New Features for Adaptive Server® Enterprise 12.5.4 ESD 4

Document ID: DC00630-01-1254-01

Last revised: December 14, 2006

Adaptive Server® Enterprise version 12.5.4, ESD 4 includes these features:

Virtually-hashed tables

You can perform hash-based index scans using nonclustered indexes or clustered indexes on data-only-locked tables. During this scan, each worker process navigates the higher levels of the index and reads the leaf-level pages of the index. Each worker process then hashes on either the data page ID or the key value in a separate hash table to determine which data pages or data rows to process.

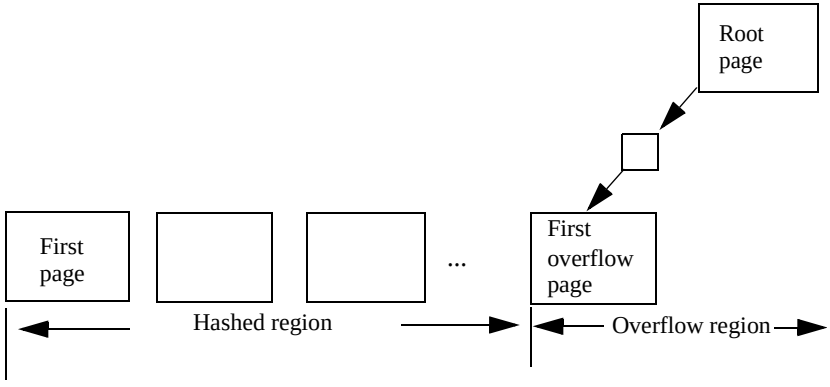
A virtually-hashed table can be a more efficient way to organize a table because it does not require a separate hash table. Instead, it stores the rows so that, using the hash key, the query processor can determine the row ID (based on the row's ordinal number) and the location of the data. Because it does not use a separate hash table to hold the information, it is called a "virtually" hashed table.

For systems that must make more efficient use of their central-processing unit (CPU), the virtually hashed table is a good option.

It is expensive for tables used for look-ups or for tables whose row position does not change, to use a clustered or nonclustered index. With recent advancements in level-2 and level-3 (L2 and L3) CPU architectures, you must utilize the cache to take advantage of the real CPU computing power. If you do not utilize the cache, the CPU spends needless cycles waiting for available memory. For clustered or non-clustered indexes, the server misses rows every time it accesses the index-level search, which consumes many CPU cycles. Virtually-hashed tables access row-location patterns by computing the hash-key value instead of performing a search.

Structure of a virtually-hashed table

A virtually-hashed table contains two regions–the "hashed" region and the "overflow" region. The hashed region stores the hashed rows, and the overflow region stores the remaining rows. You can access the overflow region with a regular clustered index scan using a B-tree clustered index.



The hashed region, root page, and the first overflow page of a virtually-hashed table are allocated when you create the table. SYINDEXES.indroot is the root page for the overflow clustered region. The first leaf page under this page is the first overflow page. SYSINDEXES.indfirst points to the first data page, so a table scan starts at the beginning of the table and scans the entire table.

Creating a virtually-hashed table

To create a virtually-hashed table, specify the maximum value for the hash region. This is the partial syntax for create table; the new parameters are shown in bold:

```
create table [database.[owner].]table_name
. . .
        | {unique | primary key}
        using clustered
        (column_name [asc | desc] [{, column_name [asc | desc]}...])=
        (hash_factor [{, hash_factor}...])
                with max num_hash_values key
```

Where:

- using clustered – indicates you are creating a virtually-hashed table. The list of columns are treated as key columns for this table.
- *column_name* [asc | desc] – because rows are placed based on their hash function, you cannot use [asc | desc] for the hash region . If you provide an order for the key columns of virtually hashed tables, it is used only in the overflow clustered region.
- *hash_factor* – required for the hash function for virtually-hashed tables. For the hash function, a hash factor is required for every key column. These factors are used with key values to generate hash value for a particular row.
- with max *num_hash_values* key – the maximum number of hash values that you can use. Defines the upper bound on the output of this hash function.

Note Virtually-hashed tables have these restrictions:

- Virtually-hashed tables must have unique rows. Virtually-hashed tables do not allow multiple rows with the same key column values because Adaptive Server cannot keep one row in the hash region and another with the same key column value in the overflow clustered region.
- You must create each virtually-hashed table on an exclusive segment.

Determining values for *hash_factor*

You can keep the hash factor for the first key as 1. The hash factor for all the remaining key columns is greater than the maximum value of the previous key allowed in the hash region multiplied by its hash factor.

Adaptive Server allows tables with hash factors greater than 1 for the first key column to have fewer rows on a page. For example, if a table has a hash factor of 5 for the first key column, after every row in a page, space for the next four rows is kept empty. To support this, Adaptive Server requires five times the amount of table space.

If the value of a key column is greater than or equal to the hash factor of the next key column, the current row is inserted in the overflow clustered region to avoid collisions in the hash region.

For example, t is a virtually-hashed table with key columns id and age, and corresponding hash factors of (10,1). Because the hash value for rows (5, 5) and (2, 35) is 55, this may result in a hash collision.

However, because the value 35 is greater than or equal to 10 (the hash factor for the next key column, id), Adaptive Server stores the second row in the overflow clustered region, avoiding collisions in the hash region.

In another example, if u is a virtually-hashed table with a primary index and hash factors of (id1, id2, id3) = (125, 25, 5) and a *max hash_value* of 200:

- Row (1,1,1) has a hash value of 155 and is stored in the hash region.
- Row (2,0,0) has a hash value 250 and is stored in overflow clustered region.
- Row (0,0,6) has a hash factor of 6 x 5, which is greater than or equal to 25, so it is stored in the overflow clustered region.
- Row (0,7,0) has a hash factor of 7 x 25, which is greater than or equal to 125, so it is stored in the overflow clustered region

Examples

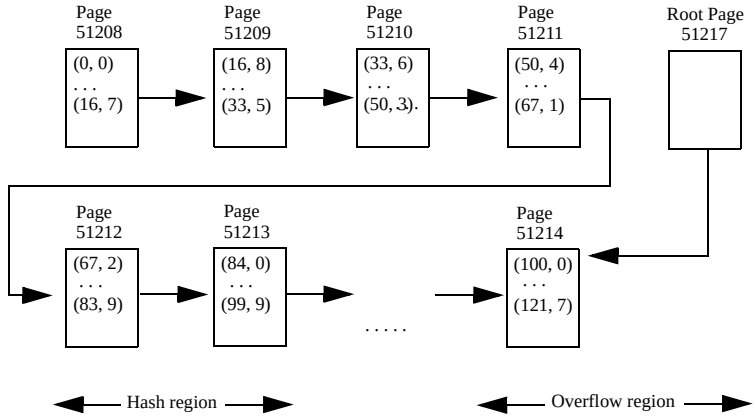
This example creates a virtually-hashed table named orders on the pubs2 database on the order_seg segment:

```
create table orders(
id int,
age int,
primary key using clustered (id,age) = (10,1) with max
1000 key)
on order_seg
```

The layout for the data is:

- The order_seg segment starts on page ID 51200.
- The ID for the first data object allocation map (OAM) page is 51201.
- The maximum rows per page is 168.

- The row size is 10.
- The root index page of the overflow clustered region is 51217.



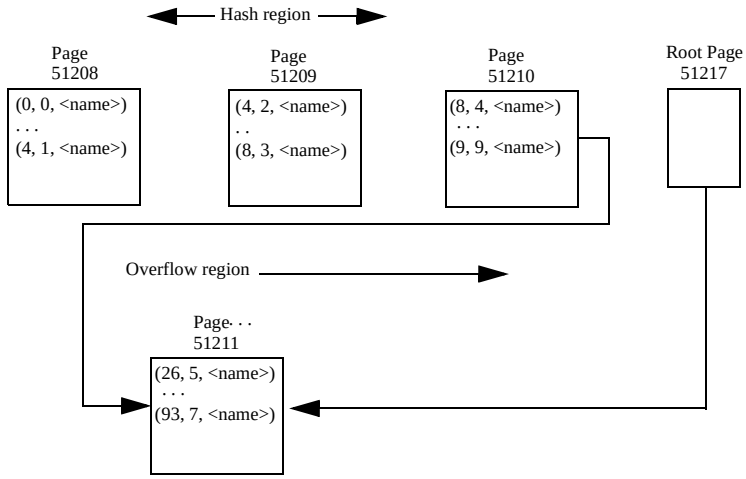
This example creates a virtually-hashed table named orders on the pubs2 database on the order_seg segment:

```
create table orders(
id int default NULL,
age int,
primary key using clustered (id,age) = (10,1) with max
100 key,
name varchar(30)
)
on order_seg
```

The layout for the data is:

- The order_seg segment starts on page ID 51200.
- The ID for the first data OAM page is 51201.
- The maximum rows per page is 42.
- The row size is 45.

- The root index page of the overflow clustered region is 51217.



Limitations for virtually-hashed tables

These are the limittions for virtually-hashed tables:

- truncate table is not supported. Use delete from table *table_name* instead.
- SQL92 does not allow two unique constraints on a relation to have the same key columns. However, the primary key clause for a virtually-hashed table is not a standard unique constraint, so you can declare a separate unique constraint with the same key columns as the virtually-hashed keys.
- Because you cannot create a virtually-hashed clustered index after you create a table, You also cannot drop a virtually-hashed clustered index.
- You must create a virtually-hashed table on an exclusive segment. You cannot share disk devices you assign to the segments for creating a virtually-hashed table with other segments.
- You cannot create two virtually-hashed tables on the same exclusive segment. Adaptive Server supports 32 different segments per database. Three segments are reserved for the default, system and log segments, so the maximum number of virtually-hashed tables per database is 29.
- You cannot use the alter table or drop clustered index commands on virtually-hashed tables.
- Virtually-hashed tables must use all-pages locking (APL).

- The key columns and hash factors of a virtually-hashed table must use the int datatype.
- You cannot include text or image columns in virtually-hashed tables, or columns with datatypes based on the text or image datatypes.

Do not create virtually-hashed tables that :

- Have frequent inserts and updates
- Use frequent table scans

Changes to commands

The changes to the create table command are documented in "Creating a virtually-hashed table" on page 3. Changes to other commands include:

- dbcc checktable – in addition to the regular checks it performs, checktable verifies that the preallocation performed during table creation is correct:
  - The number of pages preallocated matches the total number of data pages that must be allocated for the specified *max hash key* value.
  - The data pages are not preallocated in an extent where the preallocation scheme specifies that only OAM pages are allowed.
  - The OAM pages are allocated only in the first extent of an allocation unit (AU).
- dbcc checkstorage – reports a soft fault if any data page that is not the first data page is empty for non-hashed tables. However, dbcc checkstorage does not report this soft fault for the hashed region of a virtually-hashed table. Any data page in the hashed region of a virtually-hashed table can be empty.

Changes to the query processor

The query processor uses a virtually-hashed index only if you include search arguements that include a equality qualifier (for example, `where id=2`) on all key columns. If the query processor uses the virtually-hashed index, it includes a line similar to this in the showplan output:

```
Using Virtually Hashed Index.
```

The query processor includes lines similar to this in the index selection output if it selects a virtually-hashed index:

```
Unique virtually hashed index found, returns 1 row, 1
pages
```

Changes to monitor counters

This monitor counter has been added for virtually-hashed tables:

- am_srch_hashindex – counts the number of times a search is done using a hash index.

Changes to system procedures

These are the changes that have been made to system procedures to support virtually-hashed tables:

- sp_addsegment – you cannot create a segment on a device that already has an exclusive segment. If you attempt to do so, you see an error message similar to:

  ```
  A segment with a virtually hashed table exists on
  device orders_dat.
  ```

- sp_extendsegment – you cannot extend a segment on a device that already has an exclusive segment, and you cannot extend an exclusive segment on a device that has another segment.

  For example, if you attempt to extend segment orders_seg on a device *orders.dat*, which already has an exclusive segment, you see an error message similar to:

  ```
  A segment with a virtually hashed table exists on
  device orders.dat.
  ```

  If you attempt to extend exclusive segment orders_seg on device *orders.dat*, which has other segments, you see an error message similar to:

  ```
  You cannot extend a segment with a virtually hashed
  table on device orders.dat, because this device has
  other segments.
  ```

- sp_placeobject – you cannot use sp_placeobject on a virtually-hashed table, and you cannot place other objects on an exclusive segment. For example, if you attempt to place a virtually-hashed table named orders on a segment, you see an error message similar to:

  ```
  sp_placeobject is not allowed for orders, as it is a
  virtually hashed table.
  ```

  If you attempt to place object t on segment myseg, which contains a virtually-hashed table, you see an error message similar to:

  ```
  Segment myseg has a virtually hashed table;
  therefore, you cannot place an object on this
  segment.
  ```

- sp_chgattribute – does not allow you to change attributes for virtually-hashed tables. For example, if you attempt to change the attributes for table order_line (a virtually-hashed table) like this:

  ```
  sp_chgattribute 'order_line', 'exp_row_size', 1
  ```

  Adaptive Server issues an error message similar to:

  ```
  sp_chgattribute is not allowed for order_line, as it
  is a virtually hashed table.
  ```

- sp_help – for virtually-hashed table, reports:

  - That a table is virtually-hashed with this message:

    ```
    Object is Virtually Hashed
    ```

  - The hash_key_factors for the table with a message using this syntax:

    ```
    column_1:hash_factor_1,
    column_2:hash_factor_2...,
    max_hash_key=max_hash_value
    ```

    For example:

```
attribute_class         attribute       int_value
char_value                                comments
---------------------   ---------------  ------------------------
--------------------------------------    -----------
hash clustered tables   hash key factors                      NULL
id:10.0, id2:1.0, max_hash_key=1000.0           NULL
```

- sp_spaceused – the unused columns of sp_spaceused report the empty data pages in the hash region of virtually-hashed tables. unused reports the sum of space occupied by pages that are reserved but not allocated, and the amount of space occupied by pages that are preallocated in the hash region but not used. The data column reports the amount of space for data pages in use.

  If you issue sp_spaceused without parameters, it reports the space usage for the database. If there are virtually-hashed tables in the database, it computes the total space empty pages occupy in the hash regions for all tables. The unused column reports this calculated value, but the data column reports the total space minus the amount of space occupied by empty pages.

  For example, if you create this table and run sp_spaceused to investigate its space usage:

```
create table order_line(
id      int,
id2     int,
name    char(100),
primary key using clustered (id, id2) = (10, 1) with max 10000 key
) on myseg
```

sp_spaceused reports this:

```
name          rowtotal     reserved     data     index_size    unused
------------  --------     -----------  ------  ------------  -------
   order_line          0           1146KB     2KB            2KB   1142KB
```

Because the data pages in the hash region are empty, sp_spaceused includes the space they occupy (1112 KB) when computing the value for `unused`.

Batch input/output (IO)

Note This feature is available only on Linux pSeries.

Adaptive Server version 12.5.4 ESD #4 includes APIs that allow you to batch data page writes. This results in reduced context switching between user and kernel space, and an increase in performance when you run I/O-intensive workloads.

To enable batch IO, start Adaptive Server with the 1649 traceflag, which writes this message to the error log:

```
Enabling batch I/O using Linux Native Kernel
asynchronous disk I/O strategy
```

To disable this feature, start Adaptive Server with traceflag 1654.

Huge pages

Note This feature is available only on Linux pSeries.

The CPU-Cache translation lookaside buffer (TLB) stores information about conversions from an virtual page address to the physical page address, and every byte access to physical memory requires a conversion (called a "cache miss"). Although these cache misses are very expensive, you can improve the TLB hits is by enabling "huge pages."

Once enabled, huge pages use fewer pages to cover the physical address space, so the size of "book keeping" (mapping from the virtual to the physical address) decreases, requiring fewer entries in the TLB and improving the system performance.

To enable huge pages, start Adaptive Server with traceflag 1653. Adaptive Server adjusts its shared memory up to the nearest multiple of 256MB. For example, if you configure Adaptive Server with 800MB of shared memory, it is rounded off to 1GB (some versions of Linux do not allow you to allocate huge pages if the size is not a multiple of Hugepagesize).

Before you enable huge pages on Adaptive Server, check */proc/meminfo* to make sure Linux already has huge pages configured:

```
cat /proc/meminfo
....
HugePages_Total:     32
HugePages_Free:      32
Hugepagesize:    16384 kB
```

When Adaptive Server starts with traceflag 1653, it writes this message to the error log:

```
Creating shared memory with SHM_HUGETLB flag enabled
```

Note Memory you allocate for huge pages is used for the shared memory only. If you allocate too many huge pages, this may lead to Adaptive Server excessively swapping physical pages. You should only allocate the required number of huge pages.

Effect of ascending inserts

When it inserts rows in ascending order, Adaptive Server avoids index searches by remembering the last insertion point. This provides a hint for subsequent inserts because they can verify the insertion point to be the one next to the last inserted row.