New Features
Adaptive Server® Enterprise 12.5.3a

Document ID: DC00340-01-1253-02

Last revised: October 2005

This document describes new features available for Adaptive Server® Enterprise 12.5.3a.

1. Column encryption

Adaptive Server®, authentication and access control mechanisms ensure that only properly identified and authorized users can access data. Data encryption further protects sensitive data on disk or in archives against theft or security breaches.

Data encryption in Adaptive Server allows you to encrypt data at the column level. You encrypt only the sensitive data, thereby minimizing processing overhead.

The encrypted columns feature in Adaptive Server is easier to use than encryption in the middle tier or in the client application. You use sql statements to create the encryption keys and specify columns for encryption. Adaptive Server handles key generation and storage. Encryption and decryption of data occur automatically and transparently as you write and read the data in encrypted columns. No application changes are required, and there is no need to purchase third-party software.

When data is encrypted, it is stored as ciphertext. Non-encrypted data is stored as plaintext.

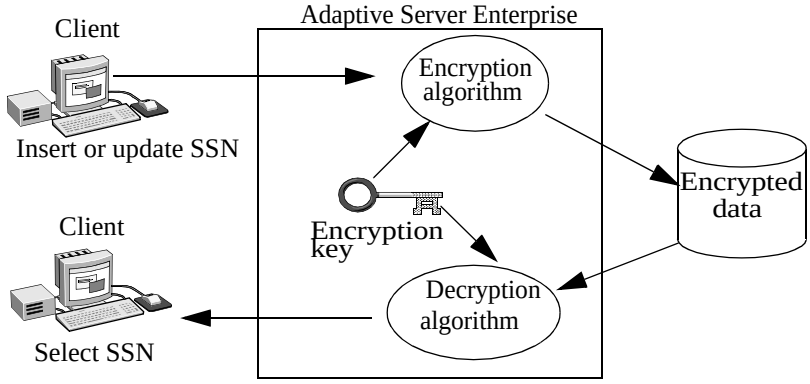
The encryption functionality is also contained in:

- Sybase Central and the Adaptive Server plug-in. See the *Sybase Central User's Manual* for more information.
- sybmigrate (the migration tool), bulk copy, and CIS, which are documented in the *Adaptive Server Enterprise System Administration Guide*.
- Replication Server. Use the *Replication Server Administration Guide* for information on encryption when replicating.

1.1 Overview

Figure 1 displays a high-level look at encryption and decryption processing in Adaptive Server. In this example, the Social Security Number (SSN) is being updated and encrypted:

Figure 1: Encryption and decryption in Adaptive Server



To create encryption keys, use create encryption key, which:

- Internally creates a symmetric key for the specified length of the key using the Security Builder Crypto™ API.
- Stores the key in encrypted form in the system catalog *sysencryptkeys*.

Adaptive Server keeps track of which key is used to encrypt a given column. Column encryption uses a symmetric encryption algorithm, which means that the same key is used for encryption and decryption.

When you insert or update data in an encrypted column, Adaptive Server transparently encrypts the data immediately before writing the row. When you select from an encrypted column, Adaptive Server decrypts the data after reading it from the row. Integer and floating point data are encrypted in a canonical form:

- Most significant bit (MSB) format for integer data.
- Institute of Electrical and Electronics Engineers (IEEE) floating point standard with MSB format for floating point data.

Data encrypted on one platform may be decrypted on another platform, provided that both platforms use the same character set.

To use encrypted columns in Adaptive Server:

1 Install the license option ASE_ENCRYPTION. See the *Adaptive Server Enterprise Installation Guide* for information.

2 Enable encryption in Adaptive Server Enterprise:

```
sp_configure 'enable encrypted columns', 0|1
```

0 – disable encryption.

1 – enable encryption.

Restart the server after you set this option.

If you turn off this option in a server that contains encrypted columns, any commands against these columns fail with an error message. Both the configuration parameter and the license option are needed to use encrypted columns. Only the System Security Officer can enable encrypted columns.

3 Set the system encryption password for a database using sp_encryption. See "Setting the system encryption password" on page 4.

4 Create the key for encrypting columns. See "Creating encryption keys" on page 5.

5 Specify the columns for encryption. See "Specifying encryption on new tables" on page 10 and "Encrypting data in existing tables" on page 11

6 Grant decrypt permission to users who must see the data. See "Permissions for decryption" on page 11.

1.2 Setting the system encryption password

The System Security Officer uses sp_encryption to set the system encryption password. The system password is specific to the database where sp_encryption is executed, and its encrypted value is stored in the sysattributes system table in that database.

```
sp_encryption system_encr_passwd, password
```

Password can be as many as 64 bytes in length, and is used by Adaptive Server to encrypt all keys in that database. Once the system encryption password has been set, you need not specify this password to access keys or data.

The system encryption password must be set in every database where encryption keys are created.

The System Security Officer can change the system password by using sp_encryption and supplying the old password.

```
sp_encryption system_encr_passwd, password [ , old_password]
```

When the system password is changed, Adaptive Server automatically reencrypts all keys in the database with the new password.

1.3 Creating and managing encryption keys

Adaptive Server generates the encryption keys and stores them in the database in encrypted form. Key owners grant table owners permission to encrypt columns in the current database with a named key.

1.3.1 Creating encryption keys

All the information related to keys and encryption is encapsulated by the create encryption key, which allows you to specify the encryption algorithm and key size, the key's default property, as well as the use of an initialization vector or padding during the encryption process.

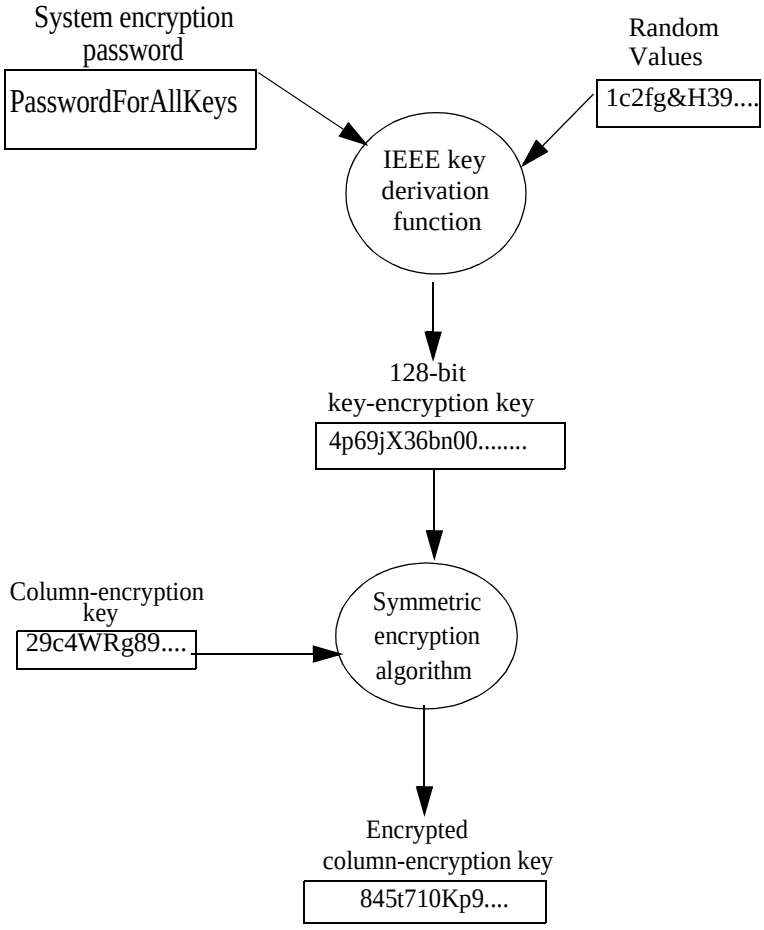
Column encryption in Adaptive Server uses the Advanced Encryption Standard (AES) symmetric key encryption algorithm, with available key sizes of 128, 192, and 256 bits. Random key generation and cryptographic functionality is provided by Security Builder Crypto™ API.

You can create separate keys for each encrypted column. Keys can be shared between columns, but each column can have only one key. To encrypt one column using an initialization vector and one column without using an initialization vector, create two separate keys, one that specifies use of an initialization vector and one that specifies no initialization vector.

The System Security Officer uses the as default clause in the create encryption key to set a default encryption key for the database. The default key is used whenever the encrypt qualifier is used without a key name on create table or alter table.

To securely protect key values, Adaptive Server uses the system encryption password to generate a 128-bit key-encrypting key, which in turn is used to encrypt the newly created key. The column-encryption key is stored in encrypted form in the *sysencryptkeys* system table.

Figure 2: Encrypting user keys



The syntax for create encryption key is:

```
create encryption key keyname [as default] for algorithm
[with [keylength num_bits]
[init_vector [null | random]]
[pad [null | random]]]
```

where:

- *keyname* – must be unique in the user's table, view, and procedure name space in the current database.

- as default – allows the System Security Officer to create a database default key for encryption. This enables the table creator to specify encryption without using a keyname on create table, alter table and select into. Adaptive Server uses the default key from the same database. The default key may be changed. See “alter encryption key” on page 35.
- *algorithm* – Advanced Encryption Standard (AES) is the only algorithm supported. AES supports key sizes of 128 bits, 192 bits, and 256 bits and a block size of 16 bytes.
- keylength *num_bits* – the size, in bits, of the key to be created. For AES, valid key lengths are 128, 192, and 256 bits. The default keylength is 128 bits.
- init_vector random – specifies use of an initialization vector during encryption. When an initialization vector is used by the encryption algorithm, the ciphertext of two identical pieces of plaintext are different, which prevents a cryptanalyst from detecting patterns of data. Use of an initialization vector can add to the security of your data.

  An initialization vector has some performance implications. Index creation, and optimized joins and searches, can be performed only on a column whose encryption key does not specify an initialization vector. See “Performance considerations” on page 19.
- init_vector null – omits the use of an initialization vector when encrypting. This makes the column suitable for supporting an index.

  The default is to use an initialization vector, that is, init_vector random. Use of an initialization vector implies using a cipher block chaining (CBC) mode of encryption; setting init_vector null implies the electronic code book (ECB) mode.
- pad null – is the default. It omits random padding of data.

  You cannot use padding if the column must support an index.
- pad random – data is automatically padded with random bytes before encryption. You can use padding instead of an initialization vector to randomize the ciphertext. Padding is suitable only for columns whose plaintext length is less than half the block length. For the AES algorithm the block length is 16 bytes.

For example, to specify a 256-bit key called “safe_key” as the database default key, the System Security Officer enters:

```
create encryption key safe_key as default for AES with
        keylength 256
```

The following example creates a 128-bit key called “salary_key” for encrypting columns using random padding:

```
create encryption key salary_key for AES with
        init_vector null pad random
```

This example creates a 192-bit key named “mykey” for encrypting columns using an initialization vector:

```
create encryption key mykey for AES with keylength 192
        init_vector random
```

The System Security Officer has default permission to create encryption keys and may grant that permission to other users.

For example:

```
grant create encryption key to key_admin_role
```

1.3.2 Using encryption keys

When you specify a column for encryption, you may use a named key from the same database, or from a different database. If you do not specify a named key, the column is automatically encrypted with the default key from the same database.

Encrypting with a key from a different database provides a distinct security advantage because it protects against access to both keys and encrypted data in the event of theft of a database dump. To access data, access to both the database archive containing data and the database archive containing encryption keys is necessary. Administrators can also protect database dumps with different passwords, making unauthorized access even more difficult.

Encrypting with a key from a different database needs special care to avoid data and key integrity problems in distributed systems. Carefully coordinate database dumps and loads. If you use a named key from a different database, Sybase recommends that:

- When you dump the database containing encrypted columns, you also dump the database where the key was created. This is necessary if new keys have been added since the last dump.
- When you dump the database containing an encryption key, dump all databases containing columns encrypted with that key. This keeps the encrypted data in sync with the available keys.

The System Security Officer can identify all the columns encrypted with a given key using sp_encryption. See “sp_encryption” on page 37.

1.3.3 Granting permissions on keys

The key owner must grant select permission on the key before another user can specify the key in a create table, alter table, and select into statements. For the database default key, the owner is the System Security Officer. Grant select permission on keys only on an "as needed" basis.

The following example allows users with db_admin_role to use the encryption key "safe_key" when specifying encryption on create table and alter table statements:

```
grant select on safe_key to db_admin_role
```

Users who process encrypted columns through insert, update, delete, and select do not need select permission on the encryption key.

1.3.4 Changing the key

As part of your information security policy, periodically change the keys used to encrypt columns. Create a new key using create encryption key, then use alter table...modify to encrypt the column with the new key

In the following example, assume that the creditcard column is already encrypted. The alter table command decrypts and reencrypts the creditcard value for every row of customer using cc_key_new.

```
create encryption key cc_key_new for AES

alter table customer modify creditcard encrypt with
cc_key_new
```

See "alter table" on page 38.

1.4 Encrypting data

You can encrypt these datatypes:

- int, smallint, tinyint
- float4 and float8
- decimal and numeric
- char and varchar
- binary and varbinary

The underlying type of encrypted data on disk is varbinary. See “Length of encrypted columns” on page 14 for more information about the length of the varbinary data.

Null values are not encrypted.

1.4.1 Specifying encryption on new tables

To encrypt columns in a new table, using this column option on the create table statement:

[encrypt [with [*database*.[*owner*].]*keyname*]]

keyname – identifies a key created using create encryption key. The creator of the table must have select permission on *keyname*. If *keyname* is not supplied, Adaptive Server looks for a default key created using the as default clause on the create encryption key. See “create table” on page 41 for the complete syntax for create table.

The following example creates two keys: a database default key, which uses default values for init_vector, pad and keylength, and a named key, cc_key, with non-default values. The ssn column in the employee table is encrypted using the default key, and the creditcard column in the customer table is encrypted with cc_key:

```
create encryption key new_key as default for AES
create encryption key cc_key for AES with
           keylength 256
           init_vector null
           pad random

create table employee_table (ssn char(15) encrypt)

create table customer (creditcard char(20)
           encrypt with cc_key)
```

1.4.2 Creating indexes on encrypted columns

You can create an index on an encrypted column if the encryption key has been specified with no initialization vector or random padding. An error occurs if you execute create index on an encrypted column that has an initialization vector or random padding. Indexes on encrypted columns are useful for equality and non-equality matches but not for range searches or ordering.

In the following example, cc_key specifies encryption without using an initialization vector or padding. This allows an index to be built on any column encrypted with cc_key:

```
create encryption key cc_key for AES
          with init_vector null

create table customer(custid int,
          creditcard varchar(16) encrypt with cc_key)

create index cust_idx on customer(creditcard)
```

1.4.3 Encrypting data in existing tables

To encrypt columns in existing tables use this column option on the alter table statement:

[encrypt [with [*database*.[*owner*].]*keyname*

keyname – identifies a key created using create encryption key. The creator of the table must have select permission on *keyname*. If *keyname* is not supplied, Adaptive Server looks for a default key created using the as default clause on the create encryption key. See the *Adaptive Server Enterprise Reference Manual* for the complete alter table syntax.

Note Encrypting a column in an existing table on which a trigger has been created causes the alter table to fail with an error. You must drop the trigger before you alter the table for encryption. Re-create the trigger after the alter table .. encrypt.

You can alter the encryption property on a column at the same time you alter other attributes such as datatype and nullability. You can also add an encrypted column using alter table.

For example:

```
alter table customer modify custid encrypt with cc_key
alter table customer add address varchar(50) encrypt
     with cc_key
```

See “alter table” on page 40 for the complete syntax.

1.5 Decrypting data

1.5.1 Permissions for decryption

You must have these two permissions to select cleartext data from an encrypted column or to search or join on an encrypted column:

- select permission on the column
- decrypt permission on the column used in the target list and in where, having, order by, update, and other such clauses

The table owner uses grant decrypt to grant explicit permission to decrypt one or more columns in a table to other users, groups and roles. Decrypt permission may be implicitly granted when a procedure or view owner grants:

- exec permission on a stored procedure that selects from an encrypted column where the owner of the procedure also owns the table containing the encrypted column.
- decrypt permission on a view column that selects from an encrypted column where the owner of the view also owns the table.

In both cases, decrypt permission need not be granted on the encrypted column in the base table.

The syntax is:

grant decrypt on [*owner*.] *table*[(*column[{* ,*column*}])] to *user* | *group* | *role*

Granting decrypt permission at the table level grants decrypt permission on all encrypted columns in the table.

Grant decrypt permission on all encrypted columns in the customer table:

```
grant decrypt on customer to accounts_role
```

The following example shows the implicit decrypt permission of user2 on the ssn column of the base table “employee”. user1 sets up the employee table and the employee_view as follows:

```
create table employee (ssn varchar(12)encrypt,
        dept_id int, start_date date, salary money)

create view emp_salary as select
            ssn, salary from employee

grant select, decrypt on emp_salary to user2
```

user2 has access to decrypted Social Security Numbers when selecting from the emp_salary view:

```
select * from emp_salary
```

1.5.2 Revoking decryption permission

You can revoke a user’s decryption permission using:

revoke decrypt on [*owner*.] *table*[(*column*[{,*column*}])] from *user* | *group* | *role*

For example:

```
revoke decrypt on customer from public
```

1.6 Dropping encryption and keys

1.6.1 Dropping encryption

If you are a table owner, you can drop the encryption on a column by using alter table with the decrypt option.

The syntax is:

drop encrypton key [*database*.[*owner*].]*keyname*

For example, drop encryption on the creditcard column in the customer table:

```
alter table customer modify creditcard decrypt
```

1.6.2 Dropping keys

The System Security Officer and key owners can drop keys. A key can be dropped only if there are no encrypted columns in any database that use the key. You cannot check suspect and offline databases for columns encrypted by the key. The command issues a warning message naming the unavailable database but the command does not fail. When the database is brought online, any tables whose columns were encrypted with the dropped key will not be usable. To restore the key, the System Administrator must load a dump of the dropped key's database from a time that precedes when the key was dropped.

Drop an encryption key using:

drop encryption key [*database*.[*owner*].]*keyname*

For example:

```
drop encryption key cust.dbo.cc_key
```

1.7 *select into* command

By default, select into creates a target table without encryption even if the source table has one or more encrypted columns. The select into requires column-level permissions, including decrypt, on the source table.

Encrypt columns on the new table by using:

```
select [all|distinct] < column_list>
        into table_name
        [(colname encrypt [with [[database.[owner].]keyname]
                [, colname encrypt
                [with [[ database.]owner].]keyname]])]
          from table_name | view_name
```

You can encrypt a specific column in the target table even if the data was not encrypted in the source table. If the column in the source table is encrypted with the same key specified for the target column, Adaptive Server bypasses the decryption step on the source table and the encryption step on the target table.

The rules for encryption on a target table are the same as those for the encrypt specifier in the create table on the source table in regard to:

- Allowable datatypes on the columns to be encrypted
- The use of the database default key when the keyname is omitted
- The requirement for select permission on the key used to encrypt the target columns.

For example, encrypt the creditcard column:

```
select creditcard, custid, sum(amount) into
     #bigspenders
     (creditcard encrypt with cust.dbo.new_cc_key)
     from daily_xacts group by creditcard
     having sum(amount) > $5000
```

1.8 Length of encrypted columns

During create table and alter table operations, Adaptive Server calculates the maximum internal length of the encrypted column. To make decisions on schema arrangements and page sizes, the Database Owner must know the maximum length of the encrypted columns.

AES is a block cipher algorithm. The length of encrypted data for block cipher algorithms is a multiple of the block size of the encryption algorithm. For AES, the block size is 128 bits, or 16 bytes. Therefore, encrypted columns occupy a minimum of 16 bytes with additional space for:

- The initialization vector. If used, the initialization vector adds 16 bytes to each encrypted column. By default, the encryption process uses an initialization vector. Specify init_vector null on create encryption key to omit the initialization vector.

- The length of the plaintext data. If the column type is char, varchar, binary, or varbinary, the data is prefixed with 2 bytes before encryption. No extra space is used by the encrypted column unless the additional 2 bytes result in the ciphertext occupying an extra block.
- A sentinel byte, which is a byte appended to the ciphertext to safeguard against the database system trimming trailing zeros.

Table 1: Ciphertext lengths

User-specified column type	Input data length	Init vector?	Internal column type	Encrypted data length
tinyint, smallint, or int	1, 2, or 4	No	varbinary(17)	17
tinyint, smallint, or int	1, 2, or 4	Yes	varbinary(33)	33
tinyint, smallint, or int	0 (null)	No	varbinary(17)	0
float, float(4), real	4	No	varbinary(17)	17
float, float(4), real	4	Yes	varbinary(33)	33
float, float(4), real	0 (null)	No	varbinary(17)	0
float(8), double	8	No	varbinary(17)	17
float(8), double	8	Yes	varbinary(33)	33
float(8), double	0 (null)	No	varbinary(17)	0
numeric(10,2)	3	No	varbinary(17)	17
numeric (10,2)	3	Yes	varbinary(33)	33
numeric (38,2)	18	No	varbinary(33)	33
numeric (38,2)	18	Yes	varbinary(49)	49
numeric (38,2)	0 (null)	No	varbinary(33)	0
char, varchar (100)	1	No	varbinary(113)	17
char, varchar(100)	14	No	varbinary(113)	17
char, varchar(100)	15	No	varbinary(113)	33
char, varchar(100)	15	Yes	varbinary(117)	49
char, varchar(100)	31	Yes	varbinary(117)	65
char, varchar(100)	0 (null)	Yes	varbinary(117)	0
binary, varbinary(100)	1	No	varbinary(113)	17
binary, varbinary(100)	14	No	varbinary(113)	17
binary, varbinary(100)	15	No	varbinary(113)	33
binary, varbinary(100)	15	Yes	varbinary(117)	49
binary, varbinary(100)	31	Yes	varbinary(117)	65
binary, varbinary(100)	0 (null)	Yes	varbinary(117)	0

char and binary are treated as variable-length datatypes and are stripped of blanks and zero padding before encryption. Any blank or zero padding is applied when the data is decrypted.

Note The column length on disk increases for encrypted columns, but the increases are invisible. For example, sp_help shows only the original size.

1.9 Auditing encrypted columns

You can audit DDL commands that relate to encrypted columns, such as creating or dropping an encryption key. Also, when you create a table the audit record contains the name of the encrypted column and the corresponding encryption key. A database-wide audit option enables you to group and manage the audit records of encrypted columns and keys.

1.9.1 Auditing options

The following table is excerpted from Adaptive Server System Administration Guide and shows the new commands audited with existing event options and the new event options.

Table 2: Auditing options, requirements, and examples

Options	*login_name*	*object_name*	Database to be in to set the option	Command or access being audited
encryption_key (database-specific)	all	Database to be audited	Any	alter encryption key create encryption key drop encryption key create table drop table alter table sp_encryption
	Example `sp_audit "encryption_key", "all", "pubs2", "on"` (Audits all the above commands in the pubs2 database.)			

1.9.2 Audit values

Table 3 lists the values that appear in the event column, arranged by sp_audit option. The “Information in extrainfo” column describes information that might appear in the extrainfo column of an audit table, based on the categories described in Table 3.

Table 3: Values in event and extrainfo columns

Audit option	Command to be audited	event	Information in extrainfo output

alter	alter table	3	*Keywords or options:*
			`ADD/DROP/MODIFY COLUMNS`
			`REPLACE COLUMN`
			`ADD CONSTRAINT`
			`DROP CONSTRAINT`
			If 1 or more encrypted columns are added, *keywords* will contain:
			`ADD/DROP/MODIFY COLUMNS column1/keyname1, [,column2/keyname2]`
			where *keyname* is the fully qualified name of the key.

create	create table	10	For encrypted columns, keywords contain column names and keynames. `EK column1/keyname1[,column2 keyname2]` where EK is a prefix indicating that subsequent information refers to encryption keys and *keyname* is the fully qualified name of the key.
encryption_key	sp_encryption	106	*Keywords* contain `ENCR_ADMIN system_encr_passwd password` ******** if password is set the first time, and contains `ENCR_ADMIN system_encr_passwd password` ******** ******** if the password is subsequently changed
	create encryption key	107	*Keywords contain:* algorithm Name-bitlength/IV [RANDOM|NULL]/PAD [RANDOM|NULL] For example: `AES-128/IV RANDOM/PAD NULL`

	alter encryption key	108	*Keywords contain*:
			`NOT DEFAULT`
			if key no longer the default key.
			`DEFAULT`
			if the key is made the default key

	drop encryption key	109	

1.9.3 New event names and numbers

You can query the audit trail for specific audit events. Use audit_event_name with the *event id* as a parameter.

audit_event_name(*event_id*)

In Table 4 the new event numbers and names are listed.

Table 4: New event numbers

Event number	Event names output
106	`Encrypted Column Administration`
107	`Create Encryption Key`
108	`Alter Encryption Key`
109	`Drop Encryption Key`

1.10 Performance considerations

Encryption is a CPU-intensive operation that may introduce a performance overhead to your application in terms of CPU usage and the elapsed time of commands that use encrypted columns. The overhead depends on the number of CPUs and Adaptive Server engines, the load on the system, the number of concurrent sessions accessing the encrypted data, and the number of encrypted columns referenced in the query. The encryption key size and the length of the encrypted data are also factors. In general, the larger the key size and the wider the data, the higher the CPU usage in the encryption operation.

This section discusses the performance implications of searching encrypted columns, and how Adaptive Server Enterprise optimizes processing of encrypted data to minimize the number of encryption and decryption operations.

1.10.1 Indexes on encrypted columns

You can create an index on an encrypted column, provided that the column's encryption key does not specify the use of an initialization vector or random padding. Using an initialization vector or random padding results in identical data encrypting to different patterns of ciphertext, which prevents the index from enforcing uniqueness.

Indexes on encrypted data are useful for equality and non-equality matching of data but not for data ordering, range searches, or finding minimum and maximum values. If Adaptive Server is performing an order-dependent search on an encrypted column, it cannot execute an indexed lookup on encrypted data. Instead, the encrypted column in each row must be decrypted and then searched. This process slows the processing of data.

1.10.2 Joins on encrypted columns

Adaptive Server optimizes the joining of two encrypted columns by performing ciphertext comparisons if the following conditions apply:

- The joining columns have the same datatype. char and varchar are considered the same datatypes, as are binary and varbinary.
- For int and float types, the columns have the same length. For numeric and decimal types, the columns have the same precision and scale.
- The joining columns are encrypted with the same key.
- The joining columns are not part of an expression. For example, a ciphertext join cannot be performed on the join where t.encr_col1 = s.encr_col1 +1.
- The encryption key was created with init_vector and pad set to NULL.
- The join operator is '=' or '<>'.
- The data has the default sort order.

For example, this sets a schema to join on ciphertext:

```
create encryption key new_cc_key for AES
        with init_vector NULL
create table customer
        (custid int,
        creditcard char(16) encrypt with new cc_key)
create table daily_xacts
       (cust_id int, creditcard char(16) encrypt with
        new_cc_key, amount money........)
```

You can also set up indexes on the joining columns:

```
create index cust_cc on customer(creditcard)

create index daily_cc on daily_xacts(creditcard)
```

Adaptive Server executes the following select statement to total a customer's daily charges on a given credit card without decrypting the creditcard column in either the customer or the daily_xacts table.

```
select sum(d.amount) from daily_xacts d, customer c
     where d.creditcard = c.creditcard and
     c.custid = 17936
```

1.10.3 Constant valued search arguments and encrypted columns

For equality and non-equality comparison of an encrypted column to a constant value, Adaptive Server optimizes the column scan by encrypting the constant value once, rather than decrypting the encrypted column in each row of the table. The same restrictions listed in “Joins on encrypted columns” on page 20 apply.

Adaptive Server cannot make use of an index to perform a range search on an encrypted column; it must decrypt each row before performing data comparisons. If a query contains other predicates, Adaptive Server selects the most efficient join order which often leaves searches against encrypted columns last, on the smallest data set.

If your query has more than one range search where there is no useful index, write the query so that the range search against the encrypted column is last. For example, the following query searches for Social Security Numbers of taxpayers in Rhode Island with incomes above $100, 000. The range search of the zipcode column is positioned before the range search of the encrypted adjusted gross income column:

```
select ss_num from taxpayers
       where zipcode like '02%' and
       agi_enc > 100000
```

1.10.4 Movement of encrypted data as ciphertext

As much as possible, Adaptive Server optimizes the copying of encrypted data by copying ciphertext instead of decrypting and reencrypting the data. This applies to select into, bulk copy, and replication.

1.11 System tables

1.11.1 *syscolumns*

In the syscolumns system table, these columns describe encryption properties:

Field	Type	Values	Description
encrtype	int	null	Type of data in encrypted form.

Field	Type	Values	Description
encrlen	int	null	Length of encrypted data.
encrkeyid	int	null	Object id of key.
encrkeydb	varchar (30)	null	Database name where the encryption key resides. NULL if key resides in the same database as the encrypted column.
encrdate	datetime	null	Creation date, copied from sysobjects.crdate.

1.11.2 *sysobjects*

sysobjects has an entry for each key with type EK (encryption key).

For cross-database key references, syscolumns.encrdate matches sysobjects.crdate.

encrkeyid in *sysencryptkeys* matches the id column in *sysobjects*.

1.11.3 *sysencryptkeys*

Each key created in a database, including the default key, has an entry in the database-specific system catalog sysencryptkeys.

Table 5: sysencryptkeys

Field	Type	Description
id	int	Encryption key ID
ekalgorithm	int	Encryption algorithm
type	smallint	Identifies the key type. The values are EK_SYMMETRIC and EK_DEFAULT.
status	int	Internal status information
eklen	smallint	User-specified length of key
value	varbinary(1282)	Encrypted value of a key. contains a symmetric encryption of the key. To encrypt keys, Adaptive Server uses AES with a 128-bit key from the system encryption password.
uid	int null	Not used
eksalt	varbinary(20)	Contains random salt used to validate decryption of the encryption key.
ekpairid	int null	Not used
pwdate	datetime null	Not used
expdate	int null	Not used
ekpwdwarn	int null	Not used

1.12 *ddlgen* utility changes

ddlgen supports generation of DDL statements for encrypted keys. To specify a key, use:

<dbName>.<owner>.<keyName>

The new type EK, for encryption key, is for generating the DDL to create an encryption key and to grant permissions on it. ddlgen generates encrypted column information and a grant decrypt statement, with the DDL of a table.

This example generates DDL for all encrypted keys in a database “accounts” on a machine named “HARBOR” using port 1955:

```
ddlgen -Uroy -Proy123 -SHARBOR:1955 -TEK
-Naccounts.dbo.%
```

Alternatively, you can specify the database name with the -D option:

```
    ddlgen -Uroy -Proy134 -SHARBOR:1955 -TEK -Ndbo.%
      -Daccounts
-------------------------------------------------------
-----------------------
-- DDL for EncryptedKey 'ssn_key'
```

```
------------------------------------------------------
-----------------------
      print 'ssn_key'

create encryption key accounts.dbo.ssn_key
      for AES
      with keylength 128
      init vector random
go


------------------------------------------------------
-----------------------
-- DDL for EncryptedKey 'ek1'
------------------------------------------------------
-----------------------
print 'ek1'

create encryption key accounts.dbo.ek1 as default
     for AES
     with keylength 192
     init vector NULL
go

use accounts
go

grant select on accounts.dbo.ek1 to acctmgr_role
go
```

ddlgen also has an extended option to generate the create encryption key that specifies the key's encrypted value as represented in *sysencryptkeys*. The option is -XOD and can be used if you must synchronize encryption keys across servers for data movement. For example, to make cc_key on server "PACIFIC" available on server "ATLANTIC", execute ddlgen using -XOD on "PACIFIC" as follows:

```
ddlgen -Sfred -Pget2work -SPACIFIC:8532 -TEK -Nsales.dbo.cc_key -XOD
```

ddlgen output is:

```
------------------------------------------------------------------------
-- DDL for EncryptedKey 'cc_key'
------------------------------------------------------------------------
print 'cc_key'

create encryption key sales.dbo.cc_key
      for AES
```

```
with keylength 128
passwd 0x0000E1D8235FEBEB118901
init_vector NULL
keyvalue 0xF772B99CE547D2932A12E0A83F2114848BD93F38016C068D720DDEBAC4DF8AA001
keystatus 32
go
```

Next change the key generated by create encryption key to specify the target database on “ATLANTIC”, and run the command on the target server. cc_key is now available on server “ATLANTIC” to decrypt data that is moved from “PACIFIC” to “ATLANTIC”.

See *Adaptive Server Enterprise Utility Guide* for more information about ddlgen syntax options, and see *Replication Server Administration Guide* for examples of using ddlgen with replicated databases.

1.13 Replicating encrypted data

If your site replicates schema changes, the following DDL statements are replicated:

- alter encryption key
- create table and alter table with extensions for encryption
- create encryption key
- grant and revoke create encryption key
- grant and revoke select on the key
- grant and revoke decrypt on the column
- sp_encryption system_encr_passwd
- drop encryption key

The keys are replicated in encrypted form.

If your system does not replicate DDL, you must manually synchronize encryption keys at the replicate site. ddlgen supports a special form of create encryption key for replicating the key’s value.

insert and update replicate encrypted columns in encrypted form, which safeguards replicated data while Replication Server processes it in stable queues on disk.

See the *Replication Server Administration Guide* for information on encryption when replicating.

1.14 Bulk copy (*bcp*)

bcp transfers encrypted data in and out of databases in either plaintext or ciphertext form. By default, bcp copies plaintext data. bcp processes plaintext data files as follows:

- Data is automatically encrypted by Adaptive Server before insertion when executing bcp in. Slow bcp is used. The user must have insert and select permission on all columns.
- Data is automatically decrypted by Adaptive Server when executing bcp out. select permission is required on all columns; in addition, decrypt permission is required on the encrypted columns.

This example copies the "customer" table out as plaintext data in native machine format:

```
bcp uksales.dbo.customer out uk_customers -n -Uroy
-Proy123
```

Use the -C option for bcp to copy the data as ciphertext. When copying ciphertext, you may copy data across different operating systems. If you are copying character data as ciphertext, both platforms must support the same character set.

The -C option for bcp allows administrators to run bcp when they lack decrypt permission on the data. When the -C option is used, bcp processes data as follows:

- Data is assumed to be in ciphertext format during execution of bcp in, and Adaptive Server performs no encryption. Use the -C option with bcp in only if the file being copied into Adaptive Server was created using the -C option on bcp out. The ciphertext must have been copied from a column with exactly the same column attributes and encrypted by the same key as the column into which the data is being copied. Fast bcp is used. The user must have insert and select permission.
- Data is copied out of Adaptive Server without decryption on bcp out. The ciphertext data is in hexadecimal format. The user must have select permission on all columns. For copying ciphertext, decrypt is not required on the encrypted columns.

- Encrypted char or varchar data retains the character set used by Adaptive Server at the time of encryption. If the data is copied in ciphertext format to another server, the character set used on the target server must match that of the encrypted data copied from the source. The character set associated with the data on the source server when it was encrypted is not stored with the encrypted data and is not known or converted on the target server.

  The System Administrator must verify that the character sets match on both the source and the target servers. You can also perform the bcp without the -C option and the character set issue is not encountered.

  The -J option for character set conversion can not be used with the -C option.

The following example copies the "customer" table. The cc_card column is copied out as ciphertext. Other columns are copied in character format. User "roy" is not required to have decrypt permission on customer cc_card.

```
 bcp uksales.dbo.customer out uk_customers -C -c -Uroy
-Proy123
```

Warning! You may use the -C flag with bcp out on a view only if the view qualification does not search encrypted columns.

1.15 Component Integration Services (CIS)

By default, encryption and decryption is handled by the remote Adaptive Server. CIS makes a one-time check for encrypted columns on the remote Adaptive Server. If the remote Adaptive Server supports encryption, CIS updates the local *syscolumns* catalog with the encrypted-column-related metadata.

- create proxy_table automatically updates *syscolumns* with any encrypted-column information from the remote tables.
- create existing table automatically updates *syscolumns* with any encrypted-column metadata from the remote tables. The encrypt keyword is not allowed in the *columnlist* for create existing table. CIS automatically marks columns as encrypted if it finds any encrypted columns on the remote table.
- create table at location with encrypted columns is not allowed.
- alter table is not allowed on encrypted columns for proxy tables.

- select into existing brings the plaintext from the source and inserts it into destination table. The local Adaptive Server then encrypts the plaintext before insertion into any encrypted columns.

The following columns are updated from the remote server's *syscolumns* catalog:

- encrtype – type of data on disk.
- encrlen – length of encrypted data.
- status2 – status bits that indicate that column is encrypted.

1.16 *load* and *dump* databases

dump and load work on the ciphertext of encrypted columns. This behavior ensures that the data for encrypted columns remains encrypted while on disk. dump and load pertain to the whole database. Default keys and keys created in the same database are dumped and loaded along with the data to which they pertain.

If the loading database contains encryption keys used in other databases, load does not succeed unless the new syntax with override is used.

```
load database key_db from "/tmp/key_db.dat" with override
```

If your keys are in a separate database from the columns they encrypt, Sybase recommends that:

- When you dump the database containing encrypted columns, you also dump the database where the key was created. This is necessary if new keys have been added since the last dump.
- When you dump the database containing an encryption key, dump all databases containing columns encrypted with that key. This keeps the encrypted data in sync with the available keys.
- After loading the database containing the encryption keys and the database containing the encrypted columns, bring both databases on line at the same time.

If you load the database containing the keys into a different-named database, errors will result when you access the encrypted columns in other databases. To change the database name of the keys' database, take the following steps:

- Before dumping the database containing the encrypted columns, use alter table to decrypt the data.
- Dump the databases containing keys and encrypted columns.

- After loading the databases, use alter table to re-encrypt the data with the keys in the newly-named database.

The consistency issues between encryption keys and encrypted columns are similar to those for cross-database referential integrity. See "Cross-database constraints and loading databases" in the *Adaptive Server Enterprise System Administration Guide*.

Do not attempt to load any dumps containing encrypted data into prior versions of Adaptive Server. Load the database into an Adaptive Server version 12.5.3a and remove any encryption from it. Perform the dump and then load the database into an Adaptive Server with a prior version. See “Downgrade procedure” on page 32.

See “Creating and managing encryption keys” on page 5 for more information on keys.

1.17 *unmount database*

When columns are encrypted by keys from other databases, unmount all related databases as a set. The interdependency of the databases containing the encrypted columns and the databases containing the keys is similar to the interdependency of databases that use referential integrity.

You use the override option to unmount a database containing columns encrypted by a key in another database.

With the following commands the encryption key created in key_db has been used to encrypt columns in col_db. These commands successfully unmount the named databases:

```
unmount database key_db, col_db
unmount database key_db with override
unmount database col_db with override
```

These commands will fail with an error message without the override:

```
unmount database key_db

unmount database col_db
```

1.18 *quiesce database*

You can use quiesce database when the database contains the encryption key.

You must use with override to quiesce a database whose columns are encrypted with keys used in other databases.

`quiesce database` ***`key_db, col_db`*** is allowed where *key_db* is the database with the encryption key and *col_db* is the database with a table that has a column encrypted with the key in *key_db*.

For example, the following commands will succeed where key_db contains the encryption key used to encrypt columns in col_db:

```
quiesce database key_tag hold key_db for external
         dump to "/tmp/keydb.dat"

quiesce database encr_tag hold col_db for external dump
      to "/tmp/col.dat" with overridewith override

quiesce database col_tag hold key_db, col_db for
    external dump to "/tmp/col.dat"
```

1.19 Drop database

To prevent accidental loss of keys, Adaptive Server fails drop database if it contains keys currently used to encrypt columns in other databases. Before dropping the database containing the encryption keys you must first remove the encryption or drop the database containing the encrypted columns.

In the following example key_db is the database where the encryption key resides and col_db is the database containing the encrypted columns:

```
drop database key_db, col_db
```

Adaptive Server raises an error and fails to drop key_db. The drop of col_db succeeds. To drop both databases, drop col_db first:

```
drop database col_db, key_db
```

1.20 *sybmigrate*

sybmigrate is the migration tool used to migrate data from one server to another.

By default, sybmigrate migrates encrypted columns in ciphertext format. This avoids the overhead of decrypting data at the source and encrypting at the target. In some cases, sybmigrate chooses the reencrypt method of migration, decrypting data at the source and encrypting at the target.

For databases with encrypted columns, sybmigrate:

1 Migrates the system encryption password. If you specify not to migrate the system encryption password, sybmigrate migrates the encrypted columns using the reencrypt method instead of migrating the ciphertext directly.

2 Migrates the encryption keys. You may select the keys to migrate. sybmigrate automatically selects keys in the current database used to encrypt columns in the same database. If you have selected migration of the system encryption password, sybmigrate migrates the encryption keys using their actual values. The key values from the *sysencryptkeys* system table have been encrypted using the system encryption password and these are the values that are migrated. If you have not migrated the system encryption password, sybmigrate migrates the keys by name, to avoid migrating keys that will not decrypt correctly at the target. Migrating the key by name causes the key at the target to be created with a different key value from the key at the source.

3 Migrates the data. By default, the data is transferred in its ciphertext form. Ciphertext data can be migrated to a different operating system. Character data requires that the target server uses the same character set as the source.

sybmigrate works on a database as a unit of work. If your database on the source server has data encrypted by a key in another database, migrate the key's database first.

sybmigrate chooses to reencrypt migrated data when:

- Any keys in the current database are specifically not selected for migration, or already exist in the target server. There is no guarantee that the keys at the target are identical to the keys are the source, so the migrating data must be reencrypted.
- The system password was not selected for migration. When the system password at the target differs from that at the source, the keys cannot be migrated by value. In turn, the data cannot be migrated as ciphertext.
- The user uses the following flag:

```
sybmigrate -T 'ALWAYS_REENCRYPT'
```

Reencrypting data can slow performance. A message to this effect is written to the migration log file when you perform migration with reencryption mode.

To migrate encrypted columns, you must have both sa_role and sso_role enabled.

1.21 Downgrade procedure

If you have never configured enable encrypted columns in your server, you need not take any action before using an older version of Adaptive Server with 12.5.3a databases. One way to verify that you have never configured encrypted columns is to check that the system table *sysencryptkeys* does not exist in any database.

All databases should be backed up prior to the downgrade procedure.

Before downgrading a server that has been configured for encrypted columns, you must either drop or modify any tables with encrypted columns to remove encryption. You then run sp_encryption remove_catalog, which verifies that there are no encrypted columns in each database and then removes the system table *sysencryptkeys*. The new columns in *syscolumns* added for 12.5.3a are ignored by an older binary and need not be removed

To downgrade from a 12.5.3a server to an earlier version of 12.5.x:

1 If encrypted columns are not currently enabled, the System Security Officer executes:

```
sp_configure 'enable encrypted columns',1
```

2 Use drop or alter to decrypt all tables with encrypted columns in all databases. The System Security Officer runs the following command in each database where encryption keys were created to list all encryption keys created in that database:

```
sp_encryption help
```

For each key listed, the System Security Officer runs the following to see a list of columns encrypted with a particular key:

sp_encryption help, <*keyname*>, 'display_cols'

For each encrypted column, one of the following steps must be performed:

- alter table to decrypt the encrypted columns
- alter table to drop the encrypted columns
- drop the table containing the encrypted column
- drop the encryption key

3 To guarantee that no other user can access Adaptive Server while a system table is removed, restart the server in single-user mode. See the *Adaptive Server Enterprise Utility Guide*.

4 A user with sso_role and sa_role must execute the following system stored procedure, which removes the *sysencryptkeys* catalog from each database:

```
sp_encryption remove_catalog
```

If a database is unavailable, the command prints an error and exits. If columns encrypted by any key in *sysencryptkeys* exist, the command does not drop *sysencryptkeys*, but prints an error or warning and continues with the next database.

If sp_encryption is successful in removing *sysencryptkeys*, it also removes these rows from *sysattributes* in each database:

- The record of the upgrade item that added *sysencryptkeys*
- The system encryption password for the database

5 Drop the system stored procedure sp_encryption from the *sybsystemprocs* database.

6 Shut down the server. You can now use a 12.5.x Adaptive Server binary from a pre-12.5.3a version.

To reenable encrypted columns, when rolling forward from a downgraded 12.5.3a server back to 12.5.3a, configure enable encrypted columns. Upon restarting the 12.5.3a server, the *sysencryptkeys* system table is installed in each database.

1.21.1 Replication issues with downgrade

When downgrading a server that has replication enabled on databases that contain encrypted data, you must do one of the following before you start the downgrade procedure:

1 Ensure that all replicated data in the primary database transaction log has been successfully transferred to the standby or replicate database. The process for doing this is application dependent.

2 Truncate the transaction log in the primary database, and zero the RS locator for that database in the Replication Server. Use the following commands:

In the primary database run:

```
sp_stop_rep_agent primary_dbname
       dbcc settrunc ('ltm', 'ignore')
       dump tran  primary_dbname with truncate_only
       dbcc settruc ('ltm', 'valid')
```

Shutdown Replication Server. In the RSSD for the Replication Server run:

```
rs_zeroltm primary_servername, primary_dbname
```

1.22 New commands

1.22.1 *create encryption key*

All the information related to keys and encryption is encapsulated by create encryption key, which allows you to specify the encryption algorithm and key size, the key's default property, as well as the use of an initialization vector or padding during the encryption process

Adaptive Server uses Security Builder Crypto™ for key generation and encryption.

The System Security Officer has default permission to create encryption keys and may grant that permission to other users.

Syntax

create encryption key [[*database.[owner].]keyname* [as default] for *algorithm*
[with [keylength *num_bits*]
[init_vector [NULL | random]]
[pad [NULL | random]]]

- *keyname* – must be unique in the user's table, view, and procedure name space in the current database.
- as default – allows the System Security Officer to create a database default key for encryption. This enables the table creator to specify encryption without using a keyname on create table, alter table and select into. Adaptive Server uses the default key from the same database. The default key may be changed. See "alter encryption key" on page 35.
- *algorithm* – Advanced Encryption Standard (AES) is the only algorithm supported. AES supports key sizes of 128 bits, 192 bits, and 256 bits and a block size of 16 bytes.
- keylength *num_bits* – the size, in bits, of the key to be created. For AES, valid key lengths are 128, 192, and 256 bits. The default keylength is 128 bits.
- init_vector random – specifies use of an initialization vector during encryption. When an initialization vector is used by the encryption algorithm, the ciphertext of two identical pieces of plaintext are different, which prevents a cryptanalyst from detecting patterns of data. Use of an initialization vector can add to the security of your data.

An initialization vector has some performance implications. Index creation, and optimized joins and searches, can be performed only on a column whose encryption key does not specify an initialization vector. See "Performance considerations" on page 19.

- init_vector null – omits the use of an initialization vector when encrypting. This makes the column suitable for supporting an index.

  The default is to use an initialization vector, that is, init_vector random. Use of an initialization vector implies using a cipher block chaining (CBC) mode of encryption; setting init_vector null implies the electronic code book (ECB) mode.

- pad null – is the default. It omits random padding of data.

  You cannot use padding if the column must support an index.

- pad random – data is automatically padded with random bytes before encryption. You can use padding instead of an initialization vector to randomize the ciphertext. Padding is suitable only for columns whose plaintext length is less than half the block length. For the AES algorithm the block length is 16 bytes.

For example, to specify a 256-bit key called "safe_key" as the database default key, the System Security Officer enters:

```
create encryption key safe_key as default for AES with
       keylength 256
```

The following example creates a 128-bit key called "salary_key" for encrypting columns using random padding:

```
create encryption key salary_key for AES with
        init_vector null pad random
```

This example creates a 192-bit key named "mykey" for encrypting columns using an initialization vector:

```
create encryption key mykey for AES with keylength 192
       init_vector random
```

1.22.2 *alter encryption key*

To change the default encryption key, enter:

alter encryption key *key1* as default

If a default key already exists it no longer has the default property. *key1* becomes the default key.

If *key1* is the default key, you can remove the default designation for *key1* as follows:

```
alter encryption key key1 as not default
```

If *key1* is not the default key, the command returns an error.

alter encryption key as default or not default can be executed only by the System Security Officer and cannot be granted to other users

1.22.3 *drop encryption key*

The key owner and the System Security Officer can drop encryption keys. The command fails if any column in any database is encrypted using the key.

Syntax

```
drop encryption key [database.[owner].]keyname
```

1.22.4 *grant create encryption key*

The System Security Officer grants permission to create encryption keys.

Syntax

```
grant create encryption key to user | role| group
```

1.22.5 *revoke create encryption key*

The System Security Officer can revoke permission from other users, groups, and roles to create encryption keys.

Syntax

```
revoke create encryption key from user | role | group
```

1.22.6 *grant decrypt*

The table owner or the System Security Officer grants decrypt permission on a table or a list of columns in a table.

Syntax

```
grant decrypt on [ owner. ]tablename[(columnname [{,columname}])]
    to user | group | role
```

Note grant all on a table or column does not grant decrypt permission.

1.22.7 *revoke decrypt*

The table owner or the System Security Officer revokes decrypt permission on a table or a list of columns in a table.

Syntax

revoke decrypt on [owner.] *tablename*[(*columnname* [{,*columname*}])] from *user* | *group* | *role*

1.23 *sp_encryption*

The System Security Officer sets the system encryption password using sp_encryption. The system password is specific to the database where sp_encryption is executed, and its encrypted value is stored in the sysattributes system table in that database.

sp_encryption system_encr_passwd, '*password*'

The password specified using sp_encryption can be up to 64 bytes in length, and is used by Adaptive Server to encrypt all keys in that database. You need not specify this password to access keys or data.

The system encryption password must be set in every database where encryption keys are created.

The System Security Officer can change the system password by using sp_encryption and supplying the old password.

sp_encryption system_encr_passwd, *'password* '[, *'old_password*']

When the system password is changed, Adaptive Server automatically reencrypts all keys in the database with the new password.

1.23.1 *sp_encryption help*

sp_encryption help displays the key's name, owner, size and encryption algorithm. It also indicates whether the key has been designated as the database default key, and whether encryption with this key uses random padding or an initialization vector.

sp_encryption help [, *keyname* [, display_cols]]

When sp_encryption help is run by a user with sso_role, the key properties of all keys in the database are displayed. When run by a user without sso_role, the key properties are displayed for only those keys for which the user has select permission in that database.

sp_encryption help, *keyname* displays the properties of keyname. If the command is run by a user without sso_role, the user must have select permission on the key.

sp_encryption help, keyname, display_cols may be run only by a user with sso_role. It lists the columns encrypted by keyname.

1.24 Changes to command syntax

The addition of the encrypted columns feature has changed or added syntax to the commands in this section.

1.24.1 alter table

Use alter table to encrypt or decrypt existing data or to add an encrypted column to a table.

Syntax

Encrypt a column:

alter table *tablename* add *column_name*
encrypt [with [*database*.[*owner*].]*keyname*

Decrypt an existing column:

[decrypt [with [*database*.[*owner*].]*keyname*]]

keyname – identifies a key created using create encryption key. The creator of the table must have select permission on *keyname*. If *keyname* is not supplied, Adaptive Server looks for a default key created using create encryption key or alter encryption key as default.

Example

Create an encryption key and encrypt ssn column in existing “employee” table.

```
alter table employee modify ssn
         encrypt with ssn_key
grant decrypt on employee(ssn) to hr_manager_role,
         hr_director_role
```

Use alter table to change an encryption key. When the encrypt qualifier is used on a column that is already encrypted, Adaptive Server decrypts the column and re-encrypts it with the new key. This operation may take a significant amount of time if the table contains a large number of rows.

1.24.2 create table

Use the encrypt qualifier to set up encryption on a table column.

Syntax

create table *tablename* (*colname* datatype [default_clause]
[encrypt [with [*database*.[*owner*].]*keyname*]])

keyname – identifies a key created using create encryption key. The creator of the table must have select permission on *keyname*. If *keyname* is not supplied, Adaptive Server looks for a default key created using the as default clause on create encryption key or alter encryption key.

Example

Create an employee table with encryption:

```
create table employee_table (ssn char(15) null encrypt)
```

1.24.3 *enable encrypted columns* configuration parameter

The configuration parameter enable encrypted columns must be set to 1 to use the encryption functionality. If the configuration option is turned off in a server that contains encrypted columns, any commands against those columns fail with an error. Both the configuration parameter and the license option are needed to enable encryption.

```
sp_configure 'enable encrypted columns', 1
```

1.24.4 *load database*

If the loading database contains encryption keys used in other databases, load does not succeed unless with override is used.

```
load database key_db from "/tmp/key_db.dat" with override
```

1.24.5 *select into*

select into requires column level permissions, including decrypt, on the source table.

Syntax

Encrypt columns on the new table using this syntax:

```
select [all|distinct] < column_list>
        into target_table
        [(colname encrypt [with [database.[owner].]keyname]
        [,colname encrypt
        [with [database.[owner].]keyname]])]
    from tabname | viewname
```

Example

Encrypting creditcard column in the table.

```
select creditcard, custid, sum(amount) into
       #bigspenders
       (creditcard encrypt with
          cust.database.new_cc_key)
       from daily_xacts group by creditcard
       having sum(amount) > $5000
```

1.24.6 *dbcc*

dbcc checkcatalog includes the following additional consistency checks:

1 For each encryption key row in *sysobjects*, *sysencryptkeys* is checked for the existence of a row defining that key.

2 For each column in *syscolumns* marked for encryption, the existence of the key is checked in *sysobjects* and *sysencryptkeys*.

1.25 Full syntax for commands

The following information shows the full syntax for the commands covered in this bulletin.

1.25.1 *alter encryption key*

alter encryption key *key1* as default | not default

1.25.2 alter table

```
alter table [[database.[owner].table_name
    { add column_name datatype
        [default {constant_expression | user | null}]
        {identity | null | not null}
        [off row | in row]
        [ [constraint constraint_name]
        { { unique | primary key }
            [clustered | nonclustered]
            [asc | desc]
            [with { fillfactor = pct,
                max_rows_per_page = num_rows,
                reservepagegap = num_pages }]
            [on segment_name]
        | references [[database.]owner.]ref_table
            [(ref_column)]
            [match full]
        | check (search_condition) ] ... }
        [encrypt [with [database .] owner ] .]keyname
        [, next_column]...
    | add {[constraint constraint_name]
    {  unique | primary key}
            [clustered | nonclustered]
            (column_name [asc | desc]
        [, column_name [asc | desc]...])
        [with { fillfactor = pct,
                max_rows_per_page = num_rows,
                reservepagegap = num_pages}]
        [on segment_name]
    | foreign key (column_name [{, column_name}...])
        references [[database.]owner.]ref_table
        [(ref_column [{, ref_column}...])]
```

```
        [match full]
    | check (search_condition)}
    | drop {column_name [, column_name]...
        | constraint constraint_name }
    | modify column_name datatype [null | not null]
            [encrypt | [with [database .]owner].] keyname
    | modify column_name datatype [null | not null
            [encrypt | [with [database .]owner].] keyname]
                |decrypt
            [, next_column]...
    | replace column_name
        default { constant_expression | user | null}
        | partition number_of_partitions
    | unpartition| { enable | disable } trigger
    | lock {allpages | datarows | datapages } }
    | with exp_row_size=num_bytes
    | [ alter_partition_clause ]
    [ partition_clause ]
```

1.25.3 *create table*

```
create table [database .[owner ].]table_name (column_name datatype)
    [default {constant_expression  | user | null}]
    {[{identity | null | not null}]
        [off row | [ in row [ (size_in_bytes) ] ]
    [[constraint constraint_name ]
        {{unique | primary key}
        [clustered | nonclustered] [asc | desc]
        [with { fillfactor = pct,
                max_rows_per_page = num_rows, }
                reservepagegap = num_pages }]
        [on segment_name]
        | references [[database .]owner .]ref_table
            [(ref_column )]
            [match full]
            | check (search_condition)}]}
            [match full]...
    [encrypt [with [[ database.] owner] . ] keyname]
    | [constraint constraint_name]
        {{unique | primary key}
            [clustered | nonclustered]
            (column_name [asc | desc]
                [{, column_name [asc | desc]}...])
            [with { fillfactor = pct
                max_rows_per_page = num_rows ,
                reservepagegap = num_pages } ]
            [on segment_name]
        |foreign key (column_name [{,column_name}...])
            references [[database.]owner.]ref_table
                [(ref_column [{, ref_column}...])]
```

```
                    [match full]
        | check (search_condition) ... }
    [{, {next_column | next_constraint}}...])
    [lock {datarows | datapages | allpages }]
    [with { max_rows_per_page = num_rows,
            exp_row_size = num_bytes,
            reservepagegap = num_pages,
            identity_gap = value } ]
    [ table_lob_clause ]
    [ on segment_name ]
    [ [ external table ] at pathname ]
    [ partition_clause ]
```

1.25.4 *select*

```
into_clause ::=
   into [[database.]owner.]table_name
   [(colname encrypt [with [database.] owner].] keyname
            [, colname encrypt [with [database. ]owner] . ]
               keyname])]
   [ lock {datarows | datapages | allpages } ]
   [ with into_option [, into_option] ...]

      into_option ::=
          | max_rows_per_page = num_rows
          | exp_row_size = num_bytes
          | reservepagegap = num_pages
          | identity_gap = gap
          [existing table table_name]
          [[external type] at "path_name"
          [column delimiter delimiter]]
```

2. Internet protocol version 6

To make Adaptive Server IPv6-aware, you must start Adaptive Server with trace flag 7841, which allows Adaptive Server to determine IPv6 availability, and makes Adaptive Server IPv6-aware.

IPv6 is available on Sun Solaris 32- and 64-bit platforms.

- Sun Solaris 32- and 64-bit platforms
- Windows
- HP 32- and 64-bit platforms

For more information about how to setup and administrate IPv6-enabled networking on your platform, see your operating system documentation.

3. Real Time Messaging

Real Time Messaging Services for Adaptive Server version 12.5.3a provides support for both TIBCO JMS and IBM WebSphere MQSeries.

4. PAM support in 64-bit Adaptive Server on AIX

Adaptive Server version 12.5.3a on AIX 64-bit supports Pluggable Authentication Module based User Authentication (PAMUA). Even though 12.5.3a 64-bit Adaptive Server is released on AIX 5.1, IBM supports PAM on 64-bit applications on AIX 5.2 only. Sybase recommends that you upgrade to AIX 5.2 to use this feature. Please contact your OS support representative to make sure you have the latest patch for PAM available on your IBM host.

64-bit PAM libraries shipped with AIX 5.1 do not have the ability to automatically load 64bit libraries from `/usr/lib/security/64`. To enable PAMUA in 64-bit ASE 12.5.3a on AIX5.1, you need to supply the full pathname of the PAM module in */etc/pam.conf file*. The following example illustrates how to specify the OS module pam_aix on an AIX 5.1 machine.

Adaptive Server authorization required `/usr/lib/security/64/pam_aix`

5. Feature matrix

This matrix shows the various features available in Adaptive Server Enterprise version 12.5.3a for different platforms.

Figure 3: Matrix

Operating System	Sol32	Sol64	HP32	HP64	AIX64	Linux x86	Windows x86
Options							
Encrypted Columns	new for 12.5.3a	new for 12.5.3a	new for 12.5.3a	new for 12.5.3a	new for 12.5.3a	new for 12.5.3a	new for 12.5.3a
High Availability	*	*	*	*	*	*	*
Distributed Transaction	*	*	*	*	*	*	*
XML Management	*	*	*	*	*	*	*
Java Option	*	*	*	*	*	*	*
Native XML	*	*	*	*	*	*	*
Java Based XML	*	*	*	*	*	*	*
Web Services	*	*	*	*	*	*	*
Security & Dir Services	*	*	*	*	*	*	*
LDAP Server Directory	*	*	*	*	new for 12.5.3a	*	*
LDAP User Authentication	*	*	*	*	new for 12.5.3a	*	*
Secure Socket Layer	*	*	*	*	*	*	*
Cybersafe Kerberos	*	*					*
MIT Kerberos	*	*		new for 12.5.3a	new for 12.5.3a	*	
Platform Native Kerberos	*	*					
Fine Grained Access Control	*	*	*	*	*	*	*
Pluggable Authentication Modu	*	*			new for 12.5.3a	*	
Content Management	*	*	*	*	*	*	*
Enhanced Full Text Search	*	*	*	*	*	*	*
Real Time Messaging	*	*		*	*	*	*
JMS support	*	*		*	*	*	*
Websphere MQ support	new for 12.5.3a	new for 12.5.3a		new for 12.5.3a	new for 12.5.3a	new for 12.5.3a	
Disaster Recovery	*		*			*	*
Features Included with ASE							
IPv6	*	*	new for 12.5.3a	new for 12.5.3a			new for 12.5.3a
Cross Platform Dump and Loa	*	*	*	*	*	*	*
Job Scheduler	*	*	*	*	*	*	*
ASE Replicator	*	*	*	*	*	*	*

- * Indicates supported feature. A blank cell indicates that feature is not available on that platform.
- High Availability support indicates Active-Active support.

Active-Passive support for High Availability is limited to Solaris/SPARC platform only.

- Option availability varies on various editions. Please check the Adaptive Server Enterprise Data Sheet for more information about the differences between various editions.