

4 operator(s) under root

```
|ROOT:EMIT Operator
|
| |HASH VECTOR AGGREGATE Operator
| |  GROUP BY
| |  Evaluate Grouped SUM OR AVERAGE AGGREGATE.
| |  Using Worktable2 for internal storage.
| |  Key Count: 1
| |
| |MERGE JOIN Operator (Join Type: Inner Join)
| |  Using Worktable1 for internal storage.
| |  Key Count: 1
| |  Key Ordering: ASC
| |
| | |SCAN Operator
| | |  FROM TABLE
| | |  r
| | |  Index : ir21
| | |  Forward Scan.
```

```
| | | | Positioning at index start.
| | | | Index contains all needed columns. Base table will not be read.
| | | | Using I/O Size 2 Kbytes for index leaf pages.
| | | | With LRU Buffer Replacement Strategy for index leaf pages.
| | | |
| | | | SCAN Operator
| | | | FROM TABLE
| | | | s
| | | | Index : is21
| | | | Forward Scan.
| | | | Positioning at index start.
| | | | Index contains all needed columns. Base table will not be read.
| | | | Using I/O Size 2 Kbytes for index leaf pages.
| | | | With LRU Buffer Replacement Strategy for index leaf pages.
r1
-----
          1          2
          2          4
(2 rows affected)
```

Forcing eager aggregation with abstract plans

The optimizer opportunistically enumerates the cheap GroupSorted-based eager aggregation plans when the child plan fragment provides an ordering on the local grouping columns.

This limitation avoids increasing the optimization search space and time. However, in some cases hash-based eager aggregation produces the cheapest plan. abstract plans can be used in such cases to force eager aggregation. advanced_grouping must be enabled to use such an abstract plan; otherwise the eager aggregation abstract plan is rejected.

In the example above, if r has no index on (r1, r2), and if r is large but has few r1--r2 distinct pairs of values, a hash join with eager grouping over r is the best plan, forced by this abstract plan:

```
1> select r1, sum(s1)
2> from r, s
3> where r2=s2
4> group by r1
5> plan
6> "(group_hashing
7>      (h_join
8>          (group_hashing
9>              (t_scan r)
```

```

10>          )
11>          (t_scan s)
12>      )
13> )"
14> go

```

QUERY PLAN FOR STATEMENT 1 (at line 1).

Optimized using the Abstract Plan in the PLAN clause.

STEP 1

The type of query is SELECT.

5 operator(s) under root

```

|ROOT:EMIT Operator
|
| |HASH VECTOR AGGREGATE Operator
| | GROUP BY
| | Evaluate Grouped SUM OR AVERAGE AGGREGATE.
| | Using Worktable3 for internal storage.
| | Key Count: 1| |
| | |HASH JOIN Operator (Join Type: Inner Join)
| | | Using Worktable2 for internal storage.
| | | Key Count: 1
| | |
| | |HASH VECTOR AGGREGATE Operator
| | | GROUP BY
| | | Evaluate Grouped COUNT AGGREGATE.
| | | Using Worktable1 for internal storage.
| | | Key Count: 2
| | |
| | |SCAN Operator
| | | FROM TABLE
| | | r
| | | Table Scan.
| | | Forward Scan.
| | | Positioning at start of table.
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.
| | |
| | |SCAN Operator
| | | FROM TABLE
| | | s
| | | Table Scan.
| | | Forward Scan.
| | | Positioning at start of table.
| | | Using I/O Size 2 Kbytes for data pages.
| | | With LRU Buffer Replacement Strategy for data pages.

```

rl

```
-----  
1           2  
2           4
```

(2 rows affected)

The hash vector aggregate operator eagerly aggregates the scan of *r*, as requested by the abstract plan.

Controlling Optimization

This chapter describes query processing options that affect the query processor's choice of join order, index, I/O size, and cache strategy.

Topic	Page
Special optimizing techniques	221
Specifying query processor choices	231
Specifying table order in joins	232
Specifying the number of tables considered by the query processor	233
Specifying query index	234
Specifying I/O size in a query	236
Specifying cache strategy	239
Controlling large I/O and cache strategies	241
Asynchronous log service	241
Enabling and disabling merge joins	244
Enabling and disabling join transitive closure	245
Controlling literal parameterization	246
Suggesting a degree of parallelism for a query	248
Concurrency optimization for small tables	258

Special optimizing techniques

Sybase recommends that, before using the tools discussed in this chapter, you read the *Performance and Tuning Series: Basics*. It will help you understand the material in this chapter.

Use the optimization techniques with caution, as they allow you to override the decisions made by the Adaptive Server query processor and if misused, can have an extremely negative effect on performance. You should understand the impact on the performance of both your individual query and the possible implications for overall system performance.

In most situations, Adaptive Server advanced, cost-based query processor produces excellent query plans. However, there are times when the query processor does not choose the proper index for optimal performance, or chooses a suboptimal join order, and you must control the access methods for the query. The optimization techniques allow you to take that control.

In addition, while you are tuning, you may want to see the effects of a different join order, I/O size, or cache strategy. Some of the optimization options let you specify query processing or access strategy without costly reconfiguration.

Adaptive Server provides tools and query clauses that affect query optimization and advanced query analysis tools that let you understand why the query processor makes the choices that it does.

Note This chapter suggests workarounds for certain optimization problems. If the workarounds do not adequately address these problems, call Sybase Technical Support.

Viewing current optimizer settings

`sp_options` allows you to view the current optimizer settings for these options:

- `set plan dump / load`
- `set plan exists check`
- `set forceplan`
- `set plan optgoal`
- `set [optCriteria]`
- `set plan opttimeoutlimit`
- `set plan replace`
- `set statistics simulate`
- `set metrics_capture`
- `set prefetch`
- `set parallel_degree number`
- `set process_limit_action`

- set resource_granularity number
- set scan_parallel_degree number
- set repartition_degree number

sp_options queries the sysoptions fake table, which stores information about each set option, its category, and its current and default settings. sysoptions also contains a bitmap that provides detailed information for each option

The syntax for sp_options is:

```
sp_options [ [show | help
              [, option_name | category_name | null
              [, dflt | non_dflt | null
              [, spid] ] ] ] ]
```

where:

- show – lists the current and default values of all options, grouped according to their category. Issuing sp_options show with an option name specified shows you the current and default value for the individual option. You can also specify a session ID, and whether you want to view options with default settings or options with nondefault settings.
- help – show usage information. Achieve the same result by issuing sp_options with no parameters.
- null – indicates the option for which you want to view the settings.
- dflt | non_dflt | null – indicates whether to show options with default settings or to show options with non-default settings.
- spid – specifies the session ID. Use the session ID to view other session settings.

For example, to display the current optimizer settings shown below, enter:

```
sp_options show
Category: Query Tuning
name
currentsetting      defaultsetting      scope
-----
optlevel
  ase_default        ase_current         3
optgoal
  allrows_mix        allrows_mix         3
opttimeoutlimit
  10                  10                  2
repartition_degree
```

```

          1          1          2
scan_parallel_degree
          0          1          2
resource_granularity
          10         10          2

. . .

outer_join_costing: outer join row counts and histogramming
          0          0          7
join_duplicate_estimates: avoid overestimates of dups in joins
          0          0          7
imdb_costing: 0 PIO costing for scans for in-memory database
          1          1          7
auto_temptable_stats: auto generation of statistics for #temptables
          0          0          7
use_mixed_dt_sarg_under_specialor: allow special OR in case of mixed . . .
          0          0          7
timeout_cart_product: timeout queries involving cartesian product and more . . .
          0          0          7

(81 rows affected)

```

See *Adaptive Server Reference Manual: Procedures*.

Any user can query sysoptions:

You can also use string manipulation or a cast. For example, if an option is numeric, you can query sysoptions by entering:

```

if (isnumeric(currentsetting))
    select @int_val = convert(int, currentsetting)
...
else
    select @char_val = currentsetting
...

```

For more information about sysoptions, see *Adaptive Server Reference Manual: Tables*.

Setting the optimization level

By default, Adaptive Server does not enable some performance related optimizer settings, and you must enable them using the `set` command. Because these optimizer settings are not enabled, any applications already running efficiently are not effected by, nor benefit from, optimizer changes when upgrading to the latest version of Adaptive Server.

Adaptive Server allows you to set the optimization level globally (that is, the optimization levels are set across the server) and at the session level. Enabling these changes increases query performance for many applications, though Sybase recommends additional performance testing.

Use the `@@optlevel` global variable to determine the current optimization level settings:

```
select @@optlevel
```

The optimization settings are organized according to which optimization changes are available for each Adaptive Server release. Table 7-1 describes the settings:

Table 7-1: Optimization level

Parameter	Description
<code>ase_current</code>	Enables all optimizer changes through the current release
<code>ase_default</code>	Disables all optimizer changes since version 1503 ESD #1
<code>ase1503esd2</code>	Enables all optimizer changes through version 15.0.3 ESD #2
<code>ase1503esd3</code>	Enables all optimizer changes through version 15.0.3 ESD #3

These optimization level criteria are enabled by default:

Table 7-2: Optimization criteria enabled by default

Setting	Description
cr421607	Support NULL=NULL merge and hash join keys
cr467566	Allow abstract plans and statement caches to work together
cr487450	Improves distinct costing for multitable outer joins and semijoins
cr497066	Infer the nullability of isnull by observing its parameters
cr500736	Support nocase sort order columns in merge join and hash join keys
cr531199	Increase the number of useful nested loop join plans the optimizer considers
cr534175	Compute group by worktables in nested subqueries only once, when possible
cr544485	Mark subquery join predicates with distinct view as SARGs
cr545059	Reduce buffer manager optimization sort usage
cr545180	Avoid reformatting with no SARGs if a useful index exists
cr545379	Disallow reformatting on user-forced index scan
cr545585	Covered iscan CPU costing too expensive
cr545653	Avoid inner table buffer estimate starvation
cr545771	Improve multi-table outer join and semijoin costing
cr546125	Allow a nonunique index scan for implicitly updatable cursors
cr552795	Eliminate unnecessary duplicate rows during reformatting
cr562947	Allow cursor table scans
data_page_prefetch_costing	Adds clustered row bias
mru_buffer_costing	Wash size buffer limit for MRU

These optimization level criteria are enabled when you enable ase1503esd2:

Table 7-3: Optimization criteria enabled with ase1503esd2

Setting	Description
cr556728	Facilitate merge joins between small tables
cr559034	Avoid preferring non-covering over covered index scans
allow_wide_top_sort	Allow top sorts to exceed max row size
avoid_bmo_sorts	Avoid sorts used only for buffer manager optimization
conserve_tempdb_space	Keep estimated temporary databases below resource granularity
distinct_exists_transform	Transform distinct to semi-join
join_duplicate_estimates	Avoid overestimates of join duplicates
outer_join_costing	Outer join row counts and histogramming
search_engine_timeout_factor	Open cursor command takes a long time with a complex select statement
timeout_cart_product	Timeout queries involving cartesian product and more than 5 tables.

Table 7-4 lists optimization criteria that are enabled when you enable ase1503esd3 or ase_current

Table 7-4: Optimization criteria enabled with ase1503esd3 and ase_current

Setting	Description
auto_template_stats	Automatically generate statistics for temporary tables
use_mixed_dt_sarg_under_specialor	Allow special or for mixed datatype SARGs in an in or list

These optimization criteria are off by default:

Table 7-5: Optimization criteria that are disabled by default

Setting	Description
full_index_filter	Eliminate noncovered full index scan strategies
no_stats_distinctness	Allow duplicate estimates without statistics

Set the optimization and criteria levels using one of these methods:

- At the session level – use set plan optlevel to set the optimization level for the current session. This configures the session to use the all optimization changes up through the current version of Adaptive Server:

```
set plan optlevel ase_current
```

- For individual logins – use `sp_modifylogin` to set the optimization level for logins. `sp_modifylogin` calls a user-created stored procedure that defines the optimization level. For example, if you create this stored procedure to enable the `ase1503esd2` optimization level, but disable the optimization level for `cr545180`:

```
create proc login_proc
as
set plan optlevel ase1503esd2
set cr545180 off
go
```

You may apply these optimization settings to any login. This applies the settings from `login_proc` to user `joe`:

```
sp_modifylogin joe, 'login script', login_proc
```

- Across the server – use the `sp_configure` “optimizer level” parameter to set the optimization level globally. This sets enables all optimizer changes up to the current version of Adaptive Server:

```
sp_configure 'optimizer level', 0, 'ase_current'
```

- Within an abstract plan – use the `optlevel` abstract plan to set the optimizer level. This example enables all optimizer changes up to the current version of Adaptive Server for the current plan:

```
select name from sysdatabases
plan '(use optlevel ase_current)'
```

- During a session with the `set` command – use the `set` command to change the optimizer criteria level for the current session. Optimizer criteria changes may improve the performance for some queries while deteriorating others. You may find better performance by applying the optimizer criteria level at a fine grain level (perhaps at the query level). Adaptive Server denotes some optimizer criteria levels by their CR numbers, while other more recent optimizer changes are specified with descriptive names. Use `sp_options` to view the list of available options in the current release. This enables the optimization criteria for CR 545180:

```
set CR545180 on
```

Note When you enable an optimization criteria, Adaptive Server retains the previous optimization level.

Optimizer Diagnostic Utility

The `sp_opt_querystats` system procedure lets you analyze the query plan generated by the Adaptive Server optimizer and the factors that influenced its choice of a query plan. This analysis may help determine if elements in the query or the execution environment affect how Adaptive Server executes the query and its performance. You need not run the selected query to perform the analysis.

`sp_opt_querystats` output includes:

- The query plan generated by `showplan`
- Enabled trace flags and switches
- I/O activity for the query generated by `set statistics io`
- Missing statistics found for any of the tables involved in the query
- The estimated plan cost calculated by the optimizer
- The final plan and cost estimations calculated by the optimizer
- The abstract plan for the query
- The result of the query if the result set is executed (for example, if `noexec` is not on)
- The logical operator tree for the query generated by `set option show`
- Query execution time generated by `set statistics time`
- After you execute the query, the query execution time generated by `set statistics time`

Configuring Adaptive Server to run sp_opt_querystats

- 1 Install the Job Scheduler. See the *Job Scheduler Users Guide*.

Note This may require that you restart Adaptive Server.

- 2 If Adaptive Server is unnamed, set the server name then restart Adaptive Server:

```
sp_addserver server_name, local
```

- 3 Any login that runs sp_opt_querystats must have the js_user_role role, and must use a non-Null password to log in to Adaptive Server.

sp_opt_querystats require the sa_role:

```
grant role role_name to login_name
```

- 4 Create an external login on the loopback server for all users who run sp_opt_querystats:

```
sp_addexternlogin loopback, <login>, <password>,
<password>
```

- 5 Sybase recommends that you set the value for maximum job output to 1000000 to ensure that Job Scheduler can capture all diagnostic output for your queries.

Note If sp_opt_querystats truncates the output, increase the value for maximum job output. Increasing this value consumes no additional resources.

Running sp_opt_querystats

Log in to Adaptive Server using the login that has the js_user_role. Run sp_opt_querystats to analyze a query (the diagnostic information begins with the phrase [BEGIN QUERY ANALYSIS] and ends with [END QUERY ANALYSIS]).

The syntax is:

```
sp_opt_querystats "query_text" | help [, "diagnostic_options" | null
[, database_name]]
```

For example:

```
sp_opt_querystats "select * from pubs2..authors where
```



```
au_id = "172-32-1176"
```

See the *Reference Manual: Procedures*.

Specifying query processor choices

Adaptive Server lets you specify these optimization choices by including commands in a query batch or in the text of the query:

- The order of tables in a join
- The number of tables evaluated at one time during join optimization
- The index used for a table access
- The I/O size
- The cache strategy
- The degree of parallelism

Under some circumstances, the query processor does not choose the best plan. Occasionally, the plan chosen by the query processor is only slightly more expensive than the “best” plan, so you must weigh the cost of maintaining forced options against the slower performance of a less than optimal plan.

The commands to specify join order, index, I/O size, or cache strategy, coupled with the query-reporting commands like `statistics io` and `showplan`, can help you determine why the query processor makes its choices.

Warning! Use the options described in this chapter with caution. Forced query plans may be inappropriate in some situations and may cause poor performance. If you include these options in your applications, regularly check query plans, I/O statistics, and other performance data.

These options are generally intended for use as tools for tuning and experimentation, not as long-term solutions to optimization problems.

Specifying table order in *joins*

Adaptive Server optimizes join orders to minimize I/O. In most cases, the order that the query processor chooses does not match the order of the from clauses in your select command. To force Adaptive Server to access tables in the order they are listed, use:

```
set forceplan [on|off]
```

The query processor still chooses the best access method for each table. If you use forceplan and specify a join order, the query processor may use different indexes on tables than it would with a different table order, or it may not be able to use existing indexes.

You might use this command as a debugging aid if other query analysis tools lead you to suspect that the query processor is not choosing the best join order. Always verify that the order you are forcing reduces I/O and logical reads by using `set statistics io on` and comparing I/O both with and without forceplan.

If you use forceplan, your routine performance maintenance checks should include verifying that the queries and procedures that use forceplan still require the option to improve performance.

You can include forceplan in the text of stored procedures.

`set forceplan` forces only join order, and not join type. There is no command for specifying the join type; you can disable merge joins at the server or session level.

You can disable hash joins at the session level. Also remember that an abstract plan allows full plan specification, including join order and join types.

See Chapter 12, “Creating and Using Abstract Plans,” and “Enabling and disabling merge joins” on page 244.

Forcing join order has these risks:

- Misuse can lead to extremely expensive queries. Always test the query thoroughly with `statistics io`, and with and without forceplan.
- It requires maintenance. You must regularly check queries and stored procedures that include forceplan. Also, each new version of Adaptive Server may eliminate the problems that lead you to incorporate index forcing, so check all queries that use forced query plans each time a new version is installed.

Before you use forceplan:

- Check the showplan output to determine whether index keys are used as expected.
- Use set option show normal to look for other optimization problems.
- Run update statistics on the index.
- Use update statistics to add statistics for search arguments on unindexed search clauses in the query, especially for search arguments that match minor keys in compound indexes.
- Use set option show_missing_stats on to look for columns that may need statistics.
- If the query joins more than four tables, use set table count, which may result in an improved join order.

See “Specifying the number of tables considered by the query processor” on page 233.

Specifying the number of tables considered by the query processor

In versions earlier than 15.0, Adaptive Server optimized joins by considering two to four permutations at a time. Versions 15.0 and later do not limit the query processor to two or four permutations. Instead, the new search engine introduces a timeout mechanism to avoid excessive time spent optimizing a query. The set table count setting discussed later in this section still affects the initial join order looked at by the search engine, and thus affects the final join order when timeout does occur. If you suspect that an inefficient join order is being chosen when the search engine times out, use set table count to increase the number of tables that are considered, which affects the initial join order considered by the search engine in starting the permutation.

Adaptive Server still optimizes joins by considering permutations of two to four tables at a time, but if you suspect that an inefficient join order is being chosen for a join query, use set table count to increase the number of tables that are considered at the same time:

```
set table count int_value
```

Valid values are 0 though 8; 0 restores the default behavior.

For example, to specify four-at-a-time optimization, use:

```
set table count 4
```

As you decrease the value, you reduce the chance that the query processor considers all possible join orders. Increasing the number of tables considered at one time during join ordering can greatly increase the time it takes to optimize a query.

Since the time it takes to optimize the query is increased with each additional table, set table count is most useful when the execution savings from improved join order outweighs the extra optimizing time. Some examples are:

- If you think that a more optimal join order can shorten total query optimization and execution time, especially for stored procedures that you expect to be executed many times once a plan is in the procedure cache
- When saving abstract plans for later use

Use statistics time to check parse and compile time, and statistics io to verify that the improved join order is reducing physical and logical I/O.

If increasing the table count produces an improvement in join optimization, but unacceptably increases CPU time, rewrite the from clause in the query, specifying the tables in the join order indicated by showplan output, and use forceplan to run the query. Be sure that your routine performance maintenance checks include verifying that the join order you are forcing still improves performance.

Specifying query index

You can use the (index *index_name* clause in select, update, and delete statements to specify the index to use for a query. You can also force a query to perform a table scan by specifying the table name. The syntax is:

```
select select_list
  from table_name [correlation_name]
    (index {index_name | table_name } )
    [, table_name ...]
  where ...
```

```
delete table_name
  from table_name [correlation_name]
    (index {index_name | table_name } ) ...
```

```
update table_name set col_name = value
  from table_name [correlation_name]
  (index (index_name | table_name))...
```

For example:

```
select pub_name, title
  from publishers p, titles t (index date_type)
 where p.pub_id = t.pub_id
    and type = "business"
    and pubdate > "1/1/93"
```

Specifying an index in a query may be helpful when you suspect that the query processor is choosing a suboptimal query plan. When you do specify the index:

- Always check statistics for the query to see whether the index you choose requires less I/O than the query processor's choice.
- Test a full range of valid values for the query clauses, especially if you are:
 - Tuning queries on tables that have skewed data distribution
 - Performing range queries, since the access methods for these queries are sensitive to the size of the range

Use (index *index_name* only after testing when you are certain that the query performs better with the specified index option. Once you include an index specification in a query, regularly verify that the resulting plan is still better than other choices made by the query processor.

If a unclustered index has the same name as the table, specifying a table name causes the unclustered index to be used. You can force a table scan using *select_list* from *tablename* (0).

Specifying indexes has these risks:

- Changes in the distribution of data could make the forced index less efficient than other choices.
- Dropping the index means that all queries and procedures that specify the index print an informational message indicating that the index does not exist. The query is optimized using the best alternative access method.
- Increased maintenance, since you must periodically check all queries using this option. Also, each new version of Adaptive Server may eliminate the problems that lead you to incorporate index forcing, so you should check all queries using forced indexes each time you install a new version.

- The index must exist at the time the query using it is optimized. You cannot create an index and then use it in a query in the same batch.

Before specifying an index in queries:

- Check showplan output for the “Keys are” message to be sure that the index keys are being used as expected.
- Use dbcc traceon(3604) or set option show normal to look for other optimization problems.
- Run update statistics on the index.
- If the index is a composite index, run update statistics on the minor keys in the index, if they are used as search arguments. This can greatly improve query processor cost estimates. Creating statistics for other columns frequently used for search clauses can also improve estimates.
- Use set option show_missing_stats on to look for columns that may need statistics.

Specifying I/O size in a query

If your Adaptive Server is configured for large I/Os in the default data cache or in named data caches, the query processor may decide to use large I/O for:

- Queries that scan entire tables
- Range queries using clustered indexes, such as queries using $>$, $<$, $> x$ and $< y$, between, and like “*charstring %*”
- Queries that scan a large number of index leaf pages

If the cache used by the table or index is configured for 16K I/O, a single I/O can read up to 8 pages simultaneously. Each named data cache can have several pools, each with a different I/O size. Specifying the I/O size in a query causes the I/O for that query to take place in the pool that is configured for that size. See the *System Administration Guide: Volume 2* for information on configuring named data caches.

To specify an I/O size that is different from the one chosen by the query processor, add the prefetch specification to the index clause of a select, delete, or update statement. The syntax is:

```
select select_list
  from table_name
```

```
( [index {index_name | table_name} ]
  prefetch size)
[, table_name ...]
where ...
```

```
delete table_name from table_name
  ( [index {index_name | table_name} ]
    prefetch size)
...
```

```
update table_name set col_name = value
  from table_name
    ( [index {index_name | table_name} ]
      prefetch size)
...
```

The valid prefetch size depends on the page size. If no pool of the specified size exists in the data cache used by the object, the query processor chooses the best available size.

If there is a clustered index on *au_lname*, this query performs 16K I/O while it scans the data pages:

```
select *
  from authors (index au_names prefetch 16)
    where au_lname like "Sm%"
```

If a query normally performs large I/O, and you want to check its I/O performance with 2K I/O, you can specify a size of 2K:

```
select type, avg(price)
  from titles (index type_price prefetch 2)
    group by type
```

Note Reference to large I/Os are on a 2K logical page size server. If you have an 8K page size server, the basic unit for the I/O is 8K. If you have a 16K page size server, the basic unit for the I/O is 16K.

Index type and large I/O size

When you specify an I/O size with prefetch, the specification can affect both the data pages and the leaf-level index pages. Table 7-6 shows the effects.

Table 7-6: Access methods and prefetching

Access method	Large I/O performed on
Table scan	Data pages
Clustered index	Data pages only, for allpages-locked tables Data pages and leaf-level index pages for data-only-locked tables
Nonclustered index	Data pages and leaf pages of nonclustered index

showplan reports the I/O size used for both data and leaf-level pages.

See “I/O size messages” on page 56.

When *prefetch* specification cannot be followed

In most cases, when you specify an I/O size in a query, the query processor incorporates the I/O size into the query’s plan. However, there are times when the specification cannot be followed, either for the query as a whole or for a single, large I/O request.

You cannot use large I/O for the query if:

- The cache is not configured for I/O of the specified size. The query processor substitutes the best size available.
- `sp_cachestrategy` has been used to disable large I/O for the table or index.

You cannot use large I/O for a single buffer if:

- Any of the pages included in that I/O request are in another pool in the cache.
- The page is on the first extent in an allocation unit. This extent holds the allocation page for the allocation unit, and only seven data pages.
- No buffers are available in the pool for the requested I/O size.

When a large I/O cannot be performed, Adaptive Server performs 2K I/O on the specific page or pages in the extent that are needed by the query.

To determine whether the *prefetch* specification is followed, use `showplan` to display the query plan and statistics `io` to see the results on I/O for the query. `sp_sysmon` reports on the large I/Os requested and denied for each cache.

See the *Performance and Tuning Series: Monitoring Adaptive Server with `sp_sysmon`*.

setting *prefetch*

By default, a query uses large I/O whenever a large I/O pool is configured and the query processor determines that large I/O would reduce the query cost. To disable large I/O during a session, use:

```
set prefetch off
```

To reenable large I/O, use:

```
set prefetch on
```

If large I/O is turned off for an object using `sp_cachestrategy`, `set prefetch on` does not override that setting.

If large I/O is turned off for a session using `set prefetch off`, you cannot override the setting by specifying a `prefetch size` as part of a `select`, `delete`, or `insert` statement.

The `set prefetch` command takes effect in the same batch in which it is run, so you can include it in a stored procedure to affect the execution of the queries in the procedure.

Specifying cache strategy

For queries that scan a table's data pages or the leaf level of an unclustered index (covered queries), the Adaptive Server query processor chooses one of two cache replacement strategies: the fetch-and-discard, most-recently used (MRU) strategy or the least recently used (LRU) strategy.

See the *Performance and Tuning Series: Physical Database Tuning*.

The query processor may choose the MRU strategy for:

- Any query that performs table scans
- A range query that uses a clustered index
- A covered query that scans the leaf level of a nonclustered index
- An inner table in a nested-loop join, if the inner table is larger than the cache
- The outer table of a nested-loop join, since it needs to be read only once
- Both tables in a merge join.

To affect the cache strategy for objects:

- Specify lru or mru in a select, update, or delete statement
- Use sp_cachestrategy to disable or reenabte the mru strategy

If you specify the MRU strategy, and a page is already in the data cache, the page is placed at the MRU end of the cache, rather than at the wash marker.

Specifying the cache strategy affects only data pages and the leaf pages of indexes. Root and intermediate pages always use the LRU strategy.

In *select*, *delete*, and *update* statements

You can use lru or mru in a select, delete, or update command to specify the I/O size for the query. (You get only sizes based on caches you have configured correctly. For example, if you specify 4K but Adaptive Server does not use a 4K page size, the command returns 2K):

```
select select_list
  from table_name
      (index index_name prefetch size [lru|mru])
  [, table_name ...]
where ...
```

```
delete table_name from table_name (index index_name
  prefetch size [lru|mru]) ...
```

```
update table_name set col_name = value
  from table_name (index index_name
  prefetch size [lru|mru]) ...
```

For example, to add the LRU replacement strategy to a 16K I/O specification, enter:

```
select au_lname, au_fname, phone
  from authors (index au_names prefetch 16 lru)
```

See “Specifying I/O size in a query” on page 236.

Controlling large I/O and cache strategies

Status bits in the sysindexes table identify whether you should consider a table or an index for large I/O prefetch or for MRU replacement strategy. By default, both are enabled. To disable or reenable these strategies, use `sp_cachestrategy`:

```
sp_cachestrategy dbname , [ownername.]tablename
[, indexname | "text only" | "table only"
[, { prefetch | mru }, { "on" | "off"}]]
```

For example, to turn off the large I/O prefetch strategy for the `au_name_index` of the authors table, enter:

```
sp_cachestrategy pubtune, authors, au_name_index,
prefetch, "off"
```

To reenable MRU replacement strategy for the titles table:

```
sp_cachestrategy pubtune, titles, "table only",
mru, "on"
```

Only a system administrator or the object owner can change or view the cache strategy status of an object.

Getting information on cache strategies

To see the cache strategy that is in effect for a given object, execute `sp_cachestrategy`, with the database and object name:

```
sp_cachestrategy pubtune, titles
object name      index name      large IO MRU
-----
titles           NULL              ON          ON
```

`showplan` output shows the cache strategy used for each object, including worktables.

Asynchronous log service

Asynchronous log service (ALS) increases scalability in Adaptive Server and provides higher throughput in logging subsystems for high-end symmetric multiprocessor systems.

You cannot use ALS if you have fewer than four engines; if you attempt to do so, with fewer than 4 online engines an error message appears.

You can enable, disable, or configure ALS using the `sp_dboption` stored procedure:

```
sp_dboption <db Name>, "async log service",
"true|false"
```

After issuing `sp_dboption`, you must issue a checkpoint in the database for which you are setting the ALS option:

```
sp_dboption "mydb", "async log service", "true"
use mydb
checkpoint
```

You can use the checkpoint to identify the one or more databases or use an all clause:

```
checkpoint [all | [dbname[, dbname[, dbname.....]]]
```

To disable ALS, enter:

```
sp_dboption "mydb", "async log service", "false"
use mydb
checkpoint
-----
```

Before you disable ALS, make sure there are no active users in the database. If there are active users in the database when you disable ALS, you see this error message:

```
Error 3647: Cannot put database in single-user mode.
Wait until all users have logged out of the database and
issue a CHECKPOINT to disable "async log service".
```

Use `sp_helpdb` to see whether ALS is enabled in a specified database:

```
sp_helpdb "mydb"
-----
mydb          3.0 MB sa          2
              July 09, 2002
              select into/bulkcopy/pllsort, trunc log on chkpt,
              async log service
```

For more information on these stored procedures, see the *Adaptive Server Reference Manual: Procedures*.

Understanding the user log cache (ULC) architecture

The Adaptive Server logging architecture features the user log cache, or ULC, by which each task owns its own log cache. No other task can write to this cache, and the task continues writing to the user log cache whenever a transaction generates a log record. When the transaction commits or aborts, or when the log cache fills up, the ULC is flushed to the common log cache, shared by all the current tasks, which is then written to the disk.

Flushing the ULC is the first part of a commit or abort operation, requiring the following steps, each of which can cause delay or increase contention:

- 1 Obtaining a lock on the last log page.
- 2 Allocating new log pages if necessary.
- 3 Copying the log records from the ULC to the log cache.

The processes in steps 2 and 3 require you to hold a lock on the last log page, which prevents any other tasks from writing to the log cache or performing commit or abort operations.

- 4 Flush the log cache to disk.

Step 4 requires repeated scanning of the log cache to issue write commands on dirty buffers.

Repeated scanning can cause contention on the buffer cache spinlock to which the log is bound. Under a large transaction load, contention on this spinlock can be significant.

When to use ALS

You can enable ALS on any specified database that has at least one of the following performance issues, if your system runs four or more online engines:

- Heavy contention on the last log page

You can tell that the last log page is under contention when the `sp_sysmon` output in the Task Management Report section shows a significantly high value. For example:

Task Management	per sec	per xact	count	% of total
-----	-----	-----	-----	-----
Log Semaphore Contention	58.0	0.3	34801	73.1%

- Underutilized bandwidth in the log device

Note Use ALS only when you identify a single database with high transaction requirements, since setting ALS for multiple databases may cause unexpected variations in throughput and response times. To configure ALS on multiple databases, first check that throughput and response times are satisfactory.

Using the ALS

Two threads scan the dirty buffers (buffers full of data not yet written to the disk), copy the data, and write it to the log. These threads are:

- The user log cache (ULC) flusher – The ULC flusher is a system task thread that is dedicated to flushing the user log cache of a task into the general log cache. When a task is ready to commit, the user enters a commit request into the flusher queue. Each entry has a handle, by which the ULC flusher can access the ULC of the task that queued the request. The ULC flusher task continuously monitors the flusher queue, removing requests from the queue and servicing them by flushing ULC pages into the log cache.
- The log writer – Once the ULC flusher has finished flushing the ULC pages into the log cache, it queues the task request into a wakeup queue. The log writer patrols the dirty buffer chain in the log cache, issuing a write command if it finds dirty buffers, and monitors the wakeup queue for tasks whose pages are all written to disk. Since the log writer patrols the dirty buffer chain, it knows when a buffer is ready to write to disk.

Enabling and disabling merge joins

By default, merge joins are enabled at the server level, for allrows mix and for allrows_dss optgoal, and are disabled at the server level for other optgoals, including allrows_oltp. When merge joins are disabled, the server costs only the other join types that are not disabled. To enable merge joins server-wide, set enable merge join to 1. The enable sort-merge joins and JTC configuration parameter from versions of Adaptive Server earlier than 15.0 does not affect the 15.0 and later query processor.

The command `set merge_join on` overrides the server level to allow use of merge joins in a session or stored procedure.

To enable merge joins, use:

```
set merge_join on
```

To disable merge joins, use:

```
set merge_join off
```

Enabling and disabling hash joins

By default, hash joins are enabled only when you run `allows_dss optgoal`. To override the server level setting, and allow use of hash join in a session or stored procedure, use `set hash_join on`.

To enable hash joins, use:

```
set hash_join on
```

To disable hash joins, use:

```
set hash_join off
```

Enabling and disabling *join* transitive closure

In Adaptive Server version 15.0 and later, join transitive closure is always on and cannot be disabled. The search engine uses the timeout mechanism to avoid excessive optimization time. Although the timeout setting no longer affects the actual use of transitive closure for the query processor, it can still affect the initial join order with which the search engine begins the permutation when the timeout occurs. You may find this discussion useful when you suspect that a suboptimal join order is being chosen at timeout.

By default, join transitive closure is not enabled at the server level, since it can increase optimization time. You can enable join transitive closure at a session level with `set jtc on`. The session-level command overrides the server-level setting for the enable sort-merge joins and JTC configuration parameter (available for versions of Adaptive Server earlier than 15.0).

For queries that execute quickly, even when several tables are involved, join transitive closure may increase optimization time with little improvement in execution cost. For example, with join transitive closure applied to this query, the number of possible joins is multiplied for each added table:

```
select * from t1, t2, t3, t4, ... tN
where t1.c1 = t2.c1
and t1.c1 = t3.c1
and t1.c1 = t4.c1
...
and t1.c1 = tN.c1
```

For joins on very large tables, however, the additional optimization time involved in costing the join orders added by join transitive closure may result in a join order that greatly improves the response time.

Use set statistics time to see how long Adaptive Server takes to optimize the query. If running queries with set jtc on greatly increases optimization time, but also improves query execution by choosing a better join order, check the showplan, set option show_search_engine normal, or set option show_search_engine long output. Explicitly add the useful join orders to the query text. Run the query without join transitive closure, and get the improved execution time, without the increased optimization time of examining all possible join orders generated by join transitive closure.

You can also enable join transitive closure and save abstract plans for queries that benefit. If you then execute those queries with loading from the saved plans enabled, the saved execution plan is used to optimize the query, making optimization time extremely short.

See *Performance and Tuning: Optimizer and Abstract Plans* for more information on using abstract plans and configuring join transitive closure server-wide.

Controlling literal parameterization

Adaptive Server version 15.0.1 and later allow you to automatically convert literal values in SQL queries to parameter descriptions (similar to variables).

To enable or disable enable literal autoparam server-wide, use:

```
sp_configure "enable literal autoparam", [0 | 1]
```


Where 1 automatically converts literal values to parameter descriptions, and 0 (the default) disables the feature.

Set literal parameterization at the session level using:

```
set literal_autoparam [off | on]
```

In versions of Adaptive Server earlier than 15.0.1, two queries that were identical except for one or more literal values resulted in the statement cache storing two separate query plans, or two additional rows, in sysqueryplans. For example, the query plans for these queries were stored separately, even though they are almost identical:

```
select count(*) from titles where total_sales > 100
select count(*) from titles where total_sales > 200
```

Examples

If you enable automatic literal parameterization, the SQL text of the `select count(*)` example referred to above is converted to:

```
select count(*) from titles where total_sales > @@@V0_INT
```

Where @@@V0_INT is an internally generated name for the parameter that represents the literal values 100 and 200.

All instances of literal values in the SQL text are replaced by internally generated parameters. For example:

```
select substring(name, 3, 4) from sysobjects where name in
('systypes', 'syscolumns')
```

is transformed to:

```
select substring(name, 3, 4) from sysobjects where name in
(@@@V0_VCHAR1, @@@V1_VCHAR1)
```

Any combination of values that replace the literals, 3, 4, `systypes` and `syscolumns` is transformed to the same SQL text with the same parameters and shares the same query plan when you enable the statement cache.

Automatic literal parameterization:

- Reduces compilation time on the second—and subsequent—executions of the query, regardless of the literal values in the query.
- Reduces the amount of SQL text storage space, including memory usage in the statement cache and the number of rows in sysqueryplans for abstract plans and query metrics.
- Reduces the amount of procedure cache used to store query plans.
- Occurs automatically within Adaptive Server, when enabled: you need not change the applications that submit the queries to Adaptive Server.

Usage issues for automatic literal parameterization include:

- Adaptive Server parameterizes the literals only for select, delete, update, and insert. For insert statements, Adaptive Server parameterizes only insert ... select statements, not insert ... values statements.
- Adaptive Server does not parameterize literals in queries that include a derived table.
- Adaptive Server does not parameterize queries similar to `select id + 1 from sysobjects group by id + 1` or `select id + 1 from sysobjects order by id + 1` because of the expressions ("id + 1") in the group by and order by clauses.
- Adaptive Server does not cache SQL statements with text longer than 16384 bytes in the statement cache (SQL statements over 16K are not cached). Transforming literals in the SQL statement into variables can significantly expand the size of the SQL text (especially if there was a large number of literals). Enabling automatic literal parameterization may result in Adaptive Server not caching some SQL statements that it would otherwise have cached.
- If two SQL statements are the same except that their literal values have different datatypes, they are not transformed into matching SQL texts. For example, the following two SQL statements return the same results, but are parameterized differently because they use the different datatypes:

```
select name from sysobjects where id = 1
select name from sysobjects where id = 1.0
```

The parameterized versions of these statements are:

```
select name from sysobjects where id = @@@V0_INT
select name from sysobjects where id = @@@V0_NUMERIC
```

Suggesting a degree of parallelism for a query

The `parallel` and `degree_of_parallelism` extensions to the `from` clause of a `select` command allow users to restrict the number of worker processes used in a scan.

For a parallel partition scan to be performed, the `degree_of_parallelism` must be equal to or greater than the number of partitions. For a parallel index scan, specify any value for the `degree_of_parallelism`.

The syntax for the `select` statement is:

```

select...
  from {tablename}
    [(index index_name
      [parallel [degree_of_parallelism | 1]]
      [prefetch size] [lru|mru]]),
   {tablename} [(index_name
      [parallel [degree_of_parallelism | 1]]
      [prefetch size] [lru|mru])] ...

```

Table 7-7 shows how to combine the index and parallel keywords to obtain serial or parallel scans.

Table 7-7: Optimizer hints for serial and parallel execution

Use:	To specify a:
(index <i>tablename</i> parallel <i>N</i>)	Parallel partition scan
(index <i>index_name</i> parallel <i>N</i>)	Parallel index scan
(index <i>tablename</i> parallel 1)	Serial table scan
(index <i>index_name</i> parallel 1)	Serial index scan
(parallel <i>N</i>)	Parallel scan, with the choice of table or index scan left to the optimizer
(parallel 1)	Serial scan, with the choice of table or index scan left to the optimizer

When you specify the parallel degree for a table in a merge join, it affects the degree of parallelism used for both the scan of the table and the merge join.

You cannot use the parallel option if you have disabled parallel processing either at the session level with the `set parallel_degree 1` command, or at the server level with the parallel degree configuration parameter. The parallel option cannot override these settings.

If you specify a *degree_of_parallelism* that is greater than the maximum configured degree of parallelism, Adaptive Server ignores the hint.

The optimizer ignores hints that specify a parallel degree if any of the following conditions is true:

- The from clause is used in the definition of a cursor.
- parallel is used in the from clause of an inner query block of a subquery, and the optimizer does not move the table to the outermost query block during subquery flattening.
- The table is a view, a system table, or a virtual table.
- The table is the inner table of an outer join.
- The query specifies exists, min, or max on the table.

- The value for the max scan parallel degree configuration parameter is set to 1.
- An unpartitioned clustered index is specified or is the only parallel option.
- A nonclustered index is covered.
- The query is processed using the OR strategy.
- The select statement is used for an update or insert.

Query level *parallel* clause examples

To specify the degree of parallelism for a single query, include *parallel* after the table name. This example executes in serial:

```
select * from titles (parallel 1)
```

This example specifies the index to be used in the query, and sets the degree of parallelism to 5:

```
select * from titles
    (index title_id_clix parallel 5)
where ...
```

To force a table scan, use the table name instead of the index name.

Optimization goals

Adaptive Server lets you choose a query optimization goal that best suits your query environment:

- *fastfirstrow* – optimizes queries so that Adaptive Server returns the first few rows as quickly as possible.
- *allows_oltp* – optimizes queries so that Adaptive Server uses a limited number of optimization criteria (described in “Optimization criteria” on page 252) to find a good query plan. *allows_oltp* is most useful for purely OLTP queries.
- *allows_mixed* – optimizes queries so that Adaptive Server uses most available optimization techniques, including *merge_join* and *parallel*, to find the best query plan. *allows_mixed*, which is the default strategy, is most useful in a mixed-query environment.

- `allows_dss` – optimizes queries so that Adaptive Server uses all available optimization techniques to find the best query plan, including hash join, advanced aggregates processing, and bushy tree plan. `allows_dss` is most useful in a DSS environment.

Setting optimization goals

You can set the optimization goal at the server, session, or query level. The server-level optimization goal is overridden at the session level, which is overridden at the query level—which means you can set a different optimization goal at each level.

At the server level

To set the optimization goal at the server level, you can:

- Use the `sp_configure` command
- Modify the optimization goal configuration parameter in the Adaptive Server configuration file

For example, to set the optimization level for the server to `fastfirstrow`, enter:

```
sp_configure "optimization goal", 0, "fastfirstrow"
```

At the session level

To set the optimization goal at the session level, use `set plan optgoal`. For example, to modify the optimization goal for the session to `allows`, enter:

```
set plan optgoal allows_oltp
```

To verify the current optimization goal at the session level, enter:

```
select @@optgoal
```

At the query level

To set the optimization goal at the query level, use the `select` or other DML command. For example, to change the optimization goal to `allows_oltp` for the current query, enter:

```
select * from A order by A.a plan "(use optgoal allows_oltp)"
```

At the query level only, you can specify the number of rows that Adaptive Server quickly returns when you set `fastfirstrow` as the optimization goal. For example, enter:

```
select * from A order by A.a plan "(use optgoal fastfirstrow 5)"
```

Some exceptions

In general, you can set query-level optimization goals using `select`, `update`, and `delete` statements. However:

- You cannot set query-level optimization goals in pure insert statements, although you can set optimization goals in `select ... insert` statements.

- `fastfirstrow` is relevant only for `select` statements; it incurs an error when used with other DML statements.

Optimization criteria

You can set specific optimization criteria for each session. The optimization criteria represent specific algorithms or relational techniques that may or may not be considered when Adaptive Server creates a query plan. By setting individual optimization criteria on or off, you can fine-tune the query plan for the current session.

Note Each optimization goal has default settings for each optimization criterion. Resetting optimization criteria may interfere with the default settings of the current optimization goal and produce an error message—although Adaptive Server applies the new setting.

Sybase recommends that you set individual optimization criteria *only rarely and with caution* if you must fine-tune a particular query. Overriding optimization goal settings can overly complicate query administration. Always set optimization criteria *after* setting any existing session level `optgoal` setting; an explicit `optgoal` setting may return an optimization criteria to its default value.

See “Default optimization criteria” on page 254.

Setting optimization criteria

Use the `set` command to enable or disable individual criteria.

For example, to enable the hash join algorithm, enter:

```
set hash_join 1
```

To disable the hash join algorithm, enter:

```
set hash_join 0
```

To enable one option and disable another, enter:

```
set hash_join 1, merge_join 0
```

Criteria descriptions

Most criteria described here decides whether a particular query engine operator can be used in the final plan chosen by the optimizer.

The optimization criteria are:

- `hash_join` – determines whether the query processor may use the hash join algorithm. Hash joins may consume more runtime resources, but are valuable when the joining columns do not have useful indexes or when a relatively large number of rows satisfy the join condition, compared to the product of the number of rows in the joined tables.
- `hash_union_distinct` – determines whether the query processor may use the hash union distinct algorithm, which is not efficient if most rows are distinct.
- `merge_join` – determines whether the query processor may use the merge join algorithm, which relies on ordered input. `merge_join` is most valuable when input is ordered on the merge key—for example, from an index scan. `merge_join` is less valuable if sort operators are required to order input.
- `merge_union_all` – determines whether the query processor may use the merge algorithm for union all. `merge_union_all` maintains the ordering of the result rows from the union input. `merge_union_all` is particularly valuable if the input is ordered and a parent operator (such as merge join) benefits from that ordering. Otherwise, `merge_union_all` may require sort operators that reduce efficiency.
- `merge_union_distinct` – determines whether the query processor may use the merge algorithm for union. `merge_union_distinct` is similar to `merge_union_all`, except that duplicate rows are not retained. `merge_union_distinct` requires ordered input and provides ordered output.
- `multi_table_store_ind` – determines whether the query processor may use reformatting on the result of a multiple table join. Using `multi_table_store_ind` may increase the use of worktables.
- `nl_join` – determines whether the query processor may use the nested-loop-join algorithm.
- `opportunistic_distinct_view` – determines whether the query processor may use a more flexible algorithm when enforcing distinctness.
- `parallel_query` – determines whether the query processor may use parallel query optimization.
- `store_index` – determines whether the query processor may use reformatting, which may increase the use of worktables.
- `append_union_all` – determines whether the query processor may use the append union all algorithm.

- `bushy_search_space` – determines whether the query processor may use bushy-tree-shaped query plans, which may increase the search space, but provide more query plan options to improve performance.
- `distinct_hashing` – determines whether the query processor may use a hashing algorithm to eliminate duplicates, which is very efficient when there are few distinct values compared to the number of rows.
- `distinct_sorted` – determines whether the query processor may use a single-pass algorithm to eliminate duplicates. `distinct_sorted` relies on an ordered input stream, and may increase the number of sort operators if its input is not ordered.
- `group-sorted` – determines whether the query processor may use an on-the-fly grouping algorithm. `group-sorted` relies on an input stream sorted on the grouping columns, and it preserves this ordering in its output.
- `distinct_sorting` – determines whether the query processor may use the sorting algorithm to eliminate duplicates. `distinct_sorting` is useful when the input is not ordered (for example, if there is no index) and the output ordering generated by the sorting algorithm could benefit; for example, in a merge join.
- `group_hashing` – determines whether the query processor may use a group hashing algorithm to process aggregates.
- `index_intersection` – determines whether the query processor may use the intersection of multiple index scans as part of the query plan in the search space.

If all the algorithms of a relational operator are disabled, the query processor reenables a default algorithm. For example, if all join algorithms (`nl_join`, `m_join`, and `h_join`) are disabled, the query processor enables `nl_join`.

The query processor can also reenable `nl_join` for semantic reasons: for example, if the joining tables are not connected through equijoins.

Default optimization criteria

Each optimization goal—`fastfirstrow`, `allrows_oltp`, `allrows_mixed`, `allrows_dss`—has a default setting (on (1) or off (0)) for each optimization criterion. For example, the default setting for `merge_join` is off (0) for `fastfirstrow` and `allrows_oltp`, and on (1) for `allrows_mixed` and `allrows_dss`. See Table 7-8 for a list of default settings for each optimization criteria.

Sybase recommends that you reset the optimization goal and evaluate performance before changing optimization criteria. Change optimization criteria only if you must fine-tune a particular query.

Table 7-8: Default settings for optimization criteria

Optimization criteria	fastfirstrow	allrows_oltp	allrows_mixed	allrows_dss
append_union_all	1	1	1	1
bushy_search_space	0	0	0	1
distinct_sorted	1	1	1	1
distinct_sorting	1	1	1	1
group_hashing	1	1	1	1
group_sorted	1	1	1	1
hash_join	0	0	0	1
hash_union_distinct	1	1	1	1
index_intersection	0	0	0	1
merge_join	0	0	1	1
merge_union_all	1	1	1	1
multi_gt_store_ind	0	0	0	1
nl_join	1	1	1	1
opp_distinct_view	1	1	1	1
parallel_query	1	0	1	1
store_index	1	1	1	1

Limiting optimization time

You can use the optimization timeout limit configuration parameter to restrict the amount of time Adaptive Server spends optimizing a query. optimization timeout limit specifies the amount of time Adaptive Server can spend optimizing a query as a percentage of the total time spent processing the query.

The timeout is activated only if:

- At least one complete plan has been retained as the best plan, and
- The optimization timeout limit has been exceeded.

Use `sp_configure` to set optimization timeout limit at the server level. For example, to limit optimization time to 10 percent of total query processing time, enter:

```
sp_configure "optimization timeout limit", 10
```

To set optimization timeout limit at the session level, use:

```
set plan optimeoutlimit n
```

This command overrides the server setting.

The default value is 10 percent; you can specify any value from 1 to 1000.

At the server level, there is a separate configuration parameter, `optimization timeout limit`, for the server-level default timeout value within stored procedure compilations. The default value is 40 percent; you can specify any value from 1 to 4000.

For more information about optimization timeout limit, see “Limiting the time spent optimizing a query” on page 16.

Controlling parallel optimization

The goal of executing queries in parallel is to get the fastest response time, even if it involves more total work from the server.

To enable and control parallel processing, Adaptive Server provides these configuration parameters:

- number of worker processes
- max parallel degree
- max resource granularity
- max repartition degree

With the exception of number of worker processes, each of these parameters can be set at the server and the session level. To view the current session-level value of a parameter, use the `select` command. For example, to view the current value of `max resource granularity`, enter:

```
select @@resource_granularity
```

Note When set or viewed at the session level, these parameters do not include “max.”

number of worker processes

Use `number of worker processes` to specify the maximum number of worker processes that Adaptive Server can use at any one time for all simultaneously running parallel queries.

`number of worker processes` is a server-wide configuration parameter only; use `sp_configure` to set the parameter. For example, to set the maximum number of worker processes to 200, enter:

```
sp_configure "number of worker processes", 200
```

Specifying the number of worker processes available for parallel processing

Use `max parallel degree` to specify the maximum number of worker processes allowed per query. You can configure `max parallel degree` at the server or the session level.

For example, to set `max parallel degree` to 60 at the server level, enter:

```
sp_configure "max parallel degree", 60
```

To set `max parallel degree` to 60 at the session level, enter:

```
set parallel_degree 60
```

The value of `max parallel degree` must be equal to or less than the current value of `number of worker processes`. Setting `max parallel degree` to 1 turns off parallel processing—Adaptive Server scans all tables and indexes serially. To enable parallel partition scans, set `max parallel degree` equal to or greater than the number of partitions in the table you are querying.

max resource granularity

Use `max resource granularity` to specify the percentage of total memory that Adaptive Server can allocate to a single query. You can set the parameter at the server or session level.

For example, to set `max resource granularity` to 35 percent at the server level, enter:

```
sp_configure "max resource granularity", 35
```

To set `max resource granularity` to 35 percent at the session level, enter:

```
set resource_granularity 35
```

The value of this parameter can affect the query optimizer's choice of operators for a query. If max resource granularity is set low, many hash- and sort-based operators cannot be chosen. max resource granularity also affects the scheduling algorithm.

max repartition degree

Use max repartition degree to suggest a number of worker processes that the query processor can use to partition a data stream. You can set max repartition degree at the server or query level.

Note The value of max repartition degree is a suggestion only; the query processor decides the optimal number.

max repartition degree is most useful when the tables being queried are not partitioned, but partitioning the resultant data stream may improve performance by allowing concurrent SQL operations.

For example, to set max repartition degree to 15 at the server level, enter:

```
sp_configure "max repartition degree", 15
```

To set max repartition degree to 15 at the session level, enter:

```
set repartition_degree 15
```

The value of max repartition degree cannot exceed the current value of max parallel degree. Sybase recommends that you set the value of this parameter equal to or less than the number of CPUs or disk systems that can work in parallel.

Concurrency optimization for small tables

For data-only-locked tables of 15 pages or fewer, Adaptive Server does not consider a table scan if there is a useful index on the table. Instead, it always chooses the cheapest index that matches any search argument that can be optimized in the query. The locking required for an index scan provides higher concurrency and reduces the chance of deadlocks, although slightly more I/O may be required than for a table scan.

If concurrency on small tables is not an issue, and you want to optimize the I/O instead, use `sp_chgattribute` to disable this optimization. For example, to turn off concurrency optimization for a table:

```
sp_chgattribute tiny_lookup_table,
    "concurrency_opt_threshold", 0
```

With concurrency optimization disabled, the query processor can choose table scans when they require fewer I/Os.

You can also increase the concurrency optimization threshold for a table. This command sets the concurrency optimization threshold for a table to 30 pages:

```
sp_chgattribute lookup_table,
    "concurrency_opt_threshold", 30
```

The maximum value for the concurrency optimization threshold is 32,767. Setting the value to -1 enforces concurrency optimization for a table of any size; this setting may be useful when a table scan is chosen over indexed access, and the resulting locking results in increased contention or deadlocks.

The current setting is stored in `systabstats.conopt_thld` and is printed as part of `optdiag` output.

Changing the locking scheme

Concurrency optimization affects only data-only-locked tables. Table 7-9 shows the effect of changing the locking scheme.

Table 7-9: Effects of alter table on concurrency optimization settings

Changing from	Effect on stored value
Allpages to data-only	Set to 15, the default
Data-only to allpages	Set to 0
One data-only scheme to another	Configured value retained

Optimization for Cursors

This chapter discusses performance issues related to cursors. Cursors are a mechanism for accessing the results of a SQL select statement one row at a time (or several rows, if you use set cursors rows). Since cursors use a different model from ordinary set-oriented SQL, the way cursors use memory and hold locks has performance implications for your applications. In particular, cursor performance issues include locking at the page and at the table level, network resources, and overhead of processing instructions.

Topic	Page
Definition	261
Resources required at each stage	264
Cursor modes	266
Index use and requirements for cursors	267
Comparing performance with and without cursors	269
Locking with read-only cursors	272
Isolation levels and cursors	273
Partitioned heap tables and cursors	274
Optimizing tips for cursors	274

Definition

A cursor is a symbolic name that is associated with a select statement. It enables you to access the results of a select statement one row at a time. Figure 8-1 shows a cursor accessing the authors table.

Figure 8-1: Cursor example

Cursor with select * from authors where state = 'KY'	Result set
	➤ A978606525 Marcello Duncan KY
	➤ A937406538 Carton Nita KY
Programming can:	➤ A1525070956Porczyk Howard KY
- Examine a row	➤ A913907285 Bier Lane KY
- Take an action based on row values	

You can think of a cursor as a “handle” on the result set of a select statement. It enables you to examine and possibly manipulate one row at a time.

Set-oriented versus row-oriented programming

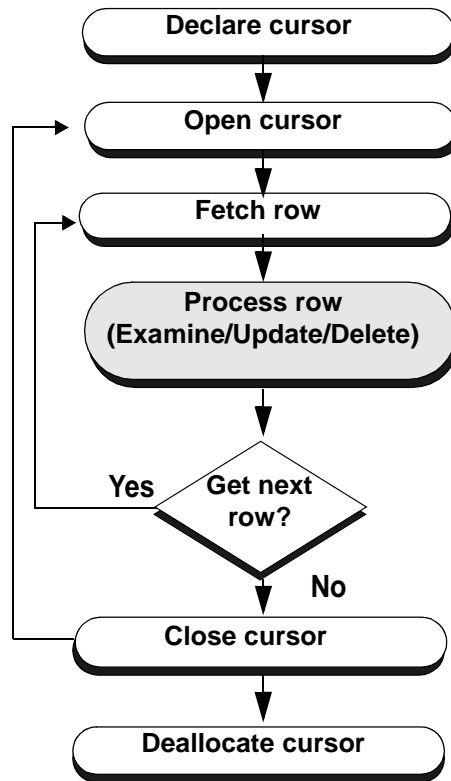
SQL was conceived as a set-oriented language. Adaptive Server is extremely efficient when it works in set-oriented mode. Cursors are required by ANSI SQL standards; when they are needed, they are very powerful. However, they can have a negative effect on performance.

For example, this query performs the identical action on all rows that match the condition in the where clause:

```
update titles
    set contract = 1
where type = 'business'
```

The optimizer finds the most efficient way to perform the update. In contrast, a cursor would examine each row and perform single-row updates if the conditions were met. The application declares a cursor for a select statement, opens the cursor, fetches a row, processes it, goes to the next row, and so forth. The application may perform quite different operations depending on the values in the current row, and the server’s overall use of resources for the cursor application may be less efficient than the server’s set level operations. However, cursors can provide more flexibility than set-oriented programming.

Figure 8-2 shows the steps involved in using cursors. The function of cursors is to get to the middle box, where the user or application code examines a row and decides what to do, based on its values.

Figure 8-2: Cursor flowchart

Example

Here is a simple example of a cursor with the “Process Rows” step shown above in pseudocode:

```

declare biz_book cursor
  for select * from titles
    where type = 'business'
go
open biz_book
go
fetch biz_book
go
/* Look at each row in turn and perform
** various tasks based on values,

```

```
** and repeat fetches, until  
** there are no more rows  
*/  
close biz_book  
go  
deallocate cursor biz_book  
go
```

Depending on the content of the row, the user might delete the current row:

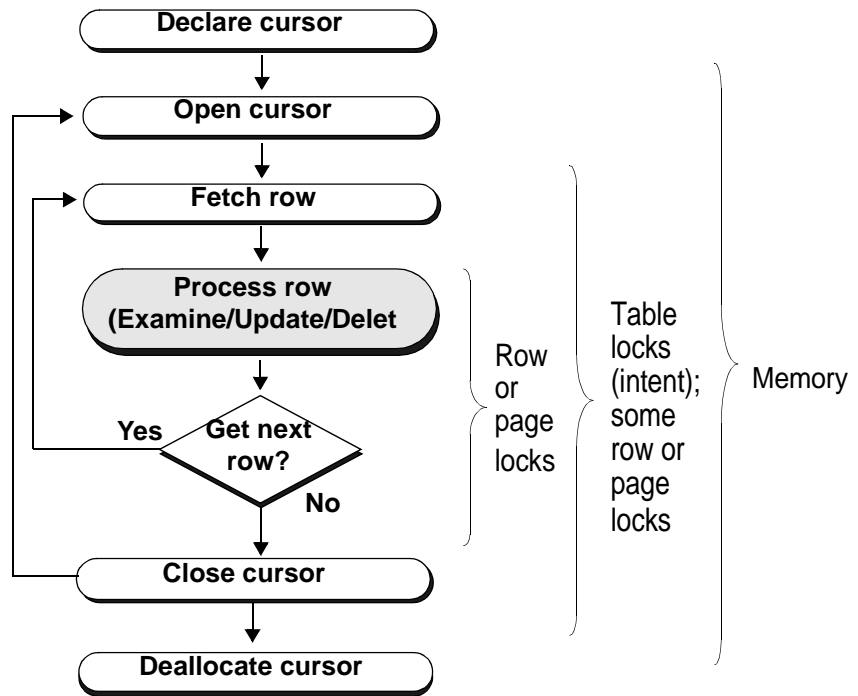
```
delete titles where current of biz_book
```

or update the current row:

```
update titles set title="The Rich  
Executive's Database Guide"  
where current of biz_book
```

Resources required at each stage

Cursors use memory and require locks on tables, data pages, and index pages. When you open a cursor, memory is allocated to the cursor and to store the query plan that is generated. While the cursor is open, Adaptive Server holds intent table locks and sometimes row or page locks. Figure 8-3 shows the duration of locks during cursor operations.

Figure 8-3: Resource use by cursor statement

The memory resource descriptions in Figure 8-3 and Table 8-1 refer to ad hoc cursors for queries sent by isql or Client-Library™. For other kinds of cursors, the locks are the same, but the memory allocation and deallocation differ somewhat depending on the type of cursor being used, as described in “Memory use and execute cursors” on page 266.

Table 8-1: Locks and memory use for isql and Client-Library client cursors

Cursor command	Resource use
declare cursor	When a cursor is declared, Adaptive Server uses only enough memory to store the query text.

Cursor command	Resource use
open	<p>When a cursor is opened, Adaptive Server allocates memory to the cursor and to store the query plan that is generated. The server optimizes the query, traverses indexes, and sets up memory variables. The server does not access rows yet, unless it needs to build worktables. However, it does set up the required table-level locks (intent locks). Row and page locking behavior depends on the isolation level, server configuration, and query type.</p> <p>See the <i>Performance and Tuning Series: Locking and Concurrency Control</i> for more information.</p>
fetch	<p>When a fetch is executed, Adaptive Server gets the row(s) required and reads specified values into the cursor variables or sends the row to the client. If the cursor needs to hold lock on rows or pages, the locks are held until a fetch moves the cursor off the row or page or until the cursor is closed. The lock is either a shared or an update lock, depending on how the cursor is written.</p>
close	<p>When a cursor is closed, Adaptive Server releases the locks and some of the memory allocation. You can open the cursor again, if necessary.</p>
deallocate cursor	<p>When a cursor is deallocated, Adaptive Server releases the rest of the memory resources used by the cursor. To reuse the cursor, declare it again.</p>

Memory use and execute cursors

The descriptions of `declare cursor` and `deallocate cursor` in Table 8-1 refer to ad hoc cursors that are sent by `isql` or Client-Library. Other kinds of cursors allocate memory differently:

- For cursors that are declared *on* stored procedures, only a small amount of memory is allocated at `declare cursor` time. Cursors declared on stored procedures are sent using Client-Library or the precompiler and are known as `execute cursors`.
- For cursors declared *within* a stored procedure, memory is already available for the stored procedure, and the `declare` statement does not require additional memory.

Cursor modes

There are two cursor modes: read-only and update. As the names suggest, read-only cursors can only display data from a `select` statement; update cursors can be used to perform positioned updates and deletes.

Read-only mode uses shared page or row locks. If read committed with lock is set to 0, and the query runs at isolation level 1, it uses instant duration locks, and does not hold the page or row locks until the next fetch.

Read-only mode is in effect when you specify for read only or when the cursor's select statement uses distinct, group by, union, or aggregate functions, and in some cases, an order by clause.

Update mode uses update page or row locks. It is in effect when:

- You specify for update.
- The select statement does not include distinct, group by, union, a subquery, aggregate functions, or the at isolation read uncommitted clause.
- You specify shared.

If *column_name_list* is specified, only those columns are updatable.

See the *Performance and Tuning Series: Locking and Concurrency Control* for more information.

Specify the cursor mode when you declare the cursor. If the select statement includes certain options, the cursor is not updatable even if you declare it for update.

Index use and requirements for cursors

When a query is used in a cursor, it may require or choose different indexes than the same query used outside of a cursor.

Allpages-locked tables

For read-only cursors, queries at isolation level 0 (dirty reads) require a unique index. Read-only cursors at isolation level 1 or 3 should produce the same query plan as the select statement outside of a cursor.

The index requirements for updatable cursors mean that updatable cursors may use different query plans than read-only cursors. Updatable cursors have these indexing requirements:

- If the cursor is not declared for update, a unique index is preferred over a table scan or a nonunique index.

- If the cursor is declared for update *without* a for update of list, a unique index is required on allpages-locked tables. An error is raised if no unique index exists.
- If the cursor is declared for update with a for update of list, then only a unique index *without* any columns from the list can be chosen on an allpages-locked table. An error is raised if no unique index qualifies.

When cursors are involved, an index that contains an IDENTITY column is considered unique, even if the index is not declared unique. In some cases, IDENTITY columns must be added to indexes to make them unique, or the optimizer might be forced to choose a suboptimal query plan for a cursor query.

Data-only-locked tables

In data-only-locked tables, fixed row IDs are used to position cursor scans, so unique indexes are not required for dirty reads or updatable cursors. The only cause for different query plans in updatable cursors is that table scans are used if columns from only useful indexes are included in the for update of list.

Table scans to avoid the Halloween problem

The Halloween problem is an update anomaly that can occur when a client using a cursor updates a column of the cursor result-set row, and that column defines the order in which the rows are returned from the table. For example, if a cursor was to use an index on last_name, first_name, and update one of these columns, the row could appear in the result set a second time.

To avoid the Halloween problem on data-only-locked tables, Adaptive Server chooses a table scan when the columns from an otherwise useful index are included in the column list of a for update clause.

For implicitly updatable cursors declared without a for update clause, and for cursors where the column list in the for update clause is empty, cursors that update a column in the index used by the cursor may encounter the Halloween problem.

Comparing performance with and without cursors

This section examines the performance of a stored procedure written two different ways:

- Without a cursor – this procedure scans the table three times, changing the price of each book.
- With a cursor – this procedure makes only one pass through the table.

In both examples, there is a unique index on titles(title_id).

Sample stored procedure without a cursor

This is an example of a stored procedure without cursors:

```

/* Increase the prices of books in the
** titles table as follows:
**
** If current price is <= $30, increase it by 20%
** If current price is > $30 and <= $60, increase
** it by 10%
** If current price is > $60, increase it by 5%
**
** All price changes must take effect, so this is
** done in a single transaction.
*/

create procedure increase_price
as

    /* start the transaction */
    begin transaction
    /* first update prices > $60 */
    update titles
        set price = price * 1.05
        where price > $60

    /* next, prices between $30 and $60 */
    update titles
        set price = price * 1.10
    where price > $30 and price <= $60

    /* and finally prices <= $30 */
    update titles
        set price = price * 1.20

```

```
        where price <= $30

        /* commit the transaction */
        commit transaction

    return
```

Sample stored procedure with a cursor

This procedure performs the same changes to the underlying table as the procedure written without a cursor, but it uses cursors instead of set-oriented programming. As each row is fetched, examined, and updated, a lock is held on the appropriate data page. Also, as the comments indicate, each update commits as it is made, since there is no explicit transaction.

```
/* Same as previous example, this time using a
** cursor. Each update commits as it is made.
*/
create procedure increase_price_cursor
as
declare @price money

/* declare a cursor for the select from titles */
declare curs cursor for
    select price
    from titles
    for update of price

/* open the cursor */
open curs

/* fetch the first row */
fetch curs into @price

/* now loop, processing all the rows
** @@sqlstatus = 0 means successful fetch
** @@sqlstatus = 1 means error on previous fetch
** @@sqlstatus = 2 means end of result set reached
*/
while (@@sqlstatus != 2)
begin
    /* check for errors */
    if (@@sqlstatus = 1)
    begin
        print "Error in increase_price"
```



```

        return
    end

    /* next adjust the price according to the
    ** criteria
    */
    if @price > $60
        select @price = @price * 1.05
    else
        if @price > $30 and @price <= $60
            select @price = @price * 1.10
        else
            if @price <= $30
                select @price = @price * 1.20
            end
        end
    end

    /* now, update the row */
    update titles
    set price = @price
    where current of curs

    /* fetch the next row */
    fetch curs into @price
end

/* close the cursor and return */
close curs
return

```

Which procedure do you think will have better performance, one that performs three table scans or one that performs a single scan via a cursor?

Cursor versus noncursor performance comparison

Table 8-2 shows statistics gathered against a 5000-row table. The cursor code takes over 4 times longer, even though it scans the table only once.

Table 8-2: Sample execution times against a 5000-row table

Procedure	Access method	Time
increase_price	Uses three table scans	28 seconds
increase_price_cursor	Uses cursor, single table scan	125 seconds

Results from tests like these can vary widely. They are most pronounced on systems that have busy networks, a large number of active database users, and multiple users accessing the same table.

In addition to locking, cursors involve more network activity than set operations and incur the overhead of processing instructions. The application program needs to communicate with Adaptive Server regarding every result row of the query. This is why the cursor code took much longer to complete than the code that scanned the table three times.

Cursor performance issues include:

- Locking at the page and table level
- Network resources
- Overhead of processing instructions

If there is a set-level programming equivalent, it may be preferable, even if it involves multiple table scans.

Locking with read-only cursors

Here is a piece of cursor code you can use to display the locks that are set up at each point in the life of a cursor. The following example uses an allpages-locked table. Execute the code in Figure 8-4, and pause at the arrows to execute `sp_lock` and examine the locks that are in place.

Figure 8-4: Read-only cursors and locking experiment input

```

declare curs1 cursor for
select au_id, au_lname, au_fname
  from authors
  where au_id like '15%'
  for read only
go
open curs1
go
fetch curs1
go
fetch curs1
go 100
close curs1
go
deallocate cursor curs1
go

```

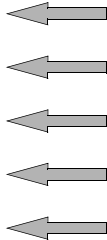


Table 8-3 shows the results.

Table 8-3: Locks held on data and index pages by cursors

Event	Data page
After declare	No cursor-related locks.
After open	Shared intent lock on authors.
After first fetch	Shared intent lock on authors and shared page lock on a page in authors.
After 100 fetches	Shared intent lock on authors and shared page lock on a different page in authors.
After close	No cursor-related locks.

If you issue another `fetch` command after the last row of the result set has been fetched, the locks on the last page are released, so there will be no cursor-related locks.

With a data-only-locked table:

- If the cursor query runs at isolation level 1, and read committed with lock is set to 0, you do not see any page or row locks. The values are copied from the page or row, and the lock is immediately released.
- If read committed with lock is set to 1 or if the query runs at isolation level 2 or 3, you see either shared page or shared row locks at the point that Table 8-3 indicates shared page locks. If the table uses datarows locking, the `sp_lock` report includes the row ID of the fetched row.

Isolation levels and cursors

The query plan for a cursor is compiled and optimized when the cursor is opened. You cannot open a cursor and then use `set transaction isolation level` to change the isolation level at which the cursor operates.

Since cursors using isolation level 0 are compiled differently from those using other isolation levels, you cannot open a cursor at isolation level 0 and open or fetch from it at level 1 or 3. Similarly, you cannot open a cursor at level 1 or 3 and then fetch from it at level 0. Attempts to fetch from a cursor at an incompatible level result in an error message.

Once the cursor has been opened at a particular isolation level, you must deallocate the cursor before changing isolation levels. The effects of changing isolation levels while the cursor is open are as follows:

- Attempting to close and reopen the cursor at another isolation level fails with an error message.

- Attempting to change isolation levels without closing and reopening the cursor has no effect on the isolation level in use and does not produce an error message.

You can include an `at isolation` clause in the cursor to specify an isolation level. The cursor in the example below can be declared at level 1 and fetched from level 0 because the query plan is compatible with the isolation level:

```
declare cprice cursor for
select title_id, price
   from titles
  where type = "business"
  at isolation read uncommitted
```

Partitioned heap tables and cursors

A cursor scan of an unpartitioned heap table can read all data up to and including the final insertion made to that table, even if insertions took place after the cursor scan started.

If a heap table is partitioned, data can be inserted into one of the many page chains. The physical insertion point may be before or after the current position of a cursor scan. This means that a cursor scan against a partitioned table is *not* guaranteed to scan the final insertions made to that table.

Note If cursor operations require all inserts to be made at the end of a single page chain, *do not* partition the table used in the cursor scan.

Optimizing tips for cursors

Here are several optimizing tips for cursors:

- Optimize cursor selects using the cursor, not an ad hoc query.
- Use union or union all instead of or clauses or in lists.
- Declare the cursor's intent.
- Specify column names in the for update clause.

- Fetch more than one row if you are returning rows to the client.
- Keep cursors open across commits and rollbacks.
- Open multiple cursors on a single connection.

Optimizing for cursor selects using a cursor

A standalone select statement may be optimized very differently than the same select statement in an implicitly or explicitly updatable cursor. When you are developing applications that use cursors, always check your query plans and I/O statistics using the cursor, rather than using a standalone select. In particular, index restrictions of updatable cursors require very different access methods.

Using *union* instead of *or* clauses or *in* lists

Cursors cannot use the dynamic index of row IDs generated by the OR strategy. Queries that use the OR strategy in standalone select statements usually perform table scans using read-only cursors. Updatable cursors may need to use a unique index and still require access to each data row, in sequence, in order to evaluate the query clauses.

A read-only cursor using union creates a worktable when the cursor is declared, and sorts it to remove duplicates. Fetches are performed on the worktable. A cursor using union all can return duplicates and does not require a worktable.

Declaring the cursor's intent

Always declare a cursor's intent: read-only or updatable. This gives you greater control over concurrency implications. If you do not specify the intent, Adaptive Server decides for you, and very often it chooses updatable cursors. Updatable cursors use update locks, thereby preventing other update locks or exclusive locks. If the update changes an indexed column, the optimizer may need to choose a table scan for the query, resulting in potentially difficult concurrency problems. Be sure to examine the query plans for queries that use updatable cursors.

Specifying column names in the *for update* clause

Adaptive Server acquires update locks on the pages or rows of all tables that have columns listed in the *for update* clause of the cursor *select* statement. If the *for update* clause is not included in the cursor declaration, all tables referenced in the *from* clause acquire update locks.

The following query includes the name of the column in the *for update* clause, but acquires update locks only on the *titles* table, since *price* is mentioned in the *for update* clause. The table uses allpages locking. The locks on *authors* and *titleauthor* are shared page locks:

```
declare curs3 cursor
for
select au_lname, au_fname, price
      from titles t, authors a,
           titleauthor ta
where advance <= $1000
      and t.title_id = ta.title_id
      and a.au_id = ta.au_id
for update of price
```

Table 8-4 shows the effects of:

- Omitting the *for update* clause entirely—no shared clause
- Omitting the column name from the *for update* clause
- Including the name of the column to be updated in the *for update* clause
- Adding *shared* after the name of the *titles* table while using *for update of price*

In this table, the additional locks, or more restrictive locks for the two versions of the *for update* clause are emphasized.

Table 8-4: Effects of for update clause and shared on cursor locking

Clause	titles	authors	titleauthor
None		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data
for update	<i>update on index</i>	<i>update on index</i>	
	update on data	<i>update on data</i>	<i>update on data</i>
for update of price		sh_page on index	
	update on data	sh_page on data	sh_page on data
for update of price + shared		sh_page on index	
	sh_page on data	sh_page on data	sh_page on data

Using set cursor rows

The SQL standard specifies a one-row fetch for cursors, which wastes network bandwidth. Using the set cursor rows query option and Open Client's transparent buffering of fetches, you can improve performance:

```
ct_cursor(CT_CURSOR_ROWS)
```

Be careful when you choose the number of rows returned for frequently executed applications using cursors—tune them to the network.

See the *Performance and Tuning Series: Basics* for an explanation of this process.

Keeping cursors open across commits and rollbacks

ANSI closes cursors at the conclusion of each transaction. Transact-SQL provides the set option close on endtran for applications that must meet ANSI behavior. By default, however, this option is turned off. Unless you must meet ANSI requirements, leave this option off to maintain concurrency and throughput.

If you must be ANSI-compliant, decide how to handle the effects on Adaptive Server. Should you perform a lot of updates or deletes in a single transaction? Or should you keep the transactions short?

If you choose to keep transactions short, closing and opening the cursor can affect throughput, since Adaptive Server needs to rematerialize the result set each time the cursor is opened. If you choose to perform more work in each transaction, this can cause concurrency problems, since the query holds locks.

Opening multiple cursors on a single connection

Some developers simulate cursors by using two or more connections from DB-Library™. One connection performs a select and the other performs updates or deletes on the same tables. This has very high potential to create application deadlocks. For example:

- Connection A holds a shared lock on a page. As long as there are rows pending from Adaptive Server, a shared lock is kept on the current page.
- Connection B requests an exclusive lock on the same pages and then waits.
- The application waits for Connection B to succeed before invoking whatever logic is needed to remove the shared lock. But this never happens.

Since Connection A never requests a lock that is held by Connection B, this is not a server-side deadlock.

Topic	Page
Overview	279
Executing QP metrics	280
Accessing metrics	280
Using metrics	282
Clearing metrics	284
Restricting query metrics capture	285
Understanding the UID in sysquerymetrics	286

Overview

Query processing (QP) metrics identify and compare empirical metric values in query execution. When a query is executed, it is associated with a set of defined metrics that are the basis for comparison in QP metrics.

Captured metrics include:

- CPU execution time – the time, in milliseconds, it takes to execute the query.
- Elapsed time – the time, in milliseconds, from after the compile to the end of the execution.
- Logical I/O – the number of logical I/O reads.
- Physical I/O – the number of physical I/O reads.
- Count – the number of times a query is executed.
- Abort count – the number of times a query is aborted by the resource governor due to a resource limit being exceeded.

Each metric, except count and abort count, has three values: minimum, maximum, and average.

Executing QP metrics

You can activate and use QP metrics at the server level or at the session level.

At the server level, use `sp_configure` with the `enable metrics capture` option. The QP metrics for ad hoc statements are captured directly into a system catalog, while the QP metrics for statements in a stored procedure are saved in a procedure cache. When the stored procedure or query in the statement cache is flushed, the respective captured metrics are written to the system catalog.

```
sp_configure "enable metrics capture", 1
```

At a session level, use `set metrics_capture` on/off:

```
set metrics_capture on/off
```

Accessing metrics

QP metrics are always captured in the default group, which is group 1 in each respective database. Use `sp_metrics 'backup'` to move saved QP metrics from the default running group to a backup group. Access metric information using a `select` statement with `order by` against the `sysquerymetrics` view. See “`sysquerymetrics` view” on page 280 for details.

You can also use a data manipulation language (DML) statement to sort the metric information and identify the specific queries for evaluation. See Chapter 2, “Understand Component Integration Services,” in the *Component Integration Services Users Guide*, which is part of the Adaptive Server Enterprise documentation set. .

sysquerymetrics view

Field	Definition
uid	User ID
gid	Group ID
id	Unique ID
hashkey	Hash key over the SQL query text
sequence	Sequence number for a row when multiple rows are required for the text of the SQL code
exec_min	Minimum execution time

Field	Definition
exec_max	Maximum execution time
exec_avg	Average execution time
elap_min	Minimum elapsed time
elap_max	Maximum elapsed time
elap_avg	Average elapsed time
lio_min	Minimum logical I/O
lio_max	Maximum logical I/O
lio_avg	Average logical I/O
pio_min	Minimum physical I/O
pio_max	Maximum physical I/O
pio_avg	Average physical I/O
cnt	Number of times the query has been executed
abort_cnt	Number of times a query is aborted by the resource governor when a resource limit is exceeded
qtext	Query text

Average values in this view are calculated using:

$$\text{new_avg} = (\text{old_avg} * \text{old_count} + \text{new_value}) / (\text{old_count} + 1) = \text{old_avg} + \text{round}((\text{new_value} - \text{old_avg}) / (\text{old_count} + 1))$$

This is an example of the sysquerymetrics view:

`select * from sysquerymetrics`

```

uid  gid  hashkey  id  sequence  exec_min
exec_max  exec_avg  elap_min  elap_max  elap_avg  lio_min
lio_max  lio_avg  pio_min  pio_max  pio_avg  cnt  abort_cnt
qtext
-----
-----
-----
-----
1  1  106588469  480001710  0  0
0  0  16  33  25  4
4  4  0  4  2  2  0
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)

```

The above example displays a record for a SQL statement. The query text of the statement is `select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)`:

- This statement has been executed twice so far (cnt = 2).

- The minimum elapsed time is 16 milliseconds; the maximum elapsed time is 33 milliseconds, and the average elapsed time is 25 milliseconds
- All the execution times are 0, and this may be due to the CPU execution time being less than 1 millisecond.
- The maximum physical I/O is 4, which is consistent with the maximum logical I/O. However, the minimum physical I/O is 0 because data is already in cache in the second run. The logical I/O, at 4, should be static whether or not the data is in memory

Using metrics

Use the information produced by QP metrics to identify:

- Query performance regression
- Most expensive query in a batch of running queries
- Most frequently run queries

When you have information on the queries that may be causing problems, you can tune the queries to increase efficiency.

For example, identifying and fine-tuning an expensive query may be more effective than tuning the cheaper ones in the same batch.

You can also identify the queries that are run most frequently, and fine-tune them to increase efficiency.

Turning on query metrics may involve extra I/O for every query executed, so there may be performance impact. However, also consider the benefits mentioned above. You may want to gather statistical information from monitoring tables instead of turning on metrics.

Both QP metrics and monitoring tables can be used to gather statistical information. However, you can use QP metrics instead of the monitoring tables to gather aggregated historical query information in a persistent catalog, rather than have transient information from the monitor tables.

Examples

You can use QP metrics to identify specific queries for tuning and possible regression on performance.

Identifying the most expensive statement

Typically, to find the most expensive statement as the candidate for tuning, `sysquerymetrics` provides CPU execution time, elapsed time, logical IO, and physical I/O as options for measure. For example, a typical measure is based on logical I/O. Use the following query to find the statements that incur too many IOs as the candidates for tuning:

```
select lio_avg, qtext from sysquerymetrics order by lio_avg
lio_avg qtext
-----
2
select c1, c2 from t_metrics1 where c1 = 333
4
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
6
select count(t_metrics1.c1) from t_metrics1, t_metrics2,
t_metrics3 where (t_metrics1.c2 = t_metrics2.c2 and
t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 0)
164
select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from
t_metrics2, t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3
= 1))

(4 rows affected)
```

The best candidate for tuning can be seen in the last statement of the above results, which has the biggest value (164) for average logical IO.

Identifying the most frequently used statement for tuning

If a query is used frequently, fine-tuning may improve its performance. Identify the most frequently used query using the `select` statement with `order by`:

```
select elap_avg, cnt, qtext from sysquerymetrics order by cnt
elap_avg cnt
qtext
-----
0          1
```

```
select c1, c2 from t_metrics1 where c1 = 333
16          2
```

```
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
24          3
```

```
select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from
t_metrics2, t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3
= 1))
```

```
78          4
```

```
select count(t_metrics1.c1) from t_metrics1, t_metrics2, t_metrics3 where
(t_metrics1.c2 = t_metrics2.c2 and t_metrics2.c2 = t_metrics3.c2 and
t_metrics3.c3 = 0)
```

```
(4 rows affected)
```

The best candidate for tuning can be seen in the last statement of the above results, which has the biggest value (78).

Identifying possible performance regression

In some cases, when a server is upgraded to a newer version, QP metrics may be useful for comparing performance. To identify queries that may have some degradation, after a server-version upgrade:

- 1 Back up the QP metrics from the old server into a backup group:

```
sp_metrics 'backup', '@gid'
```
- 2 Enable QP metrics on the new server:

```
sp_configure "enable metrics capture", 1
```
- 3 Compare QP metrics output from the old and new servers to identify any queries that may have regression problems.

Clearing metrics

Use `sp_metrics 'flush'` to flush all aggregated metrics in memory to the system catalog. The aggregated metrics for all statements in memory are set to zero.

```
sp_metrics 'drop', '@gid' [, '@id'
```

To remove one entry, use:

```
sp_metrics 'drop', '<gid>', '<id>'
```

You can also use `filter` to remove QP metrics from the system catalog, based on some metrics conditions:

```
sp_metrics 'filter', '@gid', ['@predicate']
```

This example deletes all QP metrics in group 1 where `lio_max < 100`:

```
sp_metrics 'filter', '1', 'lio_max < 100'
```

Restricting query metrics capture

These configuration parameters set the query metrics threshold for capture into the catalog:

- `metrics lio max`
- `metrics pio max`
- `metrics elap max`
- `metrics exec max`

These parameters let you filter out trivial metrics before writing metrics information to the catalog.

By default, these configuration parameters are set to 0 (off).

For example, to not capture those query plans for which `lio` is less than 10, use:

```
sp_configure 'metrics lio max', 10
```

If you do not set any of these configuration parameters, Adaptive Server captures the query metrics to the system tables. However, if you set any of these configuration parameters, Adaptive Server uses only those nonzero configuration parameters as thresholds for determining whether to capture query metrics.

For example, if you set `metrics elap max` to a non-zero value, but no others, query metrics are captured only if the elapsed time is bigger than the configured value. Because the other three configuration parameters are set to 0, they do not act as thresholds for capturing metrics.

Understanding the UID in *sysquerymetrics*

The user ID (UID) of *sysquerymetrics* is 0 when all table names in a query that are not qualified by user name are owned by the database owner.

Example 1

```
select * from t1 where c1 = 1
```

t1 is owned by database owner and is shared by different users. 0 is the UID for the entry into *sysquerymetrics* no matter which user issues the query.

Example 2

```
select * from t2 where c1 = 1
```

In this case, t2 is owned by *user1*. *user1*'s UID is used for the entry in *sysquerymetrics*, since t2 is unqualified and is not owned by the database owner.

Example 3

```
select * from u1.t3 where c1 = 1
```

Here, t3 is owned by *u1* and is qualified by *u1*, so UID 0 is used.

This increases the sharing of metrics between user IDs to reduce the number of entries in *sysqueryplans*. Aggregation of metrics for identical queries with different user IDs is done automatically. Turn on trace flag 15361 to use the UID of the user who issues the query.

Note QP metrics for insert...selec, /update, delete statements are captured when at least one table is involved. CIS-related queries and insert...values statements are not included.

Using Statistics to Improve Performance

Accurate statistics are essential to query optimization. In some cases, adding statistics for columns that are not leading index keys also improves query performance. This chapter explains how and when to use the commands that manage statistics.

Topic	Page
Statistics maintained in Adaptive Server	287
Importance of statistics	288
Updating statistics	290
Automatically updating statistics	294
Configuring automatic update statistics	297
Column statistics and statistics maintenance	300
Creating and updating column statistics	302
Choosing step numbers for histograms	306
Scan types, sort requirements, and locking	307
Using the delete statistics command	310
When row counts may be inaccurate	311

Statistics maintained in Adaptive Server

These key optimizer statistics are maintained in Adaptive Server:

- Statistics per partition – table row count; table page count. An unpartitioned table is considered to have one partition for the purposes of the systabstats catalog. Statistics per partition can be found in systabstats.

- Statistics per index: index row count – index height; index leaf page count. A local index has a separate systabstats row for each index partition. A global index, which is considered a partitioned index with one partition, has one systabstats row. Statistics per index: index row count can be found in systabstats.
- Statistics per column: data distribution. Statistics per column be found in sysstatistics.
- Statistics per group of column – density information. Statistics per group of column can be found in sysstatistics.
- Statistics per partition –
 - Column statistics – data distribution per column; density per group of columns. Column statistics can be found in sysstatistics.

Throughout this chapter, density is a statistical measurement of the uniqueness of a given column's values, and a histogram is a statistical representation of the distribution of values of a given column of the relation.

Importance of statistics

The Adaptive Server cost-based optimizer uses statistics about the tables, indexes, partitions, and columns named in a query to estimate query costs. It chooses the access method that the optimizer determines has the least cost. But this cost estimate cannot be accurate if statistics are not accurate.

Some statistics, such as the number of pages or rows in a table, are updated during query processing. Other statistics, such as the histograms on columns, are updated only when you execute update statistics, or when indexes are created.

If your query is performing slowly and you seek help from Technical Support or a Sybase newsgroup on the Internet, one of the first questions you are likely be asked is “Did you run update statistics?” Use the `optdiag` command to see when update statistics was last run for each column on which statistics exist:

```
Last update of column statistics: Aug 31 2004
4:14:17:180PM
```

Another command you may need for statistics maintenance is `delete statistics`. Dropping an index does not drop the statistics for that index. If the distribution of keys in the columns changes after the index is dropped, but the statistics are still used for some queries, the outdated statistics can affect query plans.

Histogram statistics from a global index are more accurate than histogram statistics generated by a local index. For a local index, statistics are created on each partition, and are then merged to create a global histogram using approximations as to how overlapping histogram cells from each partition should be combined. With a global index, the merge step, with merging estimates, does not occur. In most cases, there is no issue with update statistics on a local index. However, if there are significant estimation errors in queries involving partitioned tables, histogram accuracy can be improved by creating and dropping a global index on a column rather than updating the statistics on a local index.

Nonbinary character set histogram interpolation

Adaptive Server versions 15.0.2 and later allow selectivity estimates to have the same accuracy as the binary character set, without requiring an excessive number of histogram steps. This benefits queries like the following, which uses range predicates:

```
select * from t1 where charcolumn > "LMC0021" and  
charcolumn <= "LMC0029"
```

If ranges specified falls into the same histogram cell, Adaptive Server can much more accurately estimate this selectivity.

In versions of Adaptive Server earlier than 15.0.2, only the default binary character set benefited from histogram interpolation, which is used to estimate the selectivity of range predicates. For all other character sets, Adaptive Server made a selectivity estimate of 50 percent for a histogram cell. This typically required Adaptive Server to use a large number of histogram cells for character column histograms to reduce the error associated with this estimate.

Updating statistics

The update statistics command updates column-related statistics such as histograms and densities. Statistics must be updated on those columns where the distribution of keys in the index changes in ways that affect the use of indexes for your queries.

Running update statistics requires system resources. Like other maintenance tasks, Sybase recommends that you schedule it during at times when the load on the server is light. In particular, update statistics requires table scans or leaf-level scans of indexes, may increase I/O contention, may use the CPU to perform sorts, and uses the data and procedure caches. Use of these resources can adversely affect queries running on the server.

Using the sampling feature can reduce resource requirements and allow more flexibility when running this task.

Note Sampling does not affect the update statistics *table_name index_name* parameters]. Sampling affects only update index statistics and update all statistics on unindexed columns and update statistics *table_name (column_list)*.

In addition, some update statistics commands require shared locks, which may block updates. See “Scan types, sort requirements, and locking” on page 307.

You can also configure Adaptive Server to automatically run update statistics at times that have minimal impact on the system resources. See “Automatically updating statistics” on page 294.

Adding statistics for unindexed columns

When you create an index, a histogram is generated for the leading column in the index. Examples in earlier chapters have shown how statistics for other columns can increase the accuracy of optimizer statistics.

Consider adding statistics for virtually all columns that are frequently used as search arguments, as long as your maintenance schedule allows time to keep these statistics up to date.

In particular, adding statistics for minor columns of composite indexes can greatly improve cost estimates when those columns are used in search arguments or joins along with the leading index key.

Limitations for updating statistics on proxy tables and views

If you create indexes on proxy tables, you may experience performance degradation; you cannot run `create index`, because Adaptive Server cannot propagate the command to the remote view. However, if you map the proxy table to a user table, Adaptive Server can optimize the query plan by creating indexes on the proxy tables' join column.

If the number of rows returned by the view varies drastically from one execution to another, performance degrades when you gather statistics on proxy tables.

If you are using proxy tables or views, Sybase makes these recommendations:

- Do not run `update statistics` on proxy tables defined on views.
- If you ran `update statistics` on a proxy table defined on a view, increase performance by dropping, then re-creating the proxy table. You cannot use `delete statistics` to remove the current statistics, because `systabstats` retains the original row count.
- Convert the view to a permanent table.
- Try using the command line trace flag 318 (force reformatting).

update statistics commands

The `update statistics` commands create statistics, if there are none for a particular column, or replaces existing statistics. Statistics are stored in the `systabstats` and `sysstatistics` system tables:

```
update statistics table_name
[[ partition data_partition_name ] [ ( column_list ) ] |
index_name [ partition index_partition_name ] ]
[ using step values ]
[ with consumers = consumers ] [, sampling=percent]
```

```
update index statistics
table_name [[ partition data_partition_name ] |
[ index_name [ partition index_partition_name ] ] ]
[ using step values ]
[ with consumers = consumers ] [, sampling=percent]
```

```
update all statistics table_name
[ partition data_partition_name ]
[ sp_configure histogram tuning factor, <value>
```

```
update table statistics
```

table_name [partition *data_partition_name*]

delete [shared] statistics *table_name*
 [partition *data_partition_name*]
 [(*column_name* [, *column_name*] ...)]

- For update statistics:
 - *table_name* – generates statistics for the leading column in each index on the table.
 - *table_name index_name* – generates statistics for all columns of the index.
 - *partition_name* – generates statistics for only this partition.
 - *partition_name table_name (column_name)* – generates statistics for this column of this table on this partition.
 - *table_name (column_name)* – generates statistics for only this column.
 - *table_name (column_name, column_name...)* – generates a histogram for the leading column in the set, and multicolumn density values for the prefix subsets.
 - *using step values* – identifies the number of steps used. The default is 20 steps. To change the default number of steps, use *sp_configure*.
 - *sampling = percent* – the numeric value of the sampling percentage, such as 05 for 5%, 10 for 10%, and so on. The sampling integer is between zero (0) and one hundred (100).
- For update index statistics:
 - *table_name* – generates statistics for all columns in all indexes on the table.
 - *partition_name table_name* – generates statistics for all columns in all indexes for the table on this partition.
 - *table_name index_name* – generates statistics for all columns in this index.
- For update all statistics:
 - *table_name* – generates statistics for all columns of a table.
 - *table_name partition_name* – generates statistics for all columns of a table on a partition.

- *using step values* – identifies the number of steps used. The default is 20 steps. To change the default number of steps, use `sp_configure`.

`sp_configure` syntax includes the histogram tuning factor, which allows a greater selection of the number of histogram steps. The default value for histogram tuning factor is 20. See Chapter 5, “Setting Configuration Parameters” in *System Administration Guide: Volume I* for information about `sp_configure`.

Using sampling for *update statistics*

The optimizer for Adaptive Server uses the statistics on a database to set up and optimize queries. To generate optimal results, the statistics must be as current as possible.

Run the update statistics commands against data sets, such as tables, to update information about the distribution of key values in specified indexes or columns, for all columns in an index, or for all columns in a table. The commands revise histograms and density values for column-level statistics. The results are then used by the optimizer to calculate the best query plan.

Run update statistics using a sampling method, which can reduce the I/O and time when your maintenance window is small and the data set is large. If you are updating a large data set or table that is in constant use, being truncated and repopulated, you may want to do a statistical sampling to reduce the time and the size of the I/O. Because sampling does not update the density values, run a full update statistics prior to using sampling for an accurate density value.

Use caution with sampling, since the results are not fully accurate. Balance changes to histogram values against the savings in I/O.

Sampling does not update density values created by a non-sampling update statistics command. Since the density changes very slowly, replacing an accurate density with an approximation calculated by sampling usually does not improve the estimate. Density values created by a sampling update statistics command is updated. Sybase recommends that you use that one non-sampling update statistics command to establish an accurate density, which can be followed by numerous sampling update statistics commands. To have sampling update statistics update the density, delete the column statistics before using update statistics with sampling.

When you are deciding whether or not to use sampling, consider the size of the data set, the time constraints you are working with, and if the histogram produced is as accurate as needed.

The percentage to use when sampling depends on your needs. Test various percentages until you receive a result that reflects the most accurate information on a particular data set; for example:

```
update statistics authors(auth_id) with sampling = 5 percent
```

Set server-wide sampling percent using:

```
sp_configure 'sampling percent', 5
```

This command sets a server-wide sampling of 5% for update statistics that allows you to execute update statistics without the *sampling* syntax. The percentage can be between zero (0) and one hundred (100) percent.

Automatically updating statistics

The Adaptive Server cost-based query processor estimates the query costs using the statistics for the tables, indexes, and columns named in a query. The query processor chooses the access method it determines has the least cost. However, this cost estimate cannot be accurate if the statistics are not accurate. You can run update statistics to ensure that the statistics are current, however, running update statistics has an associated cost because it consumes system resources such as CPU, buffer pools, sort buffers, and procedure cache.

You can set update statistics to run automatically when it best suits your site and avoid running it at times that hamper your system. Use the `datachange` function to determine the best time for you to run update statistics. `datachange` also helps to ensure that you do not unnecessarily run update statistics. You can use the Job Scheduler templates to determine the objects, schedules, priority, and `datachange` thresholds that trigger update statistics, which ensures that critical resources are used only when the query processor generates more efficient plans.

Because update statistics is a resource-intensive task, base the decision to run update statistics on a specific set of criteria. Key parameters that can help you determine a good time to run update statistics include:

- How much data characteristics have changed since you last ran update statistics. This is known as the `datachange` parameter.
- Whether there are sufficient resources available to run update statistics. These include resources such as the number of idle CPU cycles and making sure that critical online activity does not occur during update statistics.

Data change is a key metric that helps you measure the amount of altered data since you last ran update statistics, and is tracked by the `datachange` function. Using this metric and the criteria for resource availability, you can automate the process of running update statistics. Job Scheduler includes a mechanism to automatically run update statistics, and also includes customizable templates you can use to determine when to run update statistics. These inputs include all parameters to update statistics, the `datachange` threshold values, and the time to run update statistics. Job Scheduler runs update statistics at a low priority so it does not affect critical jobs that are running concurrently.

***datachange* function**

The `datachange` function measures the amount of change in the data distribution since update statistics last ran. Specifically, it measures the number of inserts, updates, and deletes that have occurred on the given object, partition, or column, and helps you determine if running update statistics would benefit the query plan.

The syntax for `datachange` is:

```
select datachange(object_name, partition_name, colname)
```

Where:

- *object_name* – is the object name. This object is assumed to be in the current database. The *object_name* cannot be null.
- *partition_name* – is the data partition name. This can be a null value.
- *colname* – is the column name for which the `datachange` is requested. This can be a null value.

These parameters are all required.

`datachange` is expressed as a percentage of the total number of rows in the table or partition (if the partition is specified). The percentage value can be greater than 100 percent because the number of changes to an object can be much greater than the number of rows in the table, particularly when the number of deletes and updates to a table is very high.

The following set of examples illustrate the various uses for the `datachange` function. The examples use the following:

- Object name is “O.”
- Partition name is “P.”

- Column name is “C.”

Passing a valid object, partition, and column name

The value reported when you include the object, partition, and column name is determined by the `datachange` value for the specified column in the specified partition divided by the number of rows in the partition. The result is expressed as a percentage:

$$\text{datachange} = 100 * (\text{data change value for column C} / \text{rowcount (P)})$$

Using null partition names

If you include a null partition name, the `datachange` value is determined by the sum of the `datachange` value for the column across all partitions divided by the number of rows in the table. The result is expressed as a percentage:

$$\text{datachange} = 100 * (\text{Sum}(\text{data change value for (O, P(1-N), C)}) / \text{rowcount(O)})$$

Where $P(1-N)$ indicates that the value is summed over all partitions.

Using null column names

If you include null column names, the value reported by `datachange` is determined by the maximum value of the `datachange` for all columns that have histograms for the specified partition divided by the number of rows in the partition. The result is expressed as a percentage:

$$\text{datachange} = 100 * (\text{Max}(\text{data change value for (O, P, Ci)}) / \text{rowcount(P)})$$

Where the value of i varies through the columns with histograms (for example, formatid 102 in `sysstatistics`).

Null partition and column names

If you include null partition and column names, the value of `datachange` is determined by the maximum value of the `datachange` for all columns that have histograms summed across all partitions divided by the number of rows in the table. The result is expressed as a percentage:

$$\text{datachange} = 100 * (\text{Max}(\text{data change value for (O, NULL, Ci)}) / \text{rowcount(O)})$$

Where i is 1 through the total number of columns with histograms (for example, formatid 102 in `sysstatistics`).

This illustrates `datachange` gathering statistics:

```
create table matrix(col1 int, col2 int)
go
insert into matrix values (234, 560)
go
update statistics matrix(col1)
go
insert into matrix values(34,56)
go
select datachange ("matrix", NULL, NULL)
go
```

```
-----  
50.000000
```

The number of rows in matrix is two. The amount of data that has changed since the last update statistics command is 1, so the datachange percentage is $100 * 1/2 = 50$ percent.

datachange counters are all maintained in memory. These counters are periodically flushed to disk by the housekeeper or when you run `sp_flushstats`.

Configuring automatic *update statistics*

Automatically update statistics by:

- Defining update statistics jobs with Job Scheduler
- Defining update statistics jobs as part of the self-managed installation
- Creating user-defined scripts

Creating user-defined scripts is not discussed in this document.

Using Job Scheduler to update statistics

Job Scheduler includes the update statistics template, which you can use to create a job that runs update statistics on a table, index, column, or partition. The datachange function determines when the amount of change in a table or partition has reached the predefined threshold. You determine the value for this threshold when you configure the template.

Templates:

- Run update statistics on specific tables, partitions, indexes, or columns. The templates allow you to define the value for datachange at which you want update statistics to run.
- Run update statistics at the server level, which configures Adaptive Server to sweep through the available tables in all databases on the server and update statistics on all the tables, based on the threshold you determined when creating your job.

To configure Job Scheduler to automate the process of running update statistics (the chapters listed are from the *Job Scheduler Users Guide*:

- 1 Install and set up Job Scheduler (Chapter 2, “Configuring and Running Job Scheduler”)
- 2 Install the stored procedures required for the templates (Chapter 4, “Using Templates to Schedule Jobs”).
- 3 Install the templates. Job Scheduler provides the templates specifically for automating update statistics (Chapter 4, “Using Templates to Schedule Jobs”).
- 4 Configure the templates. The templates for automating update statistics are in the Statistics Management folder.
- 5 Schedule the job. After you have defined the index, column, or partition you want tracked, you can also create a schedule that determines when Adaptive Server runs the job, making sure that update statistics is run only when it does not impact performance.
- 6 Identify success or failure. The Job Scheduler infrastructure allows you to identify success or failure for the automated update statistic.

The template allows you to supply values for the various options of the update statistics command such as sampling percent, number of consumers, steps, and so on. Optionally, you can also provide threshold values for the datachange function, page count, and row count. If you include these optional values, they are used to determine when and if Adaptive Server should run update statistics. If the current values for any of the tables, columns, indexes, or partitions exceed the threshold values, Adaptive Server issues update statistics. Adaptive Server detects that update statistics has been run on a column. Any query referencing that table in the procedure cache is recompiled before the next execution.

When does Adaptive Server run *update statistics*?

There are many forms of the update statistics command (update statistics, update index statistics, and so on); use these different forms depending on your needs.

You must specify three thresholds: rowcount, pagecount, and datachange. Although values of NULL or 0 are ignored, these values do not prevent the command from running.

Table 10-1 describes the circumstances under which Adaptive Server automatically runs update statistics, based on the parameter values you provide.

Table 10-1: When does Adaptive Server automatically run update statistics?

If the user	Action taken by Job Scheduler
Specifies a datachange threshold of zero or NULL	Runs update statistics at the scheduled time.

If the user	Action taken by Job Scheduler
Specifies a datachange threshold greater than zero for a table only, and does not request the update index statistics form	Gets all the indexes for the table and gets the leading column for each index. If the datachange value for any leading column is greater than or equal to the threshold, run update statistics.
Specifies threshold values for the table and index but does not request the update index statistics form	Gets the datachange value for the leading column of the index. If the datachange value is greater than or equal to the threshold, runs update statistics.
Specifies a threshold value for a table only, and requests the update index statistics form	Gets all the indexes for the table and gets the leading column for each index. If the datachange value for any leading column exceeds the threshold, runs update statistics.
Specifies threshold values for table and index and requests the update index statistics form	Gets the datachange value for the leading column of the index. If the datachange value is greater than or equal to the threshold, runs update statistics.
Specifies threshold values for a table and one or more columns (ignores any indexes or requests for the update index statistics form)	Gets the datachange value for each column. If the datachange value for any column is greater than or equal to the threshold, runs update statistics.

The datachange function compiles the number of changes in a table and displays this as a percentage of the total number of rows in the table. You can use this compiled information to create rules that determine when Adaptive Server runs update statistics. The best time for this to happen can be based on any number of objectives:

- The percentage of change in a table
- Number of CPU cycles available
- During a maintenance window

After update statistics runs, the datachange counter is reset to zero. The count for datachange is tracked at the partition level (not the object level) for inserts, and deletes and at the column level for updates.

Examples of updating statistics with *datachange*

You can write scripts that check for the specified amount of changed data at the column, table, or partition level. The time at which you decide to run update statistics can be based on a number of variables collected by the datachange function; CPU usage, percent change in a table, percent change in a partition, and so on.

In this example, the authors table is partitioned, and the user wants to run update statistics when the data changes to the city column in the author_ptn2 partition are greater than or equal to 50%:

```
select @datachange = datachange("authors","author_ptn2", "city")
if @datachange >= 50
begin
    update statistics authors partition author_ptn2(city)
end
go
```

The user can also specify that the script is executed when the system is idle or any other parameters.

In this example, the user triggers update statistics when the data changes to the city column of the authors table are greater than or equal to 100% (the table in this example is not partitioned):

```
select @datachange = datachange("authors",NULL, "city")
if @datachange > 100
begin
    update statistics authors (city)
end
go
```

Column statistics and statistics maintenance

Histograms are kept on a per-column basis, rather than on a per-index basis. This has certain implications for managing statistics:

- If a column appears in more than one index, update statistics, update index statistics, or create index updates the histogram for the column and the density statistics for all prefix subsets.

update all statistics updates histograms for all columns in a table.

- Dropping an index does not drop the statistics for the index, since the optimizer can use column-level statistics to estimate costs, even when no index exists.

To remove the statistics after dropping an index, you must explicitly delete them using delete statistics.

If the statistics are useful to the query processor, and to keep the statistics without having an index, use `update statistics`, specifying the column name, for indexes where the distribution of key values changes over time.

- Truncating a table does not delete the column-level statistics in `sysstatistics`. In many cases, tables are truncated and the same data is reloaded.

Since `truncate table` does not delete the column-level statistics, you need not run `update statistics` after the table is reloaded, if the data is the same.

If you reload the table with data that has a different distribution of key values, run `update statistics`.

- You can drop and re-create indexes without affecting the index statistics, by specifying 0 for the number of steps in the `with statistics` clause to create index. This `create index` command does not affect the statistics in `sysstatistics`:

```
create index title_id_ix on titles(title_id)
with statistics using 0 values
```

This allows you to re-create an index without overwriting statistics that have been edited with `optdiag`.

- If two users attempt to create an index on the same table, with the same columns, at the same time, one of the commands may fail due to an attempt to enter a duplicate key value in `sysstatistics`.
- Executing `update statistics` on a column in a partition of a multipartition table updates the statistics for that partition, but also updates the global histogram for that column. This is done by merging the histograms for that column from each partition in a row-weighted fashion to arrive at a global histogram for the column.
- Updating statistics on a multipartitioned table for a column, without specifying a partition, updates the statistics for each partition of the table for that column, and, as a last step, merges the partition histograms for the column to create a global histogram for the column.
- The optimizer only uses the global histograms for a multipartitioned table during compilation, and does not read the partition histograms. This approach avoids the overhead of merging partition histograms at compilation time, and instead performs any merging work at DDL time.

Creating and updating column statistics

Creating statistics on unindexed columns can improve the performance of many queries. The optimizer can use statistics on any column in a *where* or *having* clause to help estimate the number of rows from a table that match the complete set of query clauses on that table.

Adding statistics for the minor columns of indexes and for unindexed columns that are frequently used in search arguments can greatly improve the optimizer's estimates.

Maintaining a large number of indexes during data modification can be expensive. Every index for a table must be updated for each insert and delete to the table, and updates can affect one or more indexes.

Generating statistics for a column without creating an index gives the optimizer more information to use for estimating the number of pages to be read by a query, without the processing expense of index updates during data modification.

The optimizer can apply statistics for any columns used in a search argument of a *where* or *having* clause, and for any column named in a join clause.

Use these commands to create and maintain column statistics:

- *update statistics*, when used with the name of a column, generates statistics for that column without creating an index on it. See “Adding statistics for a column with *update statistics*” on page 305 for information about syntax.

The optimizer can use these column statistics to more precisely estimate the cost of queries that reference the column.

- *update index statistics*, when used with an index name, creates or updates statistics for all columns in an index. See “Adding statistics for minor columns with *update index statistics*” on page 305 for information about syntax.

If used with a table name, *update index statistics* updates statistics for all indexed columns.

- *update all statistics* creates or updates statistics for all columns in a table. See “Adding statistics for all columns with *update all statistics*” on page 306 for information about syntax.

Good candidates for column statistics are:

- Columns frequently used as search arguments in *where* and *having* clauses

- Columns included in a composite index, and which are not the leading columns in the index, but which can help estimate the number of data rows that need to be returned by a query

When additional statistics may be useful

To determine when additional statistics are useful, run queries using `set option` commands and `set statistics io`. If there are significant discrepancies between the “rows to be returned” and I/O estimates displayed by `set` commands and the actual I/O displayed by `statistics io`, examine these queries for places where additional statistics can improve the estimates. Look especially for the use of default density values for search arguments and join columns.

The `set option show_missing_stats` command prints the names of columns that could have used histograms, and groups of columns that could have used multiattribute densities. This is particularly useful in pointing out where additional statistics can be useful.

Example 1

```
set option show_missing_stats long
go
dbcc traceon(3604)
go
DBCC execution completed. If DBCC printed error
messages, contact a user with System Administrator (SA)
role.
select * from part, partsupp
where p_partkey = ps_partkey and p_itemtype =
ps_itemtype
go
NO STATS on column part.p_partkey
NO STATS on column part.p_itemtype
NO STATS on column partsupp.pa_itemtype
NO STATS on density set for E={p_partkey, p_itemtype}
NO STATS on density set for F={ps_partkey, ps_itemtype}
- - - - -
(200 rows affected)
```

You can get the same information using the `show_final_plan_xml` option. The `set plan` uses the `client option` and `trace flag 3604` to get the output on the client side. This differs from the way the `message option` of `set plan` is used.

Example 2

```
dbcc traceon(3604)
DBCC execution completed. If DBCC printed error
messages, contact a user with System Administrator (SA)
role.
```

```

set plan for show_final_plan_xml to client on
go
select * from part, partsupp
where p_partkey = ps_partkey and p_itemtype =
ps_itemtype
go
<?xml version="1.0" encoding="UTF-8"?>
<query>
    <planVersion> 1.0 </planVersion>
    - - - - -
<optimizerStatistics>
    <statInfo>
        <objName>part</objName>
        <missingHistogram>
            <column>p_partkey</column>
            <column>p_itemtype</column>
        </missingHistogram>
        <missingDensity>
            <column>p_partkey</column>
            <column>p_itemtype</column>
        </missingDensity>
    </statInfo>
    <statInfo>
        <objName>partsupp</objName>
        <missingHistogram>
            <column>ps_partkey</column>
            <column>ps_itemtype</column>
        </missingHistogram>
        <missingDensity>
            <column>ps_partkey</column>
            <column>ps_itemtype</column>
        </missingDensity>
    </statInfo>
</optimizerStatistics>

```

Use update statistics on part and partsupp to create statistics on p_partkey and p_itemtype, thus creating a histogram on the leading column (p_partkey) and the density (p_partkey, p_itemtype). Create a histogram on p_itemtype as well. Use:

```

update statistics part(p_partkey, p_itemtype)
go
update statistics part(p_itemtype)
go

```

Since `partsupp` has a histogram on `ps_partkey`, you can create a histogram on `ps_itemtype` and a density on `(ps_itemtype, ps_partkey)`. The columns used for density may be unordered.

```
update statistics partsupp (ps_itemtype, ps_partkey)
```

If this procedure is successful, you will not see the “NO STATS” messages shown in Example 1 when you run the query again.

Adding statistics for a column with *update statistics*

To add statistics for the price column in the titles table, enter:

```
update statistics titles (price)
```

To specify the number of histogram steps for a column, use:

```
update statistics titles (price)
using 50 values
```

This command adds a histogram for the `titles.pub_id` column and generates density values for the prefix subsets `pub_id`; `pub_id, pubdate`; and `pub_id, pubdate, title_id`:

```
update statistics titles (pub_id, pubdate, title_id)
```

However, this command does not create a histogram on `pubdate` and `title_id`, since a separate `update statistics` command is needed for every column for which a histogram is desired.

Note Running `update statistics` with a table name updates histograms and densities for leading columns for indexes only; it does not update the statistics for unindexed columns. To maintain these statistics, run `update statistics` and specify the column name, or run `update all statistics`.

Adding statistics for minor columns with *update index statistics*

To create or update statistics on all columns in an index, use `update index statistics`. The syntax is:

```
update index statistics
table_name [[ partition data_partition_name ] |
[ index_name [ partition index_partition_name ] ] ]
```

```
[ using step values ]  
[ with consumers = consumers ] [, sampling = percent]
```

Adding statistics for all columns with *update all statistics*

To create or update statistics on all columns in a table, use *update all statistics*. The syntax is:

```
update all statistics table_name  
[partition data_partition_name]
```

Choosing step numbers for histograms

By default, each histogram has 20 steps, which provides good performance and modeling for columns that have an even distribution of values. A higher number of steps can increase the accuracy of I/O estimates for columns that:

- Have a large number of highly duplicated values
- Have unequal or skewed distribution of values
- Are queried using leading wildcards in like queries

The histogram tuning factor default of 20 automatically chooses a step value between the current requested step value (default 20) and the increased steps due to the factor ($20 * 20 = 400$) so that Adaptive Server automatically chooses the optimal steps value to compensate for the above cases. Overriding the step values should take into account the larger number of steps already introduced by the histogram tuning factor.

Note If your database was updated from a pre-11.9 version of Adaptive Server, the number of steps defaults to the number of steps that were used on the distribution page.

Increasing the number of steps beyond what is needed for good query optimization can degrade Adaptive Server performance, largely due to the amount of space that is required to store and use the statistics. Increasing the number of steps:

- Increases the disk storage space required for sysstatistics

- Increases the cache space needed to read statistics during query optimization
- Requires more I/O, if the number of steps is very large

During query optimization, histograms use space borrowed from the procedure cache. This space is released as soon as the query is optimized.

Choosing a step number

If your table has 5000 rows, and one value in the column that has only one matching row, you may need to request 5000 steps to get a histogram that includes a frequency cell for every distinct value. The actual number of steps is not 5000; it is either the number of distinct values plus one (for dense frequency cells), or twice the number of values plus one (for sparse frequency cells).

The `sp_configure` option histogram tuning factor automatically chooses a larger number of steps, within parameters, when there are a large number of highly duplicated values.

The default value of the histogram tuning factor is 20 in Adaptive Server version 15.0 and later. If the requested step count is 50, then update statistics can create up to $20 * 50 = 1000$ steps. This larger number of steps is used only if histogram distribution is skewed with a number of domain values that are highly duplicated. However, for a unique column, update statistics uses only 50 steps to represent the histogram. To most efficiently use histograms, specify a relatively low number of steps and allow the histogram tuning factor to determine whether more steps would be useful for optimization. For example, instead of specifying 1000 steps with a default step count of 1000 to be used by all histograms, it is better to specify 50 default steps and a histogram tuning factor of 20. This allows Adaptive Server to determine the best step count, within the range of 50 to 1000 steps, with which to represent the distribution.

Scan types, sort requirements, and locking

Table 10-2 shows the types of scans performed during update statistics, the types of locks acquired, and when sorts are needed.

Table 10-2: Scans, sorts, and locking during update statistics

update statistics specifying	Scans and sorts performed	Locking
<i>Table name</i>		
Allpages-locked table	Table scan, plus a leaf-level scan of each nonclustered index	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan, plus a leaf-level scan of each nonclustered index and the clustered index, if one exists	Level 0; dirty reads
<i>Table name and clustered index name</i>		
Allpages-locked table	Table scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
<i>Table name and nonclustered index name</i>		
Allpages-locked table	Leaf level index scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
<i>Table name and column name</i>		
Allpages-locked table	Table scan; creates a worktable and sorts the worktable	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan; creates a worktable and sorts the worktable	Level 0; dirty reads

Sorts for unindexed or nonleading columns

For unindexed columns and columns that are not the leading columns in indexes, Adaptive Server performs a serial table scan, copying the column values into a worktable. It then sorts the worktable to build the histogram. The sort is performed in serial, unless the `with consumers` clause is specified.

See Chapter 5, “Parallel Query Processing,”.

Locking, scans, and sorts during *update index statistics*

The `update index statistics` command generates a series of update statistics operations that use the same locking, scanning, and sorting as the equivalent index-level and column-level command. For example, if the `salesdetail` table has a nonclustered index named `sales_det_ix` on `salesdetail(stor_id, ord_num, title_id)`, this command:

```
update index statistics salesdetail
```

performs these update statistics operations:

```
update statistics salesdetail sales_det_ix  
update statistics salesdetail (ord_num)  
update statistics salesdetail (title_id)
```

Locking, scans and sorts during *update all statistics*

The update all statistics commands generate a series of update statistics operations for each index on the table, followed by a series of update statistics operations for all unindexed columns.

Using the *with consumers* clause

The with consumers clause for update statistics is designed for use on partitioned tables on Redundant Array of Independent Disks (RAID) devices, which appear to Adaptive Server as a single I/O device, but can produce the high throughput required for parallel sorting. See Chapter 5, “Parallel Query Processing,” for information.

Reducing the impact of *update statistics* on concurrent processes

Since update statistics uses dirty reads (transaction isolation level 0) for data-only-locked tables, you can execute it while other tasks are active on the server; it does not block access to tables and indexes. Updating statistics for leading columns in indexes requires only a leaf-level scan of the index, and does not require a sort, so updating statistics for these columns does not affect concurrent performance very much.

However, updating statistics for unindexed and nonleading columns, which require a table scan, worktable, and sort, can affect concurrent processing.

- Sorts are CPU-intensive. Use a serial sort, or a small number of worker processes to minimize CPU utilization. Alternatively, you can use execution classes to set the priority for update statistics.

See the *Performance and Tuning Series: Basics*.

- The cache space required for merging sort runs is taken from the data cache, and some procedure cache space is also required. Setting the number of sort buffers to a low value reduces the space used in the buffer cache.

If number of sort buffers is set to a large value, it takes more space from the data cache, and may also cause stored procedures to be flushed from the procedure cache, since procedure cache space is used while merging sorted values. There are approximately 100 bytes of procedure cache needed for every row that can fit into the sort buffers specified. For example, if 500 2K sort buffers are specified, and about 200 rows fit into each 2K buffer, then $200 * 100 * 500$ bytes of procedure cache are needed to support the sort. This example requires about 5000 2K procedure cache buffers, if the entire 500 data cache buffers are filled by a sort run.

Creating the worktables for sorts also uses space in tempdb.

Using the *delete statistics* command

In versions of Adaptive Server earlier than 11.9, dropping an index removed the distribution page for the index. As of version 11.9.2, maintaining column-level statistics is under explicit user control, and the optimizer can use column-level statistics even when an index does not exist. The *delete statistics* command allows you to drop statistics for specific columns.

If you create an index and then decide to drop it because it is not useful for data access, or because of the cost of index maintenance during data modifications, you must determine whether the:

- Statistics on the index are useful to the optimizer.
- Distribution of key values in the columns for this index are subject to change over time as rows are inserted and deleted.

If the distribution of key values changes, run update statistics periodically to maintain useful statistics.

This example deletes the statistics for the price column in the titles table:


```
delete statistics titles(price)
```

Note `delete statistics` removes rows only from `sysstatistics`; it does not remove rows from `systabstats`. You cannot delete the rows in `systabstats` that described partition row counts, cluster ratios, page counts, and so on. However, if you use `optdiag` simulate statistics to add any simulated `systabstats` rows to `sysstatistics`, then those rows are deleted.

When row counts may be inaccurate

Row count values for the number of rows, number of forwarded rows, and number of deleted rows may be inaccurate, especially if query processing includes many rollback commands. If workloads are extremely heavy, and the housekeeper wash task does not run often, these statistics are more likely to be inaccurate.

Running `update statistics` corrects counts in `systabstats`. When the housekeeper wash task runs, or when you execute `sp_flushstats`, row count values are saved in `systabstats`.

Note You must set the configuration parameter `housekeeper free write percent` to 1 or greater to enable housekeeper statistics flushing.

Running `dbcc checktable` or `dbcc checkdb` updates these row count values in memory.

Topic	Page
Overview	313
Managing abstract plans	314
Relationship between query text and query plans	315
Full versus partial plans	316
Abstract plan groups	318
How abstract plans are associated with queries	318

Overview

Adaptive Server can generate an abstract plan for a query, and save the text and its associated abstract plan in the sysqueryplans system table. Using a rapid hashing method, incoming SQL queries can be compared to saved query text, and if a match is found, the corresponding saved abstract plan is used to execute the query.

An abstract plan describes the execution plan for a query using a language created for that purpose. This language contains operators to specify the choices and actions that can be generated by the optimizer. For example, to specify an index scan on the titles table, using the index title_id_ix, the abstract plan says:

```
(i_scan title_id_ix titles)
```

To use this abstract plan with a query, you can modify the query text and add a `PLAN` clause:

```
select * from titles where title_id = "On Liberty"
plan
"(i_scan title_id_ix titles)"
```

This alternative requires a change to the SQL text; however, the method described in the first paragraph, that is, the sysqueryplans-based way to give the abstract plan of a query, does not involve changing the query text.

Abstract plans provide a means for system administrators and performance tuners to protect the overall performance of a server from changes to query plans. Changes in query plans can arise due to:

- Adaptive Server software upgrades that affect optimizer choices and query plans
- New Adaptive Server features that change query plans
- Changing tuning options such as the parallel degree, table partitioning, or indexing

The main purpose of abstract plans is to provide a means to capture query plans before and after major system changes. You can then compare sets of before-and-after query plans to determine the effects of changes on your queries. Other uses include:

- Searching for specific types of plans, such as table scans or reformatting
- Searching for plans that use particular indexes
- Specifying full or partial plans for poorly performing queries
- Saving plans for queries with long optimization times

Abstract plans provide an alternative to options that must be specified in the batch or query to influence optimizer decisions. Using abstract plans, you can influence the optimization of a SQL statement without modifying the statement syntax. While matching query text to stored text requires some processing overhead, using a saved plan reduces query optimization overhead.

Managing abstract plans

A full set of system procedures allows system administrators and database owners to administer plans and plan groups. Individual users can view, drop, and copy the plans for the queries that they have run.

See Chapter 14, “Managing Abstract Plans with System Procedures,” for more information.

Relationship between query text and query plans

For most SQL queries, there are many possible query execution plans. SQL describes the desired result set, but does not describe how that result set should be obtained from the database. Consider a query that joins three tables, such as:

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

There are many different possible join orders, and depending on the indexes that exist on the tables, many possible access methods, including table scans, index scans, and the reformatting strategy. Each join may use either a nested-loop join or a merge join. These choices are determined by the optimizer's query costing algorithms, and are not included in or specified in the query itself.

When you capture the abstract plan, the query is optimized in the usual way, except that the optimizer also generates an abstract plan, and saves the query text and abstract plan in `sysqueryplans`.

Limits of options for influencing query plans

Adaptive Server provides other options for influencing optimizer choices:

- Session-level options such as `set forceplan` to force join order or `set parallel_degree` to specify the maximum number of worker processes to use for the query
- Options that can be included in the query text to influence the index choice, cache strategy, and parallel degree

There are some limitations to using set commands or adding hints to the query text:

- Not all query plan steps can be influenced, for example, subquery attachment.
- Some query-generating tools do not support the in-query options or require all queries to be vendor-independent.

Full versus partial plans

Abstract plans can be full plans, describing all query processing steps and options, or they can be partial plans. A partial plan might specify that an index is to be used for the scan of a particular table, without specifying other access methods. For example:

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
plan
"(i_scan t3_c31_ix t3)"
```

The full abstract plan includes:

- The join type, `nl_join` for nested-loop joins, `m_join` for merge joins, or `h_join` for hash joins.
- The join order.
- The type of scan, `t_scan` for table scan or `i_scan` for index scan.
- The name of the index chosen for the tables that are accessed via an index scan.
- The scan properties: the parallel degree, I/O size, and cache strategy for each table in the query.

The abstract plan for the query above specifies the join order, the access method for each table in the query, and the scan properties for each table:

```
select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31
plan
"(i_scan t3_c31_ix t3)"

(nl_join ( nl_join
  ( t_scan t2 )
  ( i_scan t1_c11_ix t1 )
)
  ( i_scan t3_c31_ix t3 )
)
( prop t3
  ( parallel 1 )
  ( prefetch 16 )
  ( lru )
)
```

```
)  
( prop t1  
  ( parallel 1 )  
  ( prefetch 16 )  
  ( lru )  
)  
( prop t2  
  ( parallel 1 )  
  ( prefetch 16 )  
  ( lru )  
)
```

If the abstract plan dump mode is on, the query text and the abstract plan pair are saved in `sysqueryplans`:

```
select t1.c11, t2.c21  
from t1, t2, t3  
where t1.c11 = t2.c21  
and t1.c11 = t3.c31  
plan  
"(i_scan t3_c31_ix t3)"
```

Creating a partial plan

When abstract plans are captured, full abstract plans are generated and stored. You can write partial plans that affect only a subset of the optimizer choices. If the query above had not used the index on `t3`, but all other parts of the query plan were optimal, you could create a partial plan for the query using the `create plan` command. This partial plan specifies only the index choice for `t3`:

```
create plan  
"select t1.c11, t2.c21  
from t1, t2, t3  
where t1.c11 = t2.c21  
and t1.c11 = t3.c31"  
"( i_scan t3_c31_ix t3 )"
```

You can also create abstract plans with the `plan` clause for `select`, `delete`, `update`, and other commands that can be optimized. If the abstract plan dump mode is on, the query text and AP pair are saved in `sysqueryplans`.

See Chapter 12, “Creating and Using Abstract Plans,” for more information.

Abstract plan groups

When you install Adaptive Server, there are two abstract plan groups:

- `ap_stdout`, used by default to capture plans
- `ap_stdin`, used by default for plan association

A system administrator can enable server-wide plan capture to `ap_stdout`, so that all query plans for all queries are captured. Server-wide plan association uses queries and plans from `ap_stdin`. If some queries require specially-tuned plans, they can be made available server-wide.

A system administrator or database owner can create additional plan groups, copy plans from one group to another, and compare plans in two different groups.

The capture of abstract plans and the association of abstract plans with queries always happens within the context of the currently active plan group. Users can use session-level set commands to enable plan capture and association.

Some of the ways abstract plan groups can be used are:

- A query tuner can create abstract plans in a group created for testing purposes without affecting plans for other users on the system
- Using plan groups, “before” and “after” sets of plans can be used to determine the effects of system or upgrade changes on query optimization.

See Chapter 12, “Creating and Using Abstract Plans,” for more information on enabling the capture and association of plans.

How abstract plans are associated with queries

When an abstract plan is saved, all white space (tabs, multiple spaces, and returns, except for returns that terminate a `--`-style comment) in the query is trimmed to a single space, and a hash-key value is computed for the white-space trimmed SQL statement. The trimmed SQL statement and the hash key are stored in `sysqueryplans` along with the abstract plan, a unique plan ID, the user’s ID, and the ID of the current abstract plan group.

When abstract plan association is enabled, the hash key for incoming SQL statements is computed, and this value is used to search for the matching query and abstract plan in the current association group, with the corresponding user ID. The full association key of an abstract plans consists of:

- The user ID of the current user
- The group ID of the current association group
- The full query text

Once a matching hash key is found, the full text of the saved query is compared to the query to be executed, and used if it matches.

The association key combination of user ID, group ID, and query text means that for a given user, there cannot be two queries in the same abstract plan group that have the same query text, but different query plans.

Abstract plans in cached statements

In Adaptive Server version 15.7 and later, you can save abstract plan information in the statement cache.

In this example, which includes an abstract plan, the hash table saves `select * from t1 plan '(use optgoal allrows_mix)'`, as shown in the SQL TEXT line:

```
1> select * from t1 plan '(use optgoal allrows_mix)'
2> go
1> dbcc prsqlcache
2> go
```

Start of SSQL Hash Table at 0x0x1474c9050

Memory configured: 1000 2k pages Memory used: 17 2k pages

Bucket# 243 address 0x0x1474c9f80

```
SSQL_DESC 0x0x1474cd070
ssql_name *ss0626156152_0290084701ss*
ssql_hashkey 0x0x114a575d                      ssql_id 626156152
ssql_suid 1                      ssql_uid 1                      ssql_dbid 1                      ssql_spid 0
ssql_status 0x0xa0                      ssql_parallel_deg 1
```

```
ssql_isolate 1          ssql_tranmode 32
ssql_keep 0             ssql_usecnt 1    ssql_pgcount 6
ssql_optgoal allrows_mix ssql_optlevel ase_default
SQL TEXT: select * from t1 plan '(use optgoal allrows_mix)'
```

End of SSQL Hash Table

Creating and Using Abstract Plans

Use the `set` command to capture abstract plans and to associate incoming SQL queries with saved plans. Any user can issue session-level commands to capture and load plans during a session, and a system administrator can enable server-wide abstract plan capture and association. This chapter also describes how to specify abstract plans using SQL.

Topic	Page
Using set commands to capture and associate plans	321
set plan exists check option	327
Using other set options with abstract plans	328
Server-wide abstract plan capture and association modes	330
Creating plans using SQL	331

Using *set* commands to capture and associate plans

At the session level, any user can enable and disable capture and use of abstract plans using the `set plan dump` and `set plan load` commands. `set plan replace` determines whether existing plans are overwritten by changed plans.

Enabling and disabling abstract plan modes takes effect at the end of the batch in which the command is included (similar to `showplan`). Therefore, change the mode in a separate batch before you run your queries:

```
set plan dump on
go
/*queries to run*/
go
```

Any set plan commands used in a stored procedure do not affect the procedure (except those statements affected by deferred compilation) in which they are included, but remain in effect after the procedure completes.

Enabling plan capture mode with *set plan dump*

The set plan dump command activates and deactivates the capture of abstract plans. You can save the plans to the default group, ap_stdout, by using set plan dump with no group name:

```
set plan dump on
```

To start capturing plans in a specific abstract plan group, specify the group name. This example sets the group dev_plans as the capture group:

```
set plan dump dev_plans on
```

The group that you specify must exist before you issue the set command. The system procedure sp_add_qpgroup creates abstract plan groups; only the system administrator or database owner can create an abstract plan group. Once an abstract plan group exists, any user can dump plans to the group.

See “Creating a group” on page 373 for information on creating a plan group.

To deactivate the capturing of plans, use:

```
set plan dump off
```

You do not need to specify a group name to end capture mode. Only one abstract plan group can be active for saving or matching abstract plans at any one time. If you are currently saving plans to a group, turn off the plan dump mode, and reenale it for the new group, as shown here:

```
set plan dump on /*save to the default group*/
go
/*some queries to be captured */
go
set plan dump off
go
set plan dump dev_plans on
go
/*additional queries*/
go
```

The use of the `use database` command while `set plan dump` is in effect disables plan dump mode.

Associating queries with stored plans

The `set plan load` command activates and deactivates the association of queries with stored abstract plans.

To start the association mode using the default group, `ap_stdin`, use:

```
set plan load on
```

To enable association mode using another abstract plan group, specify the group name:

```
set plan load test_plans on
```

Only one abstract plan group can be active for plan association at one time. If plan association is active for a group, deactivate the current group and activate plan association for the new group, as shown here:

```
set plan load test_plans on
go
/*some queries*/
go
set plan load off
go
set plan load dev_plans on
go
```

The use of the `use database` command while `set plan load` is in effect disables plan load mode.

Using replace mode during plan capture

While plan capture mode is active, you can choose whether to have plans for the same query replace existing plans by enabling or disabling `set plan replace`. To activate plan replacement mode, use:

```
set plan replace on
```

Do not specify a group name with `set plan replace`; it affects the current active capture group.

To disable plan replacement:

```
set plan replace off
```

The use of the use database command while set plan replace is in effect disables plan replace mode.

When to use replace mode

When you are capturing plans, and a query has the same query text as an already-saved plan, the existing plan is not replaced unless replace mode is enabled. If you have captured abstract plans for specific queries, and you are making physical changes to the database that affect optimizer choices, replace existing plans for these changes to be saved.

Some actions that might require plan replacement are:

- Adding or dropping indexes, or changing keys or key ordering in indexes
- Changing the partitioning on a table
- Adding or removing buffer pools
- Changing configuration parameters that affect query plans

In most cases, do not enable plan load. When plan association is active, any plan specifications are used as inputs to the optimizer. For example, if a full query plan includes the prefetch property and an I/O size of 2K, and you have created a 16K pool and want to replace the prefetch specification in the plan, do not enable plan load mode.

You may want to check query plans and replace some abstract plans as data distribution changes in tables, or after rebuilds on indexes, updating statistics, or changing the locking scheme.

Using *dump*, *load*, and *replace* modes simultaneously

You can have both plan dump and plan load mode active simultaneously, with or without replace mode active.

Using *dump* and *load* to the same group

If you have enabled dump and load to the same group, without replace mode enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If a plan exists that is not valid (for example, because an index has been dropped), a new plan is generated and used to optimize the query, but is not saved.
- If only a partial plan exists, a full plan is generated, but the existing partial plan is not replaced
- If a plan does not exist for the query, a plan is generated and saved.

With replace mode also enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If the plan is not valid, a new plan is generated and used to optimize the query, and the old plan is replaced.
- If the plan is a partial plan, a complete plan is generated and used, and the existing partial plan is replaced. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query, a plan is generated and saved.

Using *dump* and *load* to different groups

If you have dump enabled to one group, and load enabled from another group, without replace mode enabled:

- If a valid plan exists for the query in the load group, it is loaded and used. The plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is not valid, a new plan is generated. The new plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group, unless a plan already exists. The specifications in the partial plan are used as input to the optimizer.
- If there is no plan for the query in the load group, the plan is generated and saved in the dump group, unless a plan for the query exists in the dump group.

With replace mode active:

- If a valid plan exists for the query in the load group, it is loaded and used.
- If the plan in the load group is not valid, a new plan is generated and used to optimize the query. The new plan is saved in the dump group.
- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query in the load group, a new plan is generated. The new plan is saved in the dump group.

Compile-time changes for some set parameters

In Adaptive Server releases earlier than 15.0.2, the set parameters took effect after the stored procedure was executed or recompiled. Adaptive Server release 15.0.2 and later allows you to use optimizer set parameters at compile time to affect the optimizer in stored procedures or batches.

Note This changed behavior may effect the composition of the result set. Sybase recommends that you review the result set created by the 15.0.2 versions of the set parameters before using them in your production systems.

You must reset the set parameter before returning from the stored procedure or the execution of subsequent stored procedures may be affected. If you intend to propagate this change to subsequent stored procedures, use `export_options` parameter.

Adaptive Server changes the compile-time behavior for these parameters:

- `distinct_sorted`
- `distinct_sorting`
- `distinct_hashing`
- `group_sorted`
- `group_hashing`
- `bushy_space_search`
- `parallel_query`

- `order_sorting`
- `nl_join`
- `merge_join`
- `hash_join`
- `append_union_all`
- `merge_union_all`
- `merge_union_distinct`
- `hash_union_distinct`
- `store_index`
- `index_intersection`
- `index_union`
- `multi_table_store_ind`
- `opportunistic_distict_view`
- `advanced_aggregation`
- `replicated_partition`
- `group_inserting`
- `basic_optimization`
- `auto_query_tuning`
- `query_tuning_mem_limit`
- `query_tuning_time_limit`
- `set plan optgoal`

set plan exists check option

Use the `exists check` mode during query plan association to speed performance when users require abstract plans for fewer than 20 queries from an abstract plan group. If a small number of queries require plans to improve their optimization, enabling `exists check` mode speeds execution of all queries that do not have abstract plans, because they do not check for plans in `sysqueryplans`.

When set plan load and set exists check are both enabled, the hash keys for up to 20 queries in the load group are cached for the user. If the load group contains more than 20 queries, exists check mode is disabled. Each incoming query is hashed; if its hash key is not stored in the abstract plan cache, then there is no plan for the query and no search is made. This speeds the compilation of all queries that do not have saved plans.

The syntax is:

```
set plan exists check {on | off}
```

You must enable load mode before you enable plan hash-key caching.

A system administrator can configure server-wide plan hash-key caching with the configuration parameter abstract plan cache. To enable server-wide plan caching, use:

```
sp_configure "abstract plan cache", 1
```

Using other set options with abstract plans

You can combine other set tuning options with set plan dump, show_abstract_plan, and set plan load.

Using show_abstract_plan to view plans

set option show_abstract_plan prints the optimal abstract plan currently running on the TDS connection. It prints the plan after optimization and before execution, and is the only set option show_ command that does not depend on trace flag 3604 or 3605.

Printing the final plan's abstract plan is similar to viewing showplan output: it provides information to the user, but the abstract plan is not saved in sysqueryplans, and is not used if you use the abstract plan load mode.

This example shows the optimal abstract plan currently running on the TDS connection:

```
1> set option show_abstract_plan on
2> go
1> select r1, sum(s1)
2> from r, s
```

```

3> where r2=s2
4> group by r1
The Abstract Plan (AP) of the final query execution plan:
( group_sorted ( nl_join ( i_scan ir12 r ) ( i_scan is21 s ) ) ) ( prop r
(parallel 1 ) ( prefetch 2 ) ( lru ) ) ( prop s ( parallel 1 ) ( prefetch 2 )
(lru ) )
To experiment with the optimizer behavior, this AP can be modified and then
passed to the optimizer using the PLAN clause:
SELECT/INSERT/DELETE/UPDATE ...
PLAN '( ... )'.
r1
-----
          1          2
          2          4

(2 rows affected)

```

Using *showplan*

When *showplan* is turned on, and abstract plan association mode has been enabled with *set plan load*, *showplan* prints the plan ID of the matching abstract plan at the beginning of the *showplan* output for the statement:

```

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using an Abstract Plan (ID : 832005995).

```

If you run queries using the plan clause added to a SQL statement, *showplan* displays:

```

Optimized using the Abstract Plan in the PLAN clause.

```

Using *noexec*

You can use *noexec* mode to capture abstract plans without actually executing the queries. If *noexec* mode is in effect, queries are optimized and abstract plans are saved, but no query results are returned.

To use *noexec* mode while capturing abstract plans, execute any needed procedures (such as *sp_add_qpgroup*) and other set options (such as *set plan dump*) before enabling *noexec* mode. The following example shows a typical set of steps:

```

sp_add_qpgroup pubs_dev
go

```

```
set plan dump pubs_dev on
go
set noexec on
go
select type, sum(price) from titles group by type
go
```

Using *fmtonly*

A similar behavior can be obtained for capturing plans in stored procedures without actually executing the stored procedures, using *fmtonly* set.

```
sp_add_qpgroup pubs_dev
go
set plan dump pubs_dev on
go
set fmtonly on
go
exec stored_proc(...)
go
```

Using *forceplan*

If set *forceplan* on is in effect, and query association is also enabled for the session, *forceplan* is ignored if a full abstract plan is used to optimize the query. If a partial plan does not completely specify the join order:

- First, the tables in the abstract plan are ordered, as specified.
- The remaining tables are ordered as specified in the from clause.
- The two lists of tables are merged.

Server-wide abstract plan capture and association modes

A system administrator can enable server-wide plan capture, association, and replacement modes with these configuration parameters:

- **abstract plan dump** – enables dumping to the default abstract plans capture group, `ap_stdout`.
- **abstract plan load** – enables loading from the default abstract plans loading group, `ap_stdin`.
- **abstract plan replace** – when plan dump mode is also enabled, enables plan replacement.
- **abstract plan cache** – enables caching of abstract plan hash IDs; abstract plan load must also be enabled. See “set plan exists check option” on page 327.

By default, these configuration parameters are set to 0, which means that capture and association modes are off. To enable a mode, set the configuration value to 1:

```
sp_configure "abstract plan dump", 1
```

Enabling any of the server-wide abstract plan modes is dynamic; you need not restart the server.

Server-wide capture and association allows the system administrator to capture all plans for all users on a server. You cannot override server-wide modes at the session level.

Creating plans using SQL

You can directly specify the abstract plan for a query by:

- Using the `create plan` command
- Adding the `plan` clause to `select`, `insert...select`, `update`, `delete` and `return` commands, and to `if` and `while` clauses

For information on writing plans, see Chapter 13, “Abstract Query Plan Guide.”

Using *create plan*

The `create plan` command specifies the text of a query, and the abstract plan to save for the query.

This example creates an abstract plan:

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
```

The plan is saved in the current active plan group. You can also specify the group name:

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
into dev_plans
```

If a plan already exists for the specified query in the current plan group, or the plan group that you specify, you must first enable replace mode in order to overwrite the existing plan.

To see the plan ID that is used for a plan you create, `create plan` can return the ID as a variable. You must declare the variable first. This example returns the plan ID:

```
create plan
    "select avg(price) from titles"
"(scalar_agg
    (i_scan type_price_ix titles)
)"
into dev_plans
and set @id

select @id
```

When you use `create plan`, the query in the plan is not executed. This means that:

- The text of the query is not parsed, so the query is not checked for valid SQL syntax.
- The plans are not checked for valid abstract plan syntax.
- The plans are not checked to determine whether they are compatible with the SQL text.

To guard against errors and problems, immediately execute the specified query with `showplan` enabled.

Using the *plan* clause

You can use the plan clause with the following SQL statements to specify the plan to use to execute the query:

- select
- insert...select
- delete
- update
- if
- while
- return

This example specifies the plan to use to execute the query:

```
select avg(price) from titles
  plan
  "(scalar_agg
    (i_scan type_price_ix titles
  )"

```

When you specify an abstract plan for a query, the query is executed using the specified plan. If you have showplan enabled, this message is printed:

```
Optimized using the Abstract Plan in the PLAN clause.
```

When you use the plan clause with a query, any errors in the SQL text, the plan syntax, and any mismatches between the plan and the SQL text are reported as errors. For example, this plan uses the wrong abstract plan operator for the query:

```
/* wrong operator! */
select * from t1,t2
where c11 = c21
  plan
  "(union
    (t_scan t1)
    (t_scan t2)
  )"

```

This plan returns the following message:

```
Abstract Plan (AP) Warning: An error occurred while applying the AP:
(union (t_scan t1) (t_scan2))
to the SQL query:
select * from t1, t2
```

```
where c11 = c21
```

Failed to apply the top operator 'union' of the following AP fragment:

```
(union (t_scan t1) (t_scan t2))
```

The query contains no union that matches the 'union' AP operator at this point.

The following template can be used as a basis for a valid AP:

```
(also_enforce (join (also_enforce (scan t1)) (also_enforce (scan t2))))  
)
```

The optimizer will complete the compilation of this query; the query will be executed normally.

Plans specified with the plan clause are saved in sysqueryplans only if plan capture is enabled. If a plan for the query already exists in the current capture group, enable replace mode to replace an existing plan.

This chapter covers some guidelines you can use in writing Abstract Plans.

Topic	Page
Overview	335
Tips on writing abstract plans	363
Using abstract plans at the query level	363
Comparing plans before and after	366
Abstract plans for stored procedures	368
Ad hoc queries and abstract plans	370

Overview

Abstract plans allow you to specify the desired execution plan of a query. Abstract plans provide an alternative to the session-level and query-level options that force a join order, or specify the index, I/O size, or other query execution options. The session-level and query-level options are described in Chapter 12, “Creating and Using Abstract Plans.”

There are several optimization decisions that you cannot specify with set commands or clauses in the query text, for example:

- Algorithms that implement a given relational operator; for example, NLJ versus MJ versus HJ or GroupSorted versus GroupHashing versus GroupInserting
- Subquery attachment
- The join order for flattened subqueries
- Reformatting

In many cases when issuing T-SQL commands, you cannot include set commands or change the query text. Abstract plans provide an alternative, more complete method of influencing optimizer decisions.

Abstract plans are relational algebra expressions that are not included in the query text. They are stored in a system catalog and associated with incoming queries based on the text of these queries.

Abstract plan language

The abstract plan language is a relational algebra that uses these operators:

- **distinct** – a logical operator describing duplicates elimination.
 - **distinct_sorted** – a physical operator describing available ordering-based duplicates elimination.
 - **distinct_sorting** – a physical operator describing sorting-based duplicates elimination.
 - **distinct_hashing** – a physical operator describing hashing-based duplicates elimination.
- **group** – a logical operator, describing vector aggregation.
 - **group_sorted** – a physical operator describing the available ordering-based vector aggregation.
 - **group_hashing** – a physical operator describing hashing-based vector aggregation.
 - **group_inserting** – a physical operator describing clustered index insertion-based vector aggregation.
- **join** – the generic join and a high-level logical join operator that describes inner, outer and existence joins, using nested-loop joins, merge joins, or hash joins.
 - **nl_join** – specifying a nested-loop join, including all inner, outer, and existence joins.
 - **m_join** – specifying a merge join, including inner and outer joins.
 - **h_join** – specifying a hash join, including all inner, outer, and existence joins.
- **union** – a logical union operator. It describes both the union and the union all SQL constructs.
 - **append_union_all** – a physical operator implementing union all. It appends the child result sets, one after the other.

- `merge_union_all` – a physical operator implementing union all. It merges the child result sets on the subset of the projection that is ordered in each child, and preserves that ordering.
- `merge_union_distinct` – a physical operator implementing union [distinct]. A merge-based duplicates removal algorithm.
- `hash_union_distinct` – a physical operator implementing union [distinct]. A hash-based duplicates removal algorithm.
- `scalar_agg` – a logical operator, describing scalar aggregation.
- `scan` – a logical operator that transforms a stored table in a flow of rows, an abstract plan derived table. It allows partial plans that do not restrict the access method.
 - `i_scan` – a physical operator implementing scan. It directs the optimizer to use an index scan on the specified table.
 - `t_scan` – a physical operator implementing scan. It directs the optimizer to use a full table scan on the specified table.
 - `m_scan` – a physical operator implementing scan. It directs the optimizer to use a multiindex table scan on the specified table, either index union, index intersection, or both.
- `store` – a physical operator describing the materialization of an abstract plan derived table in a stored worktable.
- `store_index` – a physical operator describing the materialization of an abstract plan derived table in a clustered index stored worktable; the optimizer chooses the useful key columns.
- `sort` – a physical operator describing the sorting of an abstract plan derived table; the optimizer chooses the useful key columns.
- `nested` – a filter describing the placement and structure of nested subqueries.
- `xchg` – a physical operator describing the on-the-fly repartitioning of an abstract plan derived table. The abstract plan gives the target degree, but the optimizer chooses the useful target partitioning.

These additional abstract plan keywords are used for grouping and identification:

- `sequence` – groups the elements when a sequence requires multiple steps.
- `hints` – groups a set of hints for a partial plan.

- `prop` – introduces a set of scan properties for a table: `prefetch`, `lru|mrulru` and `parallel`.
- `table` – identifies a table when correlation names are used, and in subqueries or views.
- `work_t` – identifies a worktable.
- `in` – used with `table` to identify tables named in a subquery (`subq`) or view (`view`).
- `subq` – used under the nested operator to indicate the attachment point for a nested subquery, and to introduce the subqueries' abstract plan.

All legacy abstract plan operators, such as `g_join`, are still accepted for their new counterparts.

Queries, access methods, and abstract plans

For any specific table, there can be several access methods for a specific query; index scans using different indexes, table scans, the OR strategy, and reformatting.

This simple query has several choices of access methods:

```
select * from t1
where c11 > 1000 and c12 < 0
```

The following abstract plans specify three different access methods:

- Use the index `i_c11`:

```
(i_scan i_c11 t1)
```
- Use the index `i_c12`:

```
(i_scan i_c12 t1)
```
- Do a full table scan:

```
(t_scan t1)
```
- Do a multi-scan; that is, the union or intersection of several indexes of the table, according to the complex clause (hence the more complex query used in this example):

```
select * from t1
where (c11 > 1000 or c12 < 0) and (c12 > 1000 or c112 < 0)
plan
"(m_scan t1)"
```

Abstract plans can be full plans, specifying all optimizer choices for a query, or can specify a subset of the choices, such as the index to use for a single table in the query, but not the join order for the tables. For example, using a partial abstract plan, you can specify that the query above should use some index and let the optimizer choose between `i_c11` and `i_c12`, but not do a full table scan. The empty parentheses are used in place of the index name:

```
(i_scan () t1)
```

In addition, the query could use either 2K or 16K I/O, or be performed in serial or parallel.

Derived tables

A derived table is defined by the evaluation of a query expression and differs from a regular table in that it is neither described in system catalogs nor stored on disk. In Adaptive Server, a derived table may be a SQL derived table or an abstract plan derived table.

- A SQL derived table – defined by one or more tables through the evaluation of a query expression. A SQL derived table is used in the query expression in which it is defined and exists only for the duration of the query. See the *Transact-SQL User's Guide*.
- An abstract plan derived table – a derived table used in query processing, the optimization and execution of queries. An abstract plan derived table differs from a SQL derived table in that it exists as part of an abstract plan and is invisible to the end user.

Identifying tables

Abstract plans must name all of a query's tables in a nonambiguous way, such that a table named in the abstract can be linked to its occurrence in the SQL query. In most cases, the table name is all that is needed. If the query qualifies the table name with the database and owner name, these are also needed to fully identify a table in the abstract plan. For example, this example uses the unqualified table name:

```
select * from t1
```

The abstract plan also uses the unqualified name, (`t_scan t1`). If a database name or owner name are provided in the query:

```
select * from pubs2.dbo.t1
```

The abstract plan must use qualifications, (t_scan pubs2.dbo.t1). However, the same table may occur several times in the same query, as in this example:

```
select * from t1 a, t1 b
```

Correlation names, a and b in the example above, identify the two tables in SQL. In an abstract plan, the table operator associates each correlation name with the occurrence of the table:

```
(join
  (t_scan (table (a t1)))
  (t_scan (table (b t1)))
)
```

You can also use a briefer abstract plan, which uses only the correlation names:

```
(join
  (t_scan a)
  (t_scan b)
)
```

Table names can also be ambiguous in views and subqueries, so the table operator is used for tables in views and subqueries.

For subqueries, the in and subq operators qualify the name of the table with its syntactical containment by the subquery. The same table is used in the outer query and the subquery in this example:

```
select *
from t1
where c11 in (select c12 from t1 where c11 > 100)
```

The abstract plan identifies the tables unambiguously:

```
(join
  (t_scan t1)
  (i_scan i_c11_c12 (table t1 (in (subq 1))))
)
```

For views, the in and view operators provide the identification. The query in this example references a table used in the view:

```
create view v1
as
select * from t1 where c12 > 100
select t1.c11 from t1, v1
  where t1.c12 = v1.c11
```

Here is the abstract plan:

```
(join
  (t_scan t1)
  (i_scan i_c12 (table t1 (in (view v1))))
)
```

In abstract plans generated by Adaptive Server, the view or subquery-qualified table names are generated only for the tables where they are needed to remove name ambiguity. For other tables, only the name is generated.

In abstract plans created by the user, view or subquery-qualified tables names are required in case of ambiguity; both syntaxes are accepted otherwise.

Identifying indexes

The `i_scan` operator requires two operands, the index name and the table name, as shown here:

```
(i_scan i_c12 t1)
```

To specify that some index should be used, without specifying the index, substitute empty parenthesis for the index name:

```
(i_scan () t1)
```

Specifying join order

Adaptive Server performs joins of three or more tables by joining two of the tables, and joining the abstract plan derived table from that join to the next table in the join order. This abstract plan derived table is a flow of rows, as from an earlier nested-loop join in the query execution.

This query joins three tables:

```
select *
from t1, t2, t3
where c11 = c21
      and c12 = c31
      and c22 = 0
      and c32 = 100
```

This example shows the binary nature of the join algorithm, using join operators. The plan specifies the join order `t2, t1, t3`:

```
(join
  (join
    (scan t2)
    (scan t1)
  )
  (scan t3)
)
```

The results of the $t2$ - $t1$ join are then joined to $t3$. The scan operator in this example leaves the choice of table scan or index scan up to the optimizer.

Shorthand notation for joins

In general, an N -way left deep nested loops join, with the order $t1, t2, t3, \dots, tN-1, tN$ is described by:

```
(join
  (join
    ...
    (join
      (join
        (scan t1)
        (scan t2)
      )
      (scan t3)
    )
    ...
    (scan tN-1)
  )
  (scan tN)
)
```

This notation can be used as shorthand for the `nl_join` operator:

```
(nl_join
  (scan t1)
  (scan t2)
  (scan t3)
  ...
  (scan tN-1)
  (scan tN)
)
```


Join order examples

The optimizer could select among several plans for this three-way join query:

```
select *
from t1, t2, t3
where c11 = c21
      and c12 = c31
      and c22 = 0
      and c32 = 100
```

Here are a few examples:

- Use c22 as a search argument on t2, join with t1 on c11, then with t3 on c31:

```
(nl_join
  (i_scan i_c22 t2)
  (i_scan i_c11 t1)
  (i_scan i_c31 t3)
)
```

- Use the search argument on t3, and the join order t3, t1, t2:

```
(nl_join
  (i_scan i_c32 t3)
  (i_scan i_c12 t1)
  (i_scan i_c21 t2)
)
```

- Do a full table scan of t2, if it is small and fits in cache, still using the join order t3, t1, t2:

```
(nl_join
  (i_scan i_c32 t3)
  (i_scan i_c12 t1)
  (t_scan t2)
)
```

- If t1 is very large, and t2 and t3 individually qualify a large part of t1, but together a very small part, this plan specifies a star join:

```
(nl_join
  (i_scan i_c22 t2)
  (i_scan i_c32 t3)
  (i_scan i_c11_c12 t1)
)
```

The join operators are generic in that they implement any of the outer joins, inner joins, and existence joins; the optimizer chooses the correct join semantics according to the query semantics.

Match between execution methods and abstract plans

There are some limits to join orders and join types, depending on the type of query. One example is outer joins, such as:

```
select *
from t1 left join t2
on c11 = c21
```

Adaptive Server requires the outer member of the outer join to be the outer table during join processing. Therefore, this abstract plan is illegal:

```
(join
  (scan t2)
  (scan t1)
)
```

Attempting to use this plan results in an error message, the AP application fails, and the optimizer makes the best attempt to finish compiling the query.

Specifying join order for queries using views

You can use abstract plans to enforce the join order for merged views. This example creates a view that performs a join of t2 and t3:

```
create view v2
as
select *
from t2, t3
where c22 = c32
```

This query performs a join with the t2 in the view:

```
select * from t1, v2
where c11 = c21
and c22 = 0
```

This abstract plan specifies the join order t2, t1, t3:

```
(nl_join
  (scan t2)
  (scan t1)
  (scan t3))
```

```
)
```

Since the table names are not ambiguous, the view qualification is not needed. However, the following abstract plan is also legal and has the same meaning:

```
(nl_join
  (scan (table t2 (in (view v2))))
  (scan t1)
  (scan (table t3 (in (view v2))))
)
```

This example joins with t3 in the view:

```
select * from t1, v2
where c11 = c31
      and c32 = 100
```

This plan uses the join order t3, t1, t2:

```
(join
  (scan t3)
  (scan t1)
  (scan t2)
)
```

This is an example where abstract plans can be used, if needed, to affect the join order for a query, when set forceplan cannot.

Specifying the join type

Adaptive Server can perform nested-loop, merge, or hash joins. The join operator leaves the optimizer free to choose the best join algorithm, based on costing. To specify a nested-loop join, use the `nl_join` operator; for a merge join, use the `m_join` operator, and for a hash join, use the `h_join` operator. Abstract plans captured by Adaptive Server always include the operator that specifies the algorithm, and not the join operator.

This query specifies a join between t1 and t2:

```
select * from t1, t2
      where c12 = c21 and c11 = 0
```

This abstract plan specifies a nested-loop join:

```
(nl_join
  (i_scan i_c11 t1)
  (i_scan i_c21 t2)
```

```
)
```

The nested-loop plan uses the index `i_c11` to limit the scan using the search clause, and then performs the join with `t2`, using the index on the join column.

This merge-join plan uses different indexes:

```
(m_join
  (i_scan i_c12 t1)
  (i_scan i_c21 t2)
)
```

The merge join uses the indexes on the join columns, `i_c12` and `i_c21`, for the merge keys. This query performs a full-merge join and no sort is needed.

A merge join could also use the index on `i_c11` to select only the matching rows, but then a sort is needed to provide the needed ordering.

```
(m_join
  (sort
    (i_scan i_c11 t1)
  )
  (i_scan i_c21 t2)
)
```

Finally, this plan does a hash join and a full table scan on the inner side:

```
(h_join
  (i_scan i_c11 t1)
  (t_scan t2)
)
```

Specifying partial plans and hints

Sometimes a full plan is not needed, for example, if the only problem with a query plan is that the optimizer chooses a table scan instead of using a nonclustered index, the abstract plan can specify only the index choice, and leave the other decisions to the optimizer.

The optimizer could choose a table scan of `t3` rather than using `i_c31` for this query:

```
select *
from t1, t2, t3
where c11 = c21
```

```
and c12 < c31
and c22 = 0
and c32 = 100
```

The following plan, as generated by the optimizer, specifies join order t2, t1, t3. However, the plan specifies a table scan of t3:

```
(nl_join
  (i_scan i_c22 t2)
  (i_scan i_c11 t1)
  (t_scan t3)
)
```

This full plan could be modified to specify the use of i_c31 instead:

```
(nl_join
  (i_scan i_c22 t2)
  (i_scan i_c11 t1)
  (i_scan i_c31 t3)
)
```

However, specifying only a partial abstract plan is a more flexible solution. As data in the other tables of that query evolves, the optimal join order can change. The partial plan can specify just one partial plan item. For the index scan of t3, the partial plan is simply:

```
(i_scan i_c31 t3)
```

The optimizer chooses the join order and the access methods for t1 and t2.

Abstract plans are partial by using logical operators instead of physical operators. For example, the following abstract plan is partial, although it covers the entire query, as it lets the optimizer choose the join algorithms and the access methods:

```
(join
  (scan t1)
  (scan t2)
  (scan t3)
)
```

Partial plans may also be incomplete at the top, in that the root of the abstract plan may cover only a part of the query. If this is the case, the optimizer completes the plan:

```
(nl_join
  (t_scan t1)
  (t_scan t2)
)
```

However, the plan fragment given in an abstract plan must be complete down to the leafs. For example, the following abstract plan, which reads “hash join t1 outer to something” is illegal.

```
(h_join
  (t_scan t1)
  ()
)
```

Grouping multiple hints

Sometimes more than one plan fragment is needed. For example, you might want to specify that some index should be used for each table in the query, but leave the join order up to the optimizer. When multiple hints are needed, you can group them with the hints operator:

```
(hints
  (i_scan () t1)
  (i_scan () t2)
  (i_scan () t3)
)
```

In this case, the role of the hints operator is purely syntactic; it does not affect the ordering of the scans.

There are no limits on what may be given as a hint. Partial join orders may be mixed with partial access methods. This hint specifies that t2 is outer to t1 in the join order, and that the scan of t3 should use an index, but the optimizer can choose the index for t3, the access methods for t1 and t2, and the placement of t3 in the join order:

```
(hints
  (join
    (scan t2)
    (scan t1)
  )
  (i_scan () t3)
)
```

Inconsistent and illegal plans using hints

It is possible to describe inconsistent plans using hints, such as this plan that specifies contradictory join orders:

```
(hints
  (join
    (scan t2)
```

```
        (scan t1)
      )
    (join
      (scan t1)
      (scan t2)
    )
  )
```

When the query associated with the plan is executed, the query cannot be compiled, and an error is raised.

Other inconsistent hints do not raise an exception, but may use any of the specified access methods. This plan specifies both an index scan and a table scan for the same table:

```
(hints
  (t_scan t3)
  (i_scan () t3)
)
```

In this case, either method may be chosen, and the behavior is indeterminate.

Creating abstract plans for subqueries

Subqueries are resolved in several ways in Adaptive Server, and the abstract plans reflect the query execution steps:

- **Materialization** – the subquery is executed and results are stored in a worktable or internal variable. See “Materialized subqueries” on page 350.
- **Flattening** – the query is flattened into a join with the tables in the main query. See “Flattened subqueries” on page 350.
- **Nesting** – the subquery is executed once for each outer query row. See “Nested subqueries” on page 352.

Abstract plans do not allow the choice of the basic subquery resolution method. This is a rule-based decision and cannot be changed during query optimization. Abstract plans, however, can be used to influence the plans for the outer and inner queries. In nested subqueries, abstract plans can also be used to choose where the subquery is nested in the outer query.

Materialized subqueries

This query includes a noncorrelated subquery that can be materialized:

```
select *
from t1
where c11 = (select count(*) from t2)
```

The first step in the abstract plan materializes the scalar aggregate in the subquery. The second step uses the result to scan t1:

```
( sequence
  (scalar_agg
    (i_scan i_c21 t2)
  )
  (i_scan i_c11 t1)
)
```

Flattened subqueries

Some subqueries can be flattened into joins. The join, nl_join, m_join, and h_join operators leave it to the optimizer to detect when an existence join is needed. For example, this query includes a subquery introduced with exists:

```
select * from t1
where c12 > 0
      and exists (select * from t2
                  where t1.c11 = c21 and c22 < 100)
```

The semantics of the query require an existence join between t1 and t2. The join order t1, t2 is interpreted by the optimizer as a semijoin, with the scan of t2 stopping on the first matching row of t2 for each qualifying row in t1:

```
(join
  (scan t1)
  (scan t2)
)
```

The join order t2, t1 requires other means to guarantee the duplicate elimination:

```
(join
  (distinct
    (scan t2)
  )
  (scan t1)
)
```


Using this abstract plan, the optimizer can decide to use:

- A unique index on t2.c21, if one exists, with a regular join.
- The unique reformatting strategy, if no unique index exists. In this case, the query will probably use the index on c22 to select the rows into a worktable.
- The duplicate elimination sort optimization strategy, performing a regular join and selecting the results into the worktable, then sorting the worktable.

The abstract plan does not need to specify the creation and scanning of the worktables needed for the last two options.

Example of changing the join order in a flattened subquery

The query can be flattened to an existence join:

```
select *
from t1, t2
where c11 = c21
      and c21 > 100
      and exists (select * from t3 where c31 != t1.c11)
```

The “!=” correlation can make the scan of t3 rather expensive. If the join order is t1, t2, the best place for t3 in the join order depends on whether the join of t1 and t2 increases or decreases the number of rows, and therefore, the number of times that the expensive table scan needs to be performed. If the optimizer fails to find the right join order for t3, the following abstract plan can be used when the join reduces the number of times that t3 must be scanned:

```
(nl_join
  (scan t1)
  (scan t2)
  (scan t3)
)
```

If the join increases the number of times that t3 needs to be scanned, this abstract plan performs the scans of t3 before the join:

```
(nl_join
  (scan t1)
  (scan t3)
  (scan t2)
)
```

Nested subqueries

Nested subqueries can be explicitly described in abstract plans if:

- The abstract plan for the subquery is provided.
- The location at which the subquery attaches to the main query is specified.

Abstract plans allow you to affect the query plan for the subquery, and to change the attachment point for the subquery in the outer query.

The nested operator specifies the position of the subquery in the outer query. Subqueries are “nested over” a specific abstract plan derived table. The optimizer chooses a spot where all the correlation columns for the outer query are available, and where it estimates that the subquery needs to be executed the least number of times.

The following SQL statement contains a correlated expression subquery:

```
select *
from t1, t2
where c11 = c21
      and c21 > 100
      and c12 = (select c31 from t3
                  where c32 = t1.c11)
```

The abstract plan shows the subquery nested over the scan of t1:

```
(nl_join
  (nested
    (i_scan i_c12 t1)
    (subq
      (scalar_agg
        (scan t3)
      )
    )
  )
  (i_scan i_c21 t2)
)
```

Aggregation is described in Chapter 2, “Using showplan.” The scalar_agg abstract plan operator is necessary because all abstract plans, even partial ones, must be complete down to the leafs.

Subquery identification and attachment

Subqueries in the SQL query are matched against abstract plan subqueries using their underlying tables. As tables are unambiguously identified, so are the subqueries. For example:

```
select
  (select c11 from t1 where c12 = t3.c32), c31
from t3
where
  c32 > (select c22 from t2 where c21 = t3.c31)
plan
"(nested
  (nested
    (t_scan t3)
    (subq
      (i_scan i_c11_c12 t1)
    )
  )
  (subq
    (i_scan i_c21 t2)
  )
)"
```

However, when table names are ambiguous, the identity of the subquery is needed to solve the table name ambiguity.

Subqueries are identified with numbers, in the order of their leading opened parenthesis “(“.

This example has two subqueries; both refer to table t1:

```
select 1
from t1
where
  c11 not in (select c12 from t1)
  and c11 not in (select c13 from t1)
```

In the abstract plan, the subquery which projects out of c12 is named “1” and the subquery which projects out of c13 is named “2”.

```
(nested
  (nested
    (t_scan t1)
    (subq
      (scalar_agg
        (i_scan i_c11_c12 (table t1 (in (subq 1))))
      )
    )
  )
)
```

```
    )
    (subq
      (scalar_agg
        (i_scan i_c13 (table t1 (in (subq 2)))))
      )
    )
  )
```

In this query, the second subquery is nested in the first:

```
select * from t1
where c11 not in
  (select c12 from t1
   where c11 not in
     (select c13 from t1))
```

In this case, the subquery that projects out of c12 is also named “1” and the subquery that projects out of c13 is also named “2”.

```
(nested
  (t_scan t1
    (subq
      (scalar_agg
        (nested
          (i_scan i_c12 (table t1 (in (subq 1)))))
          (subq
            (scalar_agg
              (i_scan i_c21 (table t1 (in (subq 2)))))
            )
          )
        )
      )
    )
  )
)
```

More subquery examples: reading ordering and attachment

The nested operator has the abstract plan derived table as the first operand and the nested subquery as the second operand. This allows an easy vertical reading of the join order and subquery placement:

```
select *
from t1, t2, t3
where c12 = 0
      and c11 = c21
      and c22 = c32
      and 0 < (select c21 from t2 where c22 = t1.c11)
```

In the plan, the join order is t1, t2, t3, with the subquery nested over the scan of t1:

```
(nl_join
  (nested
    (i_scan i_c11 t1)
    (subq
      (t_scan (table t2 (in (subq 1)))
    )
  )
  (i_scan i_c21 t2)
  (i_scan i_c32 t3)
)
```

Modifying subquery nesting

If you modify the attachment point for a subquery, you must choose a point at which all of the correlation columns are available. This query is correlated to two of the tables in the outer query:

```
select *
from t1, t2, t3
where c12 = 0
      and c11 = c21
      and c22 = c32
      and 0 < (select c31 from t3 where c31 = t1.c11
              and c32 = t2.c22)
```

This plan uses the join order t1, t2, t3, with the subquery nested over the t1-t2 join:

```
(nl_join
  (nested
    (nl_join
      (i_scan i_c11_c12 t1)
      (i_scan i_c22 t2)
    )
    (subq
      (t_scan (table t3 (in (subq 1))))
    )
  )
  (i_scan i_c32 t3)
)
```

Since the subquery requires columns from both outer tables, it would be incorrect to nest it over the scan of t1 or the scan of t2; such errors are silently corrected during optimization.

However, the following abstract plan makes the legal request to nest the subquery over the three-table join:

```
(nested
  (nl_join
    (i_scan i_c11_c12 t1)
    (i_scan i_c22 t2)
    (i_scan i_c32 t3)
  )
  (subq
    (t_scan (table t3 (in (subq 1))))
  )
)
```

Abstract plans for materialized processing of views

In most cases, view processing merges the view definition in the main query. There are, however, cases when a view needs to be materialized, as in the case of a self-join:

```
create view v3(cc31, sum_c32)
as
select c31, sum(c32)
from t3
group by c31

select *
from v3 a, v3 b
where a.c31 = b.c31
```

In such a case, the abstract plan exposes the worktable and the store operator that materializes it. The two scans of the worktable are identified through their correlation names:

```
(sequence
  (store
    (group_sorted
      (i_scan i_c31 t3)
    )
  )
  (m_join
    (sort
      (t_scan (work_t (a Worktable)))
    )
    (sort
      (t_scan (work_t (b Worktable)))
    )
  )
)
```

```
)  
)
```

The handling of vector aggregation in an abstract plan is described in the next section.

Abstract plans for queries containing aggregates

This query returns a scalar aggregate:

```
select max(c11) from t1
```

There is a physical operator that implements scalar aggregation, therefore, the optimizer has no choice. However, choosing an index on c11 allows the max() optimization:

```
(scalar_agg  
  (i_scan ic11 t1)  
)
```

Since the scalar aggregate is the top abstract plan operator, removing it and using the following partial plan has the same outcome:

```
(i_scan ic11 t1)
```

The scalar_agg abstract plan is typically needed when it is part of a subquery and the abstract plan must cover the parent query as well.

Vector aggregation is different, in that there are several physical operators to implement the group logical operator, which means that the optimizer has a choice to make. Thus, the abstract plan can force it.

```
select max(c11)  
from t1  
group by c12
```

The following abstract plan examples force each of the three vector aggregation algorithms:

Note group_sorted requires an ordering on the grouping column, so it needs to use an index.

```
(group_sorted  
  (i_scan i_c12 t1)  
)  
(group_hashing  
  (t_scan t1))
```

```
)  
  
(group_inserting  
  (t_scan t1)  
)
```

Abstract plans for queries containing unions

The union abstract plan operator describes plans for SQL queries that contain unions:

```
select*  
from  
  t1,  
  (select * from t2  
   union  
   select * from t3  
  ) u(u1, u2)  
where c11=u1  
plan  
"(nl_join  
 (union  
  (t_scan t2)  
  (t_scan t3)  
 )  
 (i_scan i_c11 t1)  
)"
```

There are two types of union in SQL: union distinct and union [all]. union [all] is the default.

The `m_union_distinct` and `h_union_distinct` abstract plan operators force the removal of merge or hash-based UNION DISTINCT duplicates. It is illegal to use these operators with a UNION ALL. The merge-based algorithm needs, from each of the union children, an ordering covering all union projection columns.

In the following example, the needed ordering is provided, for the first child, by the `(c11, c12)` composite index and, for the second child, by the sort.

```
select c11, c12 from t1  
union distinct  
select c21, c22 from t2  
plan  
"(m_union_distinct
```



```
(i_scan i_c11_c12 t1)
(sort
 (t_scan t2)
)
)"
```

The `union_all` and `m_union_all` abstract plan operators force the append- or merge-based UNION ALL. It is illegal to use these operators with a UNION DISTINCT. The merge algorithm needs no ordering for itself; it makes any useful ordering from the children available to the parent.

In the following example, the ordering provided by the two `i_scan` operators is made available, by their `m_union_all` parent, to the `m_join` above.

```
select *
from
  t1,
  (select c21, c22 from t2
   union
   select c31, c32 from t3
 ) u(u1, u2)
where c11=u1
plan
"(m_join
 (m_union_all
  (i_scan i_c21 t2)
  (i_scan i_c31 t3)
 )
 (i_scan i_c11 t1)
)"
```

Using abstract plans when queries need ordering

An ordering is needed either explicitly, in an ORDER BY query, or implicitly by merge-based operators such as `m_join`, `m_union_distinct`, and `group_sorted`.

An ordering is produced either explicitly, by the `sort` abstract plan operator (the optimizer build the sort key on all columns known to need an ordering), or implicitly by an `i_scan` on the indexed columns.

All merge-based operators that require ordering preserve it in their results for a parent that also requires it.

In the following example, the `i_scan` of `t1` provides the ordering needed by the `m_join`. The `i_scan` of `t2`, and the sort over `t3`'s scan, provides the ordering needed by `m_union_distinct`. This ordering also provides the ordering needed by the `m_join`. Finally, no top sort is required as the ordering needed by `ORDER BY` is provided by the `m_join`.

```
select *
from
  t1,
  (select c21, c22 from t2
   union distinct
   select c31, c32 from t3
  ) u(u1, u2)
where c11=u1
order by c11, u2
plan
"(m_join
  (m_union_distinct
    (i_scan i_c21_c22 t2)
    (sort
      (t_scan t3)
    )
  )
  (i_scan i_c11 t1)
)"
```

Specifying the reformatting strategy

In this query, `t2` is very large, and has no index:

```
select *
from t1, t2
where c11 > 0
      and c12 = c21
      and c22 = 0
```

The abstract plan that specifies the reformatting strategy on `t2` is:

```
(nl_join
  (t_scan t1)
  (store_index
    (t_scan t2)
  )
)
```

The `store_index` abstract plan operator must be placed on the inner side of an `n_l_join`. It can be placed over any abstract plan; there is no longer a single table scan limitation. The legacy `(scan (store...))` syntax is still accepted.

Specifying the OR strategy

An OR strategy uses a set of index scans to limit the scan with each of the OR terms, then passes the resulting row IDs through a `UnionDistinct` operator to get, with a `RidJoin` from the table, the tuples corresponding to the unique row IDs.

The `m_scan` (multiscan) abstract plan operator forces index union, hence the OR strategy:

```
select * from t1
where c11 > 10 or c12 > 100
plan
"(m_scan t1)"
```

When the *store* operator is not specified

Storing the stream of tuples into a worktable to meet the intraoperator needs of an algorithm (Sort, GroupInserting, and so on), is treated as a implementation detail of the algorithm and thus is not exposed in the abstract plan.

Abstract plans expose only the worktables created for interoperator reasons, such as the self-joined materialized view. In such a case, none of the operators needs a work table. The cause is, rather, the global nature of the plan, of computing an intermediate derived table once and using it twice.

Abstract plans for parallel processing

Partitioned tables scanned in parallel produce partitioned streams of tuples. Different operators have specific needs for parallel processing. For instance, in all joins, either both children must be equipartitioned, or one child must be replicated.

The abstract plan `xchg` operator forces the optimizer to repartition, on-the-fly, in n ways, its child-derived table. The abstract plan only gives the degree. The optimizer chooses the most useful partitioning columns and style (hash, range, list, or round-robin).

In the following example, assume that `t1` and `t2` are hash partitioned two ways and three ways on the join columns, and `i_c21` is a local index:

```
select *
from t1, t2
where c11=c21
```

The following abstract plan repartitions `t1` three ways, does a three-way parallel `nl_join`, serializes the results, and returns a single data stream to the client:

```
(xchg 1
  (nl_join
    (xchg 3
      (t_scan t1)
    )
    (i_scan i_c21 t2)
  )
)
```

It is not necessary to specify `t2`'s parallel scan. It is hash-partitioned three ways, and, as it is joined with an `xchg-3`, no other plan is legal.

The following abstract plan scans and sorts `t1` and `t2` in parallel, as each is partitioned, then serializes them for the `m_join`:

```
(m_join
  (xchg 1
    (sort
      (t_scan t1)
    )
  )
  (xchg 1
    (sort
      (t_scan t2)
    )
  )
)
(prop t1 (parallel 2))
(prop t2 (parallel 3))
```

The parallel abstract plan construct is used to make sure that the optimizer chooses the parallel scan with the native degree.

Tips on writing abstract plans

Here are some additional tips for writing and using abstract plans:

- Look at the current plan for the query and at plans that use the same query execution steps as the plan you need to write. It is often easier to modify an existing plan than to write a full plan from scratch.
 - Capture the plan for the query.
 - Use `sp_help_qplan` to display the SQL text and plan.
 - Edit this output to generate a create plan command, or attach an edited plan to the SQL query using the plan clause.
- It is often best to specify partial plans for query tuning in cases where most optimizer decisions are appropriate, but only an index choice, for example, needs improvement.

By using partial plans, the optimizer can choose other paths for other tables as the data in other tables changes.

- Once saved, abstract plans are static. Data volumes and distributions may change so that saved abstract plans are no longer optimal.

Subsequent tuning changes made by adding indexes, partitioning a table, or adding buffer pools may mean that some saved plans are not performing as well as possible under current conditions. Most of the time, operate with a small number of abstract plans that solve specific problems.

Perform periodic plan checks to verify that the saved plans are still better than the plan that the optimizer would choose.

Using abstract plans at the query level

You can use abstract plans to force the query plan the query processor chooses to allow several query-level settings. See Chapter 7, “Controlling Optimization,” for more information about using abstract plans at the query level.

The optimization criteria are handled at the session level by the following set statements:

```

set
    nl_join|merge_join|hash_join|...
on | off

```

The use ... abstract plan syntax accepts any number of use forms before the abstract plan derived table. In versions of Adaptive Server earlier than 15.0, optgoal and opttimeout could not be in the same abstract plan with a derived table. For example, this statement would need to be separate from a optgoal statement in a query:

```

select ...
    plan
        "(use opttimeoutlimit 10) (i_scan r)"

```

However, you can include several statements in the same abstract plan by:

- Using several use statements. For example:

```

select ...
    plan
        "(use optgoal allrows_dss) (use nl_join off)
        (...)"

```

- Placing several items within one use form. For example:

```

select ...
    plan
        "(use (optgoal allrows_dss) (nl_join off))
        (...)"

```

At the query level, use the optimization goal (opt_goal) or timeout (opttimeout) setting with the use ... abstract plan syntax. At the session level, use these settings with the set plan ... syntax:

- Optimization goal.
- Optimization timeout

For example, join r outer to s and enable the hash_join without an optimization goal (opt_goal):

```

select ...
>
>          plan
>
>          "(use hash_join on)
>
>          (join (scan r) (scan s))"

```

This example uses the opt_goal and allrows_oltp statements, but with hash_join enabled:

```

select ...
>
>    plan
>
>    "(use opt_goal allrows_oltp) (use hash_join
on) "

```

When setting the optimization goal and the optimization criteria at the query level, the order of the use statements does not affect the outcome.

- The abstract plan optimization goal is set first, and sets the optimization goal defaults for the optimization criteria.
- You can set abstract plan optimization, which supersedes optimization goal defaults criteria, you set the optimization goal.

Operator name alignment for abstract plan and optimizer criteria

The names of algorithms differ in how you use them in abstract plans and how you use them in the set command. For example, a hash join is called `h_join` in abstract plans, but is called `hash_join` in the set command. Adaptive Server accepts both keywords in the extended abstract plan syntax. For example:

```

select ...

plan

"(h_join (t_scan r) (t_scan s))"

```

is equivalent to:

```

select ...

plan

"(hash_join (t_scan r) (t_scan s))"

```

and:

```

select ...
plan
"(use h_join on) "

```

and:

```

select ...
plan

```

```
"(use hash_join on) "
```

When a table abstract plan is present, it takes precedence:

```
select ..  
from r, s, t  
...  
plan  
"(use hash_join off)  
(h_join (t_scan r) (t_scan s)) "
```

The query uses the hash_join for r and s scans; but for the join with t it does not use hash_join as specified by the use abstract plan form, since it was not specified in the table abstract plan.

Extending the optimizer criteria set syntax

The set *opt criteria* statement accepts on/off/default, where default indicates that you are using the current optimization goal setting for this optimization criteria (for the complete set syntax, see *Reference Manual: Commands*).

Comparing plans before and after

Use abstract query plans to assess the impact of an Adaptive Server software upgrade or system tuning changes on your query plans. You must save plans before the changes are made, perform the upgrade or tuning changes, and then save plans again and compare the plans. The basic set of steps is:

- 1 Enable server-wide capture mode by setting the configuration parameter abstract plan dump to 1. All plans are then captured in the default group, ap_stdout.
- 2 Allow enough time for the captured plans to represent most of the queries run on the system. You can check whether additional plans are being generated by checking whether the count of rows in the ap_stdout group in sysqueryplans is stable:

```
select count(*) from sysqueryplans where gid = 2
```


- 3 Copy all plans from `ap_stdout` to `ap_stdin` (or some other group, if you do not want to use server-wide plan load mode), using `sp_copy_all_qplans`.
- 4 Drop all query plans from `ap_stdout`, using `sp_drop_all_qplans`.
- 5 Perform the upgrade or tuning changes.
- 6 Allow sufficient time for plans to be captured to `ap_stdout`.
- 7 Compare plans in `ap_stdout` and `ap_stdin`, using the `diff` mode parameter of `sp_cmp_all_qplans`. For example, this query compares all plans in `ap_stdout` and `ap_stdin`:

```
sp_cmp_all_qplans ap_stdout, ap_stdin, diff
```

This displays only information about the plans that are different in the two groups.

Effects of enabling server-wide capture mode

When server-wide capture mode is enabled, plans for all queries that can be optimized are saved in all databases on the server. Some possible system administration impacts are:

- When plans are captured, the plan is saved in `sysqueryplans` and log records are generated. The amount of space required for the plans and log records depends on the size and complexity of the SQL statements and query plans. Check space in each database where users will be active.

You may need to perform more frequent transaction log dumps, especially in the early stages of server-wide capture when many new plans are being generated.

- If users execute system procedures from the master database, and `installmaster` was loaded with server-wide plan capture enabled, then plans for the statements that can be optimized in system procedures are saved in `master.sysqueryplans`.

This is also true for any user-defined procedures created while plan capture was enabled. You may want to provide a default database at login for all users, including system administrators, if space in `master` is limited.

- The sysqueryplans table uses datarows locking to reduce lock contention. However, especially when a large number of new plans are being saved, there may be a slight impact on performance.
- While server-wide capture mode is enabled, using bcp saves query plans in the master database. If you perform bcp using a large number of tables or views, check sysqueryplans and the transaction log in master.

Time and space to copy plans

If you have a large number of query plans in ap_stdout, be sure there is sufficient space to copy them on the system segment before starting the copy. Use sp_spaceused to check the size of sysqueryplans, and sp_helpsegment to check the size of the system segment.

Copying plans also requires space in the transaction log.

sp_copy_all_qplans calls sp_copy_qplan for each plan in the group to be copied. If sp_copy_all_qplans fails at any time due to lack of space or other problems, any plans that were successfully copied remain in the target query plan group.

Abstract plans for stored procedures

For abstract plans to be captured for the SQL statements that can be optimized in stored procedures:

- The procedures must be created while plan capture or plan association mode is enabled. (This saves the text of the procedure in sysprocedures.)
- The procedure must be executed with plan capture mode enabled, and the procedure must be read from disk, not from the procedure cache.

This sequence of steps captures the query text and abstract plans for all statements in the procedure that can be optimized:

```
set plan dump dev_plans on
go
create procedure myproc as ...
go
```

```
exec myproc  
go
```

If the procedure is in cache, and the plans for the procedure are not being captured, execute the procedure with recompile. Similarly, once a stored procedure has been executed using an abstract query plan, the plan in the procedure cache is used so that query plan association does not take place unless the procedure is read from disk.

You can use `set fmtonly on` to capture plans for a stored procedure without actually executing the statements in a stored procedure.

Procedures and plan ownership

When plan capture mode is enabled, abstract plans for the statements in a stored procedure that can be optimized are saved with the user ID of the owner of the procedure.

During plan association mode, association for stored procedures is based on the user ID of the owner of the procedure, not the user who executes the procedure. This means that once an abstract query plan is created for a procedure, all users who have permission to execute the procedure use the same abstract plan.

Procedures with variable execution paths and optimization

Executing a stored procedure saves abstract plans for each statement that can be optimized, even if the stored procedure contains control-of-flow statements that can cause different statements to be run, depending on parameters to the procedure or other conditions.

Adaptive Server loads and saves the abstract plans when the stored procedures are compiled, not when they are executed.

When Adaptive Server compiles a stored procedure (usually when it is first run), it saves an abstract plan for each optimized statement. Adaptive Server does not influence the abstract plan capture, or whether the stored procedure contains control-of-flow statements that cause different statements to be executed, depending on the procedure's parameters.

If you run the query a second time (without recompilation) with different parameters that use a different code path, because Adaptive Server already optimized and saved the plans for all statements from the earlier compilation, both the plans and the abstract plans for the statements in this different code path are available, and are based on the prior stored procedure's run parameter values, whether or not these statement were executed.

However, abstract plans for procedures do not solve the problem caused by procedures with statements that are optimized differently depending on conditions or parameters. For example is a procedure where users provide the low and high values for a between clause, with a query such as:

```
select title_id
from titles
where price between @lo and @hi
```

Depending on the parameters, the best plan could either be an index access or a table scan. The abstract plan may specify either access method, depending on the parameters used for the initial execution of the procedure. Abstract plans that are saved while executing queries or stored procedures in tempdb are lost if the server is restarted.

Ad hoc queries and abstract plans

Abstract plan capture saves the full text of the SQL query and abstract plan association is based on the full text of the SQL query. If users submit ad hoc SQL statements, rather than using stored procedures or Embedded SQL, abstract plans are saved for each different combination of query clauses. This can result in a very large number of abstract plans.

For example, if users check the price of a specific title_id using select statements, an abstract plan is saved for each statement. The following two queries each generate an abstract plan:

```
select price from titles where title_id = "T19245"
select price from titles where title_id = "T40007"
```

In addition, there is one plan for each user, that is, if several users check for the title_id "T40007," a plan is save for each user ID.

If such queries are included in stored procedures, there are two benefits:

- Only only one abstract plan is saved, for example, for the query:

```
select price from titles where title_id =  
@title_id
```

- The plan is saved with the user ID of the user who owns the stored procedure, and abstract plan association is made based on the procedure owner's ID.

Using Embedded SQL, the only abstract plan is saved with the host variable:

```
select price from titles  
where title_id = :host_var_id
```


Managing Abstract Plans with System Procedures

This chapter provides an introduction to the basic functionality and use of the system procedures for working with abstract plans. For detailed information on each procedure, see the *Reference Manual: Procedures*.

Topic	Page
Managing an abstract plan group	373
Finding abstract plans	377
Managing individual abstract plans	377
Managing all plans in a group	381
Importing and exporting groups of plans	385

Managing an abstract plan group

You can use system procedures to create, drop, rename, and provide information about an abstract plan group.

Creating a group

Use `sp_add_qpgroup` to create and name an abstract plan group. Unless you are using the default capture group, `ap_stdout`, you must create a plan group before you can begin capturing plans. For example, to start saving plans in a group called `dev_plans`, you must create the group, then issue the `set plan dump` command, specifying the group name:

```
sp_add_qpgroup dev_plans
set plan dump dev_plans on
/*SQL queries to capture*/
```

Only a system administrator or database owner can add abstract plan groups. Once a group is created, any user can dump or load plans from the group.

Dropping a group

Use `sp_drop_qpgroup` to drop an abstract plan group.

The following restrictions apply to `sp_drop_qpgroup`:

- Only a system administrator or database owner can drop abstract plan groups.
- You cannot drop a group that contains plans. To remove all plans from a group, use `sp_drop_all_qplans`, specifying the group name.
- You cannot drop the default abstract plan groups `ap_stdin` and `ap_stdout`.

This command drops the `dev_plans` plan group:

```
sp_drop_qpgroup dev_plans
```

Getting information about a group

`sp_help_qpgroup` prints information about an abstract plan group, or about all abstract plan groups in a database.

When you use `sp_help_qpgroup` without a group name, it prints the names of all abstract plan groups, the group IDs, and the number of plans in each group:

```
sp_help_qpgroup
Query plan groups in database 'pubtune'
Group          GID          Plans
-----
ap_stdin       1              0
ap_stdout      2              2
p_prod         4              0
priv_test      8              1
ptest         3             51
ptest2        7            189
```

When you use `sp_help_qpgroup` with a group name, the report provides statistics about plans in the specified group. This example reports on the group `ptest2`:

```
sp_help_qpgroup ptest2
Query plans group 'ptest2', GID 7

Total Rows  Total QueryPlans
```



```

-----
              452              189
sysqueryplans rows consumption, number of query
plans per row count
Rows          Plans
-----
              5              2
              3              68
              2             119
Query plans that use the most sysqueryplans rows
Rows          Plan
-----
              5  1932533918
              5  1964534032
Hashkeys
-----
              123
There is no hash key collision in this group.

```

When reporting on an individual group, `sp_help_qpgroup` reports:

- The total number of abstract plans, and the total number of rows in the `sysqueryplans` table.
- The number of plans that have multiple rows in `sysqueryplans`. The plans are listed in descending order, starting with the plans with the largest number of rows.
- Information about the number of hash keys and hash-key collisions. Abstract plans are associated with queries by a hashing algorithm over the entire query.

When a system administrator or the database owner executes `sp_help_qpgroup`, the procedure reports on all of the plans in the database or in the specified group. When any other user executes `sp_help_qpgroup`, it reports only on plans that he or she owns.

`sp_help_qpgroup` provides several report modes. The report modes are:

Mode	Information returned
full	The number of rows and number of plans in the group, the number of plans that use two or more rows, the number of rows and plan IDs for the longest plans, and number of hash keys, and hash-key collision information. This is the default report mode.
stats	All of the information from the full report, except hash-key information.
hash	The number of rows and number of abstract plans in the group, the number of hash keys, and hash-key collision information.

Mode	Information returned
list	The number of rows and number of abstract plans in the group, and the following information for each query/plan pair: hash key, plan ID, first few characters of the query, and the first few characters of the plan.
queries	The number of rows and number of abstract plans in the group, and the following information for each query: hash key, plan ID, first few characters of the query.
plans	The number of rows and number of abstract plans in the group, and the following information for each plan: hash key, plan ID, first few characters of the plan.
counts	The number of rows and number of abstract plans in the group, and the following information for each plan: number of rows, number of characters, hash key, plan ID, first few characters of the query.

This example shows the output for the counts mode:

```
sp_help_qpgroup ptest1, counts
Query plans group 'ptest1', GID 3

Total Rows  Total QueryPlans
-----
          48              19

Query plans in this group

Rows  Chars  hashkey  id  query
-----
    3    623  1801454852  876530156  select title from titles ...
    3    576  476063777  700529529  select au_lname, au_fname...
    3    513  444226348  652529358  select au1.au_lname, au1....
    3    470  792078608  716529586  select au_lname, au_fname...
    3    430  789259291  684529472  select au1.au_lname, au1....
    3    425  1929666826  668529415  select au_lname, au_fname...
    3    421  169283426  860530099  select title from titles ...
    3    382  571605257  524528902  select pub_name from publ...
    3    355  845230887  764529757  delete salesdetail where ...
    3    347  846937663  796529871  delete salesdetail where ...
    2    379  1400470361  732529643  update titles set price =...
```

Renaming a group

A system administrator or database owner can rename an abstract plan group with `sp_rename_qpgroup`. This example changes the name of the group from `dev_plans` to `prod_plans`:

```
sp_rename_qpgroup dev_plans, prod_plans
```

The new group name cannot be the name of an existing group.

Finding abstract plans

Use `sp_find_qplan` to search both the query text and the plan text to find plans that match a given pattern.

This example finds all plans where the query includes the string “from titles”:

```
sp_find_qplan "%from titles%"
```

This example searches for all abstract plans that perform a table scan:

```
sp_find_qplan "%t_scan%"
```

When a system administrator or database owner executes `sp_find_qplan`, the procedure examines and reports on plans owned by all users. When other users execute the procedure, `sp_find_qplan` searches and reports on only plans that they own.

To search just one abstract plan group, specify the group name. This example searches only the `test_plans` group, finding all plans that use a particular index:

```
sp_find_qplan "%i_scan title_id_ix%", test_plans
```

For each matching plan, `sp_find_qplan` prints the group ID, plan ID, query text, and abstract plan text.

Managing individual abstract plans

You can use system procedures to print the query and text of individual plans, to copy, drop, or compare individual plans, or to change the plan associated with a particular query.

Viewing a plan

Use `sp_help_qplan` to report on individual abstract plans. It provides three types of reports that you can specify: brief, full, and list. The brief report prints only the first 78 characters of the query and plan; use full to see the entire query and plan, or list to display only the first 20 characters of the query and plan.

This example prints the default brief report:

```
gid          sp_help_qplan 588529130
             hashkey      id
-----
             8  1460604254  588529130
query
-----
select min(price) from titles
plan
-----
(plan
  (i_scan type_price titles)
  ()
)
(prop titles
  (parallel ...
```

A system administrator or database owner can use `sp_help_qplan` to report on any plan in the database. Other users can view only the plans that they own.

`sp_help_qpgroup` reports on all plans in a group. See “Getting information about a group” on page 374.

Copying a plan to another group

Use `sp_copy_qplan` to copy an abstract plan from one group to another existing group. This example copies the plan with plan ID 316528161 from its current group to the `prod_plans` group:

```
sp_copy_qplan 316528161, prod_plans
```

`sp_copy_qplan` verifies that the query does not already exist in the destination group. If a possible conflict exists, `sp_copy_qplan` runs `sp_cmp_qplans` to check plans in the destination group. In addition to the message printed by `sp_cmp_qplans`, `sp_copy_qplan` prints messages when:

- The query and plan you are trying to copy already exists in the destination group
- Another plan in the group has the same user ID and hash key
- Another plan in the group has the same hash key, but the queries are different

If there is a hash-key collision, the plan is copied. If the plan already exists in the destination group or if it would give an association key collision, the plan is not copied. The messages printed by `sp_copy_qplan` contain the plan ID of the plan in the destination group, so you can use `sp_help_qplan` to check the query and plan.

A system administrator or the database owner can copy any abstract plan. Other users can copy only plans that they own. The original plan and group are not affected by `sp_copy_qplan`. The copied plan is assigned a new plan ID, the ID of the destination group, and the user ID of the user who ran the query that generated the plan.

Dropping an individual abstract plan

Use `sp_drop_qplan` to drop individual abstract plans. This example drops the specified plan:

```
sp_drop_qplan 588529130
```

A system administrator or database owner can drop any abstract plan in the database. Other users can drop only plans that they own.

To find abstract plan IDs, use `sp_find_qplan` to search for plans using a pattern from the query or plan, or use `sp_help_qpgroup` to list the plans in a group.

Comparing two abstract plans

Given two plan IDs, `sp_cmp_qplans` compares two abstract plans and the associated queries. For example:

```
sp_cmp_qplans 588529130, 1932533918
```

`sp_cmp_qplans` prints one message reporting the comparison of the query, and a second message about the plan, as follows:

- For the two queries, one of:

- The queries are the same.
- The queries are different.
- The queries are different but have the same hash key.
- For the plans:
 - The query plans are the same.
 - The query plans are different.

This example compares two plans where the queries and plans both match:

```
sp_cmp_qplans 411252620, 1383780087
The queries are the same.
The query plans are the same.
```

This example compares two plans where the queries match, but the plans are different:

```
sp_cmp_qplans 2091258605, 647777465
The queries are the same.
The query plans are different.
```

`sp_cmp_qplans` returns a status value showing the results of the comparison.

Table 14-1: Return status values for `sp_cmp_qplans`

Return value	Meaning
0	The query text and abstract plans are the same.
+1	The queries and hash keys are different.
+2	The queries are different, but the hash keys are the same.
+10	The abstract plans are different.
100	One or both of the plan IDs does not exist.

A system administrator or database owner can compare any two abstract plans in the database. Other users can compare only plans that they own.

Changing an existing plan

Use `sp_set_qplan` to change the abstract plan for an existing plan ID without changing the ID or the query text. You can use `sp_set_qplan` only when the plan text is 255 or fewer characters.

```
sp_set_qplan 588529130, "(i_scan title_ix titles)"
```

A system administrator or database owner can change the abstract plan for any saved query. Other users can modify only plans that they own.

When you execute `sp_set_qplan`, the abstract plan is not checked against the query text to determine whether the new plan is valid for the query, or whether the tables and indexes exist. To test the validity of the plan, execute the associated query.

You can also use `create plan` and the `plan` clause to specify the abstract plan for a query. See “Creating plans using SQL” on page 331.

Managing all plans in a group

You can use system procedures to copy all plans in one abstract plan group to another group, compare all abstract plans in two groups and reports, and drop all abstract plans in a group.

Copying all plans in a group

Use `sp_copy_all_qplans` to copy all of the plans in one abstract plan group to another group. This example copies all of the plans from the `test_plans` group to the `helpful_plans` group:

```
sp_copy_all_qplans test_plans, helpful_plans
```

The `helpful_plans` group must exist before you execute `sp_copy_all_qplans`. It can contain other plans.

`sp_copy_all_qplans` copies each plan in the group by executing `sp_copy_qplan`, so copying a plan may fail for the same reasons that `sp_copy_qplan` might fail. See “Comparing two abstract plans” on page 379.

Each plan is copied as a separate transaction, and failure to copy any single plan does not cause `sp_copy_all_qplans` to fail. If `sp_copy_all_qplans` fails for any reason, and has to be restarted, you see a set of messages for the plans that have already been successfully copied, telling you that they exist in the destination group.

A new plan ID is assigned to each copied plan. The copied plans have the original user's ID. To copy abstract plans and assign new user IDs, you must use `sp_export_qpgroup` and `sp_import_qpgroup`. See “Importing and exporting groups of plans” on page 385.

A system administrator or database owner can copy all plans in the database. Other users can copy only plans that they own.

Comparing all plans in a group

Use `sp_cmp_all_qplans` to compare all abstract plans in two groups and reports:

- The number of plans that are the same in both groups
- The number of plans that have the same association key, but different abstract plans
- The number of plans that are present in one group, but not the other

This example compares the plans in `ap_stdout` and `ap_stdin`:

```
sp_cmp_all_qplans ap_stdout, ap_stdin
If the two query plans groups are large, this might take some
time.
Query plans that are the same
count
-----
      338
Different query plans that have the same association key
count
-----
      25
Query plans present only in group 'ap_stdout' :
count
-----
      0
Query plans present only in group 'ap_stdin' :
count
-----
      1
```


With the additional specification of a report-mode parameter, `sp_cmp_all_qplans` provides detailed information, including the IDs, queries, and abstract plans of the queries in the groups. The mode parameter lets you get the detailed information for all plans, or just those with specific types of differences.

Table 14-2: Report modes for `sp_cmp_all_qplans`

Mode	Reported information
counts	The counts of plans that are the same, plans that have the same association key, but different groups, and plans that exist in one group, but not the other. This is the default report mode.
brief	The information provided by counts, plus the IDs of the abstract plans in each group where the plans are different, but the association key is the same, and the IDs of plans that are in one group, but not in the other.
same	All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans match.
diff	All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans are different.
first	All counts, plus the IDs, queries, and plans for all abstract plans that are in the first plan group, but not in the second plan group.
second	All counts, plus the IDs, queries, and plans for all abstract plans that are in the second plan group, but not in the first plan group.
offending	All counts, plus the IDs, queries, and plans for all abstract plans that have different association keys or that do not exist in both groups. This is the combination of the diff, first, and second modes.
full	All counts, plus the IDs, queries, and plans for all abstract plans. This is the combination of same and offending modes.

This example shows the brief report mode:

```
sp_cmp_all_qplans ptest1, ptest2, brief
```

If the two query plans groups are large, this might take some time.
Query plans that are the same

```
count
-----
      39
Different query plans that have the same association key
```

```
count
-----
      4

      ptest1    ptest2

      id1       id2
-----
```

```
764529757 1580532664
780529814 1596532721
796529871 1612532778
908530270 1724533177
```

Query plans present only in group 'ptest1' :

```
count
-----
          3
```

```
id
-----
524528902
1292531638
1308531695
```

Query plans present only in group 'ptest2' :

```
count
-----
          1
```

```
id
-----
2108534545
```

Dropping all abstract plans in a group

Use `sp_drop_all_qplans` to drop all abstract plans in a group. This example drops all abstract plans in the `dev_plans` group:

```
sp_drop_all_qplans dev_plans
```

When a system administrator or the database owner executes `sp_drop_all_qplans`, all plans belonging to all users are dropped from the specified group. When another user executes this procedure, it affects only the plans owned by that user.

Importing and exporting groups of plans

Use `sp_export_qpgroup` and `sp_import_qpgroup` to copy groups of plans between `sysqueryplans` and a user table. This allows a system administrator or database owner to:

- Copy abstract plans from one database to another on the same server
- Create a table that can be copied out of the current server with `bcp`, and copied into another server
- Assign different user IDs to existing plans in the same database

Exporting plans to a user table

Use `sp_export_qpgroup` to copy all plans for a specific user from an abstract plan group to a user table. This example copies plans owned by the database owner (`dbo`) from the `fast_plans` group, creating a table called `transfer`:

```
sp_export_qpgroup dbo, fast_plans, transfer
```

`sp_export_qpgroup` uses `select...into` to create a table with the same columns and datatypes as `sysqueryplans`. If you do not have the `select into/bulkcopy/plsort` option enabled in the database, you can specify the name of another database. This command creates the export table in `tempdb`:

```
sp_export_qpgroup mary, ap_stdout, "tempdb..mplans"
```

The table can be copied out using `bcp`, and copied into a table on another server. The plans can also be imported to `sysqueryplans` in another database on the same server, or the plans can be imported into `sysqueryplans` in the same database, with a different group name or user ID.

Importing plans from a user table

Use `sp_import_qpgroup` to copy plans from tables created by `sp_export_qpgroup` into a group in `sysqueryplans`. This example copies the plans from the table `tempdb.mplans` into `ap_stdin`, assigning the user ID for the database owner:

```
sp_import_qpgroup "tempdb..mplans", dbo, ap_stdin
```

You cannot copy plans into a group that already contains plans for the specified user.

Index

A

- abstract plan cache** configuration parameter 331
- abstract plan derived tables 339
- abstract plan dump** configuration parameter 331
- abstract plan groups
 - adding 373
 - creating 373
 - dropping 374
 - exporting 385
 - importing 385
 - information about 374
 - overview of use 318
 - plan association and 318
 - plan capture and 318
 - procedures for managing 373–385
- abstract plan load** configuration parameter 331
- abstract plan replace** configuration parameter 331
- abstract plans
 - comparing 379
 - copying 378
 - finding 377
 - information about 378
 - pattern matching 377
 - viewing with **sp_help_qplan** 378
- accessing
 - query processing metrics 280
- adding
 - abstract plan groups 373
 - statistics for unindexed columns 290
- adding statistics 290
- adjustment
 - managing runtime 200
 - recognizing runtime 200
 - reducing runtime 201
 - runtime 199
- ALS
 - user log cache 243
- ALS. see Asynchronous Log Service 241
- application design

- cursors and 277
- index specification 235
- associating queries with plans
 - plan groups and 318
 - session-level 323
- association key
 - defined 319
 - plan association and 319
 - sp_cmp_all_qplans** and 382
 - sp_copy_qplan** and 379
- attribute-insensitive operation
 - parallelism 155
- attribute-sensitive operation
 - parallelism 169
- automatically
 - update statistics** 297

B

- buffers unavailable 238

C

- capturing plans
 - session-level 322
- cheap direct updates 28
- clearing query processing metrics 284
- close** command
 - memory and 266
- close on endtran** option, **set** 277
- clustered indexes
 - prefetch and 237
- column-level statistics 300
 - generating the **update statistics** 305
 - truncate table** and 301
 - update statistics** and 300
- comparing abstract plans 379
- composite indexes

- update index statistics** and 305
- compute by processing 88
- concurrency optimization
 - for small tables 259
- concurrency optimization threshold
 - deadlocks and 259
- connections
 - cursors and 278
- controlling parallelism at session level 139
- controlling parallelism for a query 141
- converted search arguments 6
- copying
 - abstract plans 378
 - plan groups 381
 - plans 378, 381
- covered queries
 - specifying cache strategy for 239
- creating
 - abstract plan groups 373
 - column statistics 302
 - search arguments 18
- cursor rows** option, **set** 277
- cursors
 - execute 266
 - Halloween problem 268
 - indexes and 267
 - isolation levels and 273
 - locking and 264
 - modes 266
 - multiple 278
 - read-only 266
 - stored procedures and 266
 - updatable 266

D

- data modification update modes 27
- data pages prefetching 237
- datachange** function
 - statistics 295
- datatypes
 - join** 13
- deadlocks
 - concurrency optimization threshold settings 259
 - table scans and 259

- deallocate cursor** command
 - memory and 266
- debugging aids
 - set forceplan on** 232
- declare cursor** command
 - memory and 265
- default settings
 - number of tables optimized 234
- deferred
 - index updates 31
 - updates 30
- degree
 - setting max parallel 136
- delete operations
 - joins and update mode 30
 - update mode in joins 30
- delete** operator 62, 195
- delete statistics** command
 - managing statistics and 310
- deleting plans 379, 384
- density **join** 12
- derived tables
 - abstract plan derived tables 339
 - SQL 19
 - SQL derived tables 339
- differing parallel query results 144
- direct updates 27
 - cheap 28
 - expensive 29
 - in-place 27
 - joins and 30
- discontinued trace commands
 - XML 119
- drop index** command
 - statistics and 310
- dropping
 - abstract plan groups 374
 - indexes specified with **index** 235
 - plans 379, 384
- duplication
 - update performance effect of 30
- dynamic parameters, analyzing 122

E

- elimination partition 197
- emit operator 52
- enable parallelism 135
- engine query execution 20
- equi-join**, transitive closure 7
- exceptions, optimization goals 16
- exchange**
 - operator 148
 - worker process mode 151
- exchange**, pipemanagement 149
- execute cursors
 - memory use of 266
- executing
 - query processing metrics 280
- execution
 - preventing with **set noexec on** 37
- exists check** mode 327
- expensive direct updates 29
- exporting plan groups 385
- expressions, **join** 14

F

- factors, analyzed for optimization 5
- fetching cursors, memory and 266
- finding abstract plans 377
- fixed-length columns
 - indexes and update modes 36
- for update** option, **declare cursor**
 - optimizing and 276
- forceplan** option, **set** 232
- from** table 54
- function
 - datachange** , statistics 295

G

- goals, optimization 15
- goals, optimization exceptions 16
- group sorted agg
 - operator 85
- group sorted agg** operator 85
- GroupSorted (Distinct)** operator 81

H

- Halloween problem
 - cursors and 268
- hash join
 - operator 74
- hash union
 - operator 91
- hash vector aggregate
 - operator 86
- hash-based table scan 157
- HashDistinctOp** operator 83
- histograms
 - join** 12
 - steps, number of 306

I

- I/O
 - direct updates and 27
 - prefetch** keyword 236
 - range queries and 236
 - specifying size in queries 236
 - update operations and 29
- IDENTITY columns
 - cursors and 268
- importing abstract plan groups 385
- index scan 159
 - clustered 163
 - clustered, partitioned table 163
 - covered using nonclustered global 162
 - global nonclustered 159
 - nonclustered, partitioned table 163
 - noncovered, global nonclustered 159
- indexes
 - cursors using 267
 - large I/O for 236
 - search arguments 10
 - specifying for queries 234
 - update index statistics** on 305
 - update modes and 35
 - update operations and 28, 29
 - update statistics** on 305
- in-place updates 27
- insert**
 - operator 62, 195

Index

isolation levels
 cursors 273

J

Job Scheduler
 update statistics 297
join 12
 both tables with useless partitioning 173
 number of tables considered by optimizer 233
 operator 69
 outer 180
 parallelism 169
 parallelism, one table with useful partitioning 171
 parallelism, replicated 175
 parallelism, tables with same useful partitioning 169
 semi 180
 serial 178
 table order in 232
 update mode and 30
 updates using 28, 29, 30
join
 density 12
 expressions 14
 histograms 12
 mixed data types 13
 or predicates 14
 ordering 14
joins
 updates using 29
jtc option, **set** 246

K

keys, index
 update operations on 28

L

large I/O
 index leaf pages 236
Lava
 operators 23

 query engine 21
 query execution 25
 query plans 21
lightweight procedures and dynamic SQL statements
 122
locking
 statistics 307
log scan 59
LRU replacement strategy
 specifying 240

M

maintenance tasks
 forced indexes 235
 forceplan checking 232
maintenance, statistics 300
max repartition degree
 setting 137
max resource granularity
 setting 136
memory
 cursors and 264
merge join operator 71
merge union operator 91
messages, dropped index 235
minor columns
 update index statistics and 305
modifying abstract plans 380
MRU replacement strategy, disabling 241

N

names
 index clause and 235
 index prefetch and 237
nary nested loop join operator 76
nested loop join 70
networks
 cursor activity of 272
nonequality, operators 11
nonleading columns sort statistics 308
null columns
 optimizing updates on 35

number (quantity of)
 cursor rows 277
 tables considered by optimizer 233

O

object sizes
 tuning 19

open command
 memory and 266

operations
insert, delete, update 195

operator
delete operator 62
exchange 148
 group sorted agg 85
group sorted agg 85
 hash join 74
 hash union 91
 hash vector aggregate 86
 merge union 91
 nary nested loop join 76
 remote scan 99
 restrict 94
 RID join 100
 scan 52
 scroll 99
 sequencer 97
 sort 94
 sqfilter 102
 store 95
 text delete 63
 union all 90
update operator 62

operator ,**insert** operator 62

operator, emit 52

operator, merge join 71

operators
GroupSorted (Distinct) 81
HashDistinctOp 83
 Lava 23
 optimization 4
 query plans 52
ScalarAggOp 93
SortOp (Distinct) 82
 vector aggregation 84

operators, nonequality 11

optimization
 additional paths 9
 cursors 266
 example search arguments 11
 limit time optimizing query 16
 operators 4
 predicate transformation 9
 query transformation 6
 techniques 4

optimization goals, exceptions 16

optimization problems 17

optimization, factors analyzed 5

optimization, goals 15

optimizer 34–36
 overriding 221
 query 3
 updates and 34

optimizer diagnostic utility 229–230
 configuring Adaptive Server 230
 run **sp_opt_querystats** 230
sp_opt_querystats 229–??
sp_opt_querystats , information provided 229

option
set rowcount 145

or list 53

or predicates
join 14

OR strategy, cursors and 275

order, tables in a join 232

ordering, **join** 14

output
 statement 44
 XML diagnostic 112

overhead
 cursors 272
 deferred updates 30

P

pages, data
 prefetch and 237

parallel
 query execution model 148

- query plans 146
 - query processing 133
 - setting max degree 136
 - setting max resource granularity 136
 - table scan 156
 - union all 167
 - parallel degree, setting max scan 138
 - parallel processing
 - query 134
 - parallelism 17
 - attribute-insensitive operation 155
 - attribute-sensitive operation 169
 - controlling at session level 139
 - controlling for a query 141
 - distinct vector aggregation 184
 - in-partitioned vector aggregation 180
 - join 169
 - join, both tables with useless partitioning 173
 - join, one table with useful partitioning 171
 - join, replicated 175
 - join, tables with same useful partitioning 169
 - outer joins 180
 - query with IN list 184
 - query with OR clause 186
 - query with order by clause 188
 - reformatting 176
 - repartitioned vector aggregation 181
 - semi joins 180
 - serial join 178
 - serial vector aggregation 183
 - setting number of worker processes 135
 - SQL operations 154
 - table scan 155
 - two-phased vector aggregation 182
 - vector aggregation 180
 - parallelism, enable 135
 - partial plans
 - specifying with **create plan** 317
 - partition
 - elimination 197
 - skew 198
 - table scan 157
 - performance
 - number of tables considered by optimizer 234
 - permissions
 - XML 122
 - pipe management, **exchange** 149
 - plan dump** option, **set** 321
 - plan groups
 - adding 373
 - copying 381
 - copying to a table 385
 - creating 373
 - dropping 374
 - dropping all plans in 384
 - exporting 385
 - information about 374
 - overview of use 318
 - plan association and 318
 - plan capture and 318
 - reports 374
 - plan load** option, **set** 323
 - plan replace** option, **set** 323
 - plans
 - changing 380
 - comparing 379
 - copying 378, 381
 - deleting 384
 - dropping 379, 384
 - finding 377
 - modifying 380
 - searching for 377
 - predicate transformation 9
 - prefetch
 - data pages 237
 - disabling 239
 - enabling 239
 - queries 236
 - sp_cachestrategy** 241
 - prefetch** keyword, I/O size and 236
 - problems optimizing queries 17
 - process_limit_action** 200
- ## Q
- QP metrics. *See* query processing metrics
 - queries
 - execution settings 37
 - specifying I/O size 236
 - specifying index for 234
 - queries, problems optimizing 17

- query
 - Lava execution 25
 - limit optimizing time 16
 - not run in parallel 199
 - optimizer 3
 - OR clause 186
 - parallel execution model 148
 - parallel processing 134
 - select-into clause 192
 - set local variables 145
 - with IN list 184
 - with order by clause 188
 - query analysis 34–36
 - dynamic parameters 122
 - showplan** and 37
 - sp_cachestrategy** 241
 - query optimization 111
 - query plans 47
 - Lava 21
 - operators 52
 - parallel 146
 - suboptimal 235
 - updatable cursors and 275
 - query processing
 - parallel 133
 - query processing metrics
 - accessing 280
 - clearing 284
 - executing 280
 - sysquerymetrics view 282
 - using 282
 - query, execution engine 20
- ## R
- range queries, large I/O for 236
 - read-only cursors 266
 - indexes and 267
 - locking and 272
 - reduce **update statistics** impact 309
 - referential integrity
 - constraints 65
 - update operations and 28
 - updates using 30
 - reformatting parallelism 176
 - remote scan operator 99
 - replication, update operations and 28
 - reports
 - cache strategy 241
 - plan groups 374
 - restrict operator 94
 - results, differing parallel query 144
 - RID join operator 100
 - RID scan 57
 - row counts statistics, inaccurate 311
 - row ID (RID) update operations and 28
 - runtime
 - adjustment 199
 - managing adjustment 200
 - recognizing adjustment 200
 - reducing adjustments 201
- ## S
- sampling
 - statistics 293
 - use for updating statistics 293
 - scalar aggregation
 - serial 166
 - two phased 165
 - ScalarAggOp** operator 93
 - scan
 - clustered index 163
 - clustered index on partitioned tables 163
 - index 159
 - index global nonclustered 159
 - index noncovered of global nonclustered 159
 - index, covered use nonclustered global 162
 - local indexes 163
 - nonclustered, partitioned table 163
 - operator 52
 - scan types statistics 307
 - scroll operator 99
 - search arguments
 - creating 18
 - example of optimization 11
 - indexes 10
 - transitive closure 7
 - search arguments, converted 6
 - searching for abstract plans 377

- select** command
 - specifying index 234
- select-into query 192
- sequencer
 - operator 97
- serial
 - scalar aggregation 166
 - union all 168
- serial table scan 155
- set
 - local variables 145
 - XML command 112
- set** command
 - examples 140
 - forceplan** 232
 - jtc** 246
 - plan dump** 321
 - plan exists** 327
 - plan load** 323
 - plan replace** 323
- set plan dump** command 322
- set plan exists check** 327
- set plan load** command 323
- set plan replace** command 323
- set rowcount**
 - option 145
- set theory operations
 - compared to row-oriented programming 262
- setting
 - mac parallel degree 136
 - max repartition degree 137
 - max resource granularity 136
 - max scan parallel degree 138
 - number of worker processes 135
- shared** keyword, cursors and 267
- shared locks
 - read-only cursors 267
- showplan**
 - statement level output 44
 - using 37, 201
- skew, partition 198
- sort
 - operator 94
 - statistics, unindexed columns 308
- sort requirements
 - statistics 307
- SortOp (Distinct)** operator 82
- sp_add_qpgroup** system procedure 373
- sp_cachestrategy** system procedure 241
- sp_chgattribute** system procedure
 - concurrency_opt_threshold** 259
- sp_cmp_qplans** system procedure 379
- sp_copy_all_qplans** system procedure 381
- sp_copy_qplan** system procedure 378
- sp_drop_all_qplans** system procedure 384
- sp_drop_qpgroup** system procedure 374
- sp_drop_qplan** system procedure 379
- sp_export_qpgroup** system procedure 385
- sp_find_qplan** system procedure 377
- sp_help_qpgroup** system procedure 374
- sp_help_qplan** system procedure 378
- sp_import_qpgroup** system procedure 385
- sp_opt_querystats**
 - configuring Adaptive Server 230
 - running 230
 - system procedure 229
 - truncating output 230
- sp_set_qplan** system procedure 380
- speed (server)
 - cheap direct updates 28
 - deferred index deletes 33
 - deferred updates 30
 - direct updates 27
 - expensive direct updates 29
 - in-place updates 27
 - updates 27
- sqfilter, operator 102
- SQL
 - parallelism 154
- SQL derived tables 339
- SQL standards
 - cursors and 262
- SQL tables
 - derived 19
- statement level output 44
- statistics
 - adding for unindexed columns 290
 - column-level 300, 302, 305
 - creating column statistics 302
 - datachange** function 295
 - deleting table and column with **delete statistics** 310

- drop index** and 300
 - getting additional 303
 - locking 307
 - sampling 293
 - scan types 307
 - sort requirements 307
 - sorts for unindexed columns 308
- truncate table** and 301
- update statistics** 291
- update statistics** automatically 297
- updating 290, 302
- using 287
- using Job Scheduler 297
- statistics** clause, **create index** command 301
- statisticmaintenance 300
- statisticssorts, nonleading columns 308
- steps
 - deferred updates 30
 - direct updates 27
- store
 - operator 95
- stored procedures
 - cursors within 270
- subqueries 189
- sysquerymetrics view
 - query processing metrics 282

T

- table count** option, **set** 233
- table scan
 - forcing 234
 - hash-based 157
 - parallel 156
 - parallelism 155
 - partition-based 157
 - serial 155
- techniques, optimization 4
- testing, index forcing 235
- text delete, operator 63
- transaction logs, update operation and 27
- transformation
 - predicate 9
- transformations
 - query optimization 6

- transitive closure
 - equi-join** 7
 - search arguments 7
- triggers
 - update mode and 34
 - update operations and 28
- truncate table** command
 - column-level statistics and 301
- tuning
 - according to object size 19
 - advanced techniques for 221–259
 - range queries 235
- two phased scalar aggregation 165

U

- unindexed columns 290
- union all
 - operator 90
 - parallel 167
 - serial 168
- union** operator
 - cursors and 275
- unique indexes
 - update modes and 35
- update** 195
- update all statistics** 302
- update all statistics** command 306
- update cursors 266
- update index statistics 302, 305, 308
- update locks
 - cursors and 267
- update modes
 - cheap direct 28
 - deferred 30
 - deferred index 31
 - direct 30
 - expensive direct 29
 - indexing and 35
 - in-place 27
 - joins and 30
 - optimizing for 34
 - triggers and 34
- update operations 27
- update** operator 62

Index

- update statistics** 291
 - column-level 305
 - column-level statistics 305
 - managing statistics and 300
 - with consumers** clause 309
- updating statistics 290, 293, 302
 - use sampling 293
- user IDs
 - changing with **sp_import_qpgroup** 385
- user log cache, in ALS 243

V

- variables, setting local 145
- vector aggregation 180
 - distinct 184
 - in-partitioned 180
 - repartitioned 181
 - serial 183
 - two-phased 182
- vector aggregation operators 84
- view
 - sysquerymetrics, query processing metrics 282

W

- with statistics** clause, **create index** command 301
- worker process mode
 - exchange** 151
- worker processes
 - setting number 135

X

- XML
 - diagnostic output 112
 - discontinued trace commands 119
 - permissions 122
 - set** 112