ASE 12.5 Internals: An Overview of Permanent Data Structures

An in-depth study of how

ASE data is organized

Lubo Cipin consults with Objective Edge Inc. in Toronto and, while working at Sybase, upgraded the internals course to ASE 12.0. He can be reached at lubocipin@yahoo.ca.

This article provides a background for ASE management, troubleshooting, and performance tuning, primarily covering the permanent data structures of ASE 12.5. It includes a discussion of pages, large columns, and indexes; the mechanism behind page allocation; and database consistency checking, corruption and correction. We will conclude with a brief look at the choice of a logical page size for the server, a capability introduced in release 12.5. A diagram of the permanent data structures down to the page level can be found on page 3.

Additional information on the topics covered here, and on the structure of data, index and transaction log rows can be found in Sybase courses and documentation.

Data Structures

A data structure is a specific collection of data used by software. Permanent data structures are those which ASE maintains in memory and stores on disk and in backups. The dynamic data structures are those which exist only in memory while the server is running, e.g., the process and lock structures.

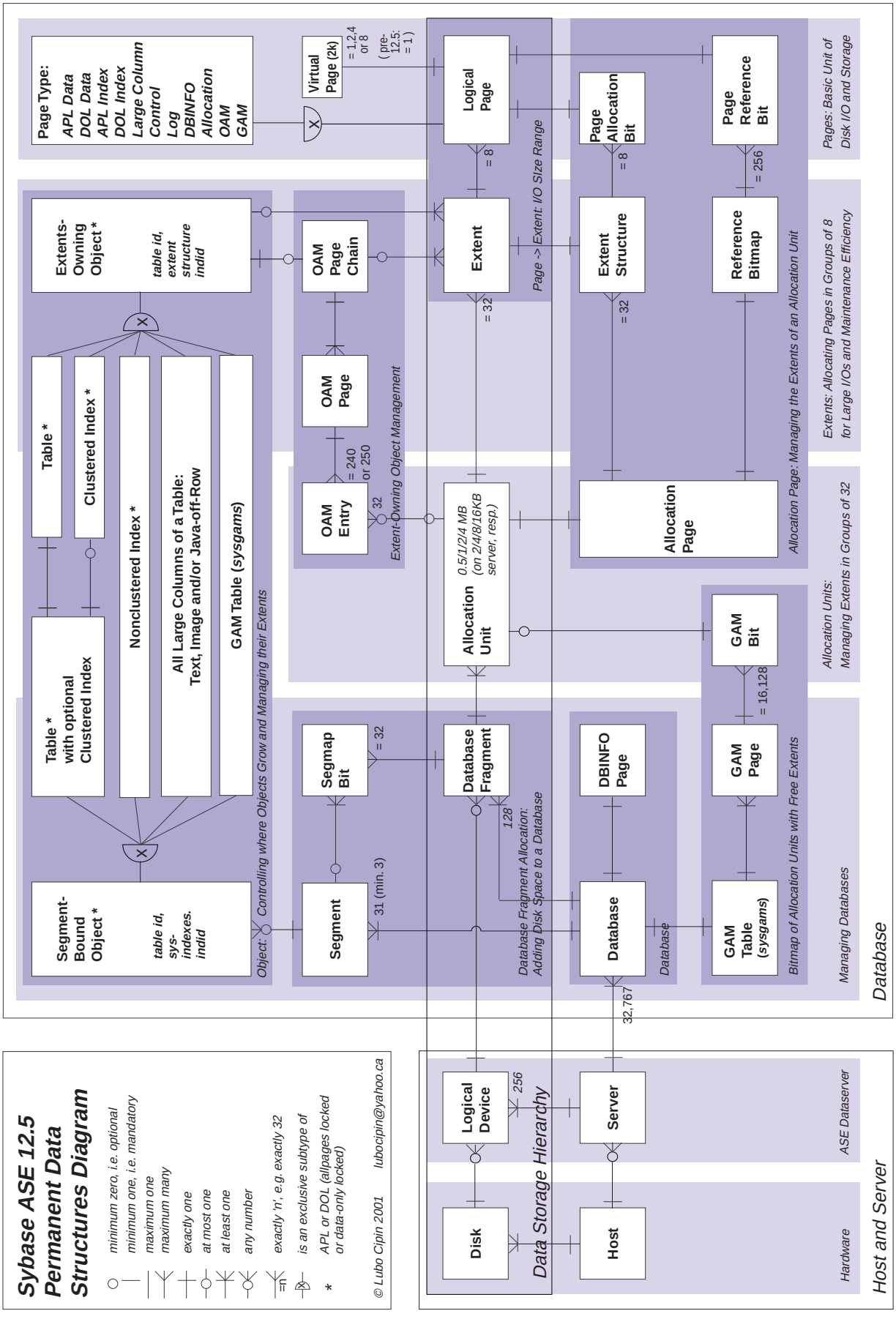
The Diagram

The boxes in the diagram on page 3 represent different data structures. The lines between the boxes represent direct relationships and their optionality and cardinality: e.g., *can* or *must* a device be related to a *minimum* of zero or one database, and a *maximum* of one or several? The bi-directional relationships answer these two questions for both directions. An equal sign indicates that the cardinality is a fixed number, e.g. an extent always consists of eight pages. A half-circle symbol represents a subtype relationship, e.g., a DOL data page is a type of logical page. An "x" in the half-circle means that the subtype relationship is exclusive.

Pages and Tables

In ASE, data is stored as rows in pages which make up user and system tables. A regular data or index row cannot span multiple pages, it must fit in a single page. When a row cannot be inserted or expanded in the page where it belongs, a page is added to the table or the row is moved to another page. Each row consists of column values whose datatypes can be of fixed or variable length. Nullable columns, regardless of datatype, are stored as variable length columns.

All pages have a header which contain page-specific system information such as its logical address, previous and next page pointers, object id, index level, and page type. All page headers have 32 bytes, except for data-only locked (DOL) data and index page headers, which have 44 bytes. The only DOL system tables in ASE are some of the new system tables introduced in each release since 11.9.2. Unless the server-wide table locking


Sybase ASE 12.5
Permanent Data
Structures Diagram
minimum zero, i.e. optional
minimum one, i.e. mandatory
maximum one
maximum many
exactly one
at most one
at least one
any number
exactly 'n', e.g. exactly 32
is an exclusive subtype of
APL or DOL (allpages locked or data-only locked)
© Lubo Cipin 2001 lubocipin@yahoo.ca
Page Type:
APL Data
DOL Data
APL Index
DOL Index
Large Column
Control
Log
DBINFO
Allocation
OAM
GAM
Virtual Page (2k)
= 1,2,4 or 8
(pre-12.5: = 1)
Logical Page
= 8
Page Allocation Bit
Page Reference Bit
= 256
Pages: Basic Unit of Disk I/O and Storage
Extents-Owning Object *
table id, extent structure indid
Table *
Clustered Index *
Nonclustered Index *
All Large Columns of a Table: Text, Image and/or Java-off-Row
GAM Table (sysgams)
OAM Page Chain
OAM Page
= 240 or 250
OAM Entry
32
Extent
= 32
Page -> Extent: I/O Size Range
Extent Structure
Reference Bitmap
Extents: Allocating Pages in Groups of 8 for Large I/Os and Maintenance Efficiency
Allocation Page
Allocation Page: Managing the Extents of an Allocation Unit
Allocation Unit
0.5/1/2/4 MB (on 2/4/8/16KB server, resp.)
Segment-Bound Object *
table id, sys-indexes. indid
Object: Controlling where Objects Grow and Managing their Extents
Extent-Owning Object Management
Table * with optional Clustered Index
Segment
31 (min. 3)
Segmap Bit
= 32
Database Fragment
128
Database Fragment Allocation: Adding Disk Space to a Database
DBINFO Page
Database
32,767
Database
GAM Bit
= 16,128
GAM Page
GAM Table (sysgams)
Bitmap of Allocation Units with Free Extents
Allocation Units: Managing Extents in Groups of 32
Managing Databases
Database
Logical Device
256
Server
Disk
Host
Data Storage Hierarchy
Hardware
ASE Dataserver
Host and Server

scheme, which by default is allpages locked (APL), is changed, tables have to be altered explicitly to become data-only locked (DOL). New tables can also be explicitly created as DOL. A table consists of data pages, an OAM (Object Allocation Map) page chain, an optional number of large column pages, and, if the table is partitioned, one control page per partition.

Large Columns

Among the variable length datatypes, there are three which are commonly called *large objects*: text, image, and Java-off-row (available since release 12.0). Java classes can be used to define the datatype for a column. If the serialized Java objects can fit within a binary column they are called *Java-in-row* values. If they are larger, which is usually the case, they can be stored in the same way as image values and are called *Java-off-row*. In this article and the diagram, the large objects are called large *columns* as we already have two other meanings for the word "object."

Large column values are stored in two parts: 1) the actual data value spanning as many doubly-linked large column pages as necessary, and 2) a 16-byte entry—including a pointer to the first page of that page chain—in the regular row of a data page. All the large columns of a table are treated as a single object: They are represented by a single row in the sysindexes table, are bound to a single segment, and share the same set of extents.

APL and DOL Indexes

A table may have one clustered index, several nonclustered indexes, and/or several large columns. Indexes can be used to speed up many queries by reducing the number of logical I/Os and to enforce column value uniqueness. Prior to release 11.9.2, all tables had a locking scheme which is now called all-pages locking or APL. This refers to the fact that all modified index pages are locked as well as the data page when a row is inserted, updated, or deleted. Lock contention on APL tables often takes place on their index pages.

In the two data-only locked (DOL) locking schemes introduced in release 11.9.x (datapages locked and datarows locked) index pages are, for all intents and purposes, never locked. This reduces the lock contention as well as lock maintenance overhead. When a table is altered to a DOL locking scheme, it is rebuilt because the page size, page header, and data and index row structures are different for DOL tables and their indexes. Another difference is that there is a row offset table in DOL index pages but not in APL index pages.

The leaf (i.e., lowest) level of an APL clustered index is the data level. In DOL tables, data pages are not split. If a row does not fit where it belongs (in case the table has a clustered index), it is moved to another page instead. With each additional insert or update against it a DOL table may get less ordered or "clustered." Because of this, the data rows can be guaranteed to be in perfect order according to the clustered index key only after the clustered index is rebuilt or right after the *reorg rebuild* command has completed. This in turn makes it necessary for the clustered index to have a leaf level, containing perfectly ordered index keys, which is separate from the data pages. Structurally a DOL clustered index is the same as a DOL nonclustered index. Prior to 11.9.2, a clustered APL table was always represented with a single row, with index id 1 in the *sysindexes* table. Now, a DOL clustered table is represented as one row with index id 0 for the table, and another row with an index id between 2 and 250 for the clustered index. Despite these separate rows, a DOL table and its clustered index cannot be separated onto different segments.

In APL tables and indexes, the page chain is maintained for every level. In DOL tables and indexes, the next and previous page pointers of data page and index page chains are no longer maintained, except for the leaf level of any index, clustered or nonclustered. The leaf level of any index needs to form an intact page chain to support covered index scans (both matching and non-matching) and range scans, and to avoid the need for sorting to process clauses such as an ORDER BY. As of release 11.5, the leaf level of indexes can also be used for bi-directional index scans, when ASE is configured for this.

ASE Servers

A single host machine may run multiple ASE servers simultaneously. In this case, each server listens to its own set of port numbers and has its own dedicated shared memory, of an individual size, which is shared by all the internal processes but not with the other ASE servers. The different servers may have access to any of the disks of the host machine, but each has its own logical devices and databases.

Databases

Each ASE server maintains a small number of system databases which store the permanent server-level system data, and a potentially large number of user databases. In terms of permanent data structures all databases, be they user or system databases, consist of user tables and their indexes, system tables and their indexes, the system table *syslogs* which stores the transaction log, a single DBINFO page and a

number of OAM, GAM (Global Allocation Map), and allocation pages. The last three types of pages are used to manage objects, a list of the allocation units with free extents for the entire database, and allocation units respectively.

Each database has a single DBINFO page which contains a few dozen fields holding a variety of internal database information, including database option settings, pointers to the replication truncation point and the most recent checkpoint record in the transaction log, the current timestamp for this database, and the value of the next object id.

Disks and Logical Devices

To reserve space for ASE, logical devices have to be initialized with the DISK INIT command. These can be raw partitions or file systems. As of release 12.0, write consistency to disk can be guaranteed for file systems (with the *DSYNC* option). However, file system performance may or may not be faster than raw device performance, depending on factors such as the type of read or write activity performed. When writes are guaranteed and ASE receives a signal indicating that the write has completed, this means that the write is physically on the disk, or at least in the guaranteed buffer of a logical volume manager, and not only in an O/S buffer. A logical device might span almost an entire disk but typically spans only a portion of one. There is one row in the *sysdevices* table for each logical device.

Sybase mirroring takes place at the logical device level. Another consideration in choosing how to allocate devices is that each logical device has its own pending I/O queue in ASE's shared memory. If the server runs with multiple engines and there is heavy disk activity against one of the logical devices, there may be a bottleneck in adding pending I/Os to this single queue and in waiting for the earlier pending I/Os to complete. In this case higher disk I/O throughput may be achieved simply by having several logical devices. Having multiple queues can help reduce any (spinlock) contention experienced in attaching new pending I/Os. The other possible cause for the bottleneck can be addressed by mapping the logical devices onto separate disks, and better yet, separate disk controllers.

Database Fragments

When a database is created or expanded it is assigned one or several fragments of previously initialized logical devices. Each individual fragment belongs to a single database. The name "database fragment" is therefore more descriptive than "device fragment," although both terms are used synonymously. A database fragment can span an entire logical device or a portion of it. A database's fragments are not freed up and available for use by others until it has been dropped or shrunk by mirroring a device it has fragments on, making the secondary device the primary, unmirroring the old primary, and dropping it.

On the other hand, a database can easily be given more disk space by allocating additional fragments of any logical devices. Consecutive fragments of a device may belong to separate databases, depending on the order of database creations and expansions on that device. For each database fragment, there is one row in the *sysusages* table. The exception to this is that sometimes, after a dump is loaded, ASE collapses a couple of these rows into a single one; the minimal conditions for this is that the rows be consecutive, that they describe fragments of a single device, belong to the same database, and have the same usage (data, log, or data and log). Note also that a single CREATE or ALTER DATABASE statement may result in multiple fragments and *sysusages* rows when there is not a single continuous space available at the requested size.

Allocation Units and Extents

Each database fragment consists of a number of allocation units, typically a large number of these. Each allocation unit consists of exactly 256 logical pages. (We will call these simply "pages" throughout, though at the end of this article we discuss how logical pages are different from virtual pages.) Pages are allocated to objects one extent at a time: 8 contiguous pages. There are 32 extents in each allocation unit (32 x 8 = 256). An extent is never shared by multiple objects; it can be allocated to only one object at a time. In the context of extent ownership, an object is either a table, a clustered index, a nonclustered index, the set of all large columns of a table, or the database's GAM table.

When an object is created, it is allocated an entire extent. When it needs additional space, the allocation happens again

one extent at a time; however, when inserting large amounts of data with fast bulkcopy, up to 31 extents can be allocated at once in order to reduce allocation overhead. Consecutive extents of an allocation unit may belong to one and the same object or to different objects, depending on the order in which the objects were created, expanded, shrunk, or deleted. When a new extent is allocated, this sometimes translates into an actual availability of fewer than 8 pages. For example, when an object is allocated the first extent of an allocation unit it will "own" only 7 of the pages as the very first page of an allocation unit is always an allocation page.

A table and any of its indexes never share an extent. This has a number of advantages: 1) Maintaining page usage statistics is easier. 2) Dropping tables or indexes only requires deallocating all their extents, and not every page separately. 3) As we will see, the separate sets of extents of these objects can be allocated from different disks to parallelize I/Os, when this is appropriate (with the exception of a clustered index). 4) A table and any of its indexes can each be bound to different data caches, to maximize memory efficiency and reduce the number of avoidable disk I/Os. The page size in ASE is the smallest unit of disk I/O; the extent size is the size of the largest I/O. Large I/O bufferpools can be configured for in any data cache(s).

Segments Defined by ASE

The allocation and deallocation of extents happens automatically as needed, and by default all user tables and their indexes share the same database fragments. When necessary, however, you can make use of segments to specify where certain user tables and/or indexes will grow. A segment is a logical name which refers or points to one or several database fragments for future extent allocation. An object may be allocated new extents only from the database fragments referred by the segment it is bound to.

The *syslogs* table is always bound to the *logsegment*. All other system tables and their indexes are bound to the *system* segment, and by default all user tables and their indexes are bound to the *default* segment.

In creating and expanding databases, it is common to make the *logsegment* point to a separate set of database fragments than the *system* and *default* segments. This is done for recoverability and/or for performance reasons. By default, the *system* and *default* segments point to the same set of fragments. Typically, *tempdb* is the only database for which it is sometimes worthwhile to make the *system* segment point to other fragments than the *default* segment. This increases disk I/O throughput when there is a lot of disk activity on user-created temporary tables. Indeed, temporary tables are bound by default to the *default* segment, while worktables are always bound to the *system* segment. The worktables are the tables which ASE creates in processing certain SQL queries (e.g., GROUP BY).

User-Defined Segments

When a database is created, it always has the three segments we have discussed in the previous section. In addition to these, the DBA can create user-defined segments in order to place a user table and/or any of its nonclustered indexes and/or the set of all its large columns on a separate set of database fragments. This may be done for capacity planning or performance reasons.

A table may grow so quickly that the DBA needs to be warned well before it uses up too much available disk space. In order to limit and monitor its growth this table can be bound to a used-defined segment. It will not be able to grow on, i.e., acquire new extents from, any database fragments other than those pointed to by that segment. Monitoring the remaining capacity of the segment can be automated with freespace thresholds. Multiple freespace thresholds may be placed, for capacity planning, on any one segment, including user-defined segments. When the segment's database fragments have become filled to and past a certain freespace threshold, then the stored procedure associated with this threshold is executed, assuming such a stored procedure has been created and bound to the threshold. This procedure can be used, for example, to send a warning to the DBA.

In order to improve overall disk throughput, it can be useful to parallelize the disk I/Os of a busy table with respect to other user tables of the same database. This table can be bound to a user-defined segment and thus made to grow on a separate disk (or multiple disks). In some perhaps rare

instances of busy tables which experience a lot of disk I/Os, it may prove useful to not only bind this table away from the *default* segment but also bind some of its nonclustered indexes to a separate segment from the table. The potential performance benefit here comes from spreading disk I/O activity over multiple disks and separate pending I/O queues.

Note that this cannot be done with clustered indexes: An APL or DOL table and its clustered index form a single object with respect to segments. When a clustered index is created on a different segment, all of the extents of its table are automatically moved to that segment, and this is often utilized to move a table to a new segment. On the other hand, remember that with respect to owning extents, a table and its clustered index *are* separate objects.

ASE encodes which segments point to a given database fragment by changing the corresponding *segmap* bit to a "1". The *segmap* bitmap is a 32-bit column value in the *sysusages* table: there is one *segmap* per database fragment, and one *segmap* bit per segment. Note that although there are 32 bits in this bitmap, a database can have no more than 31 segments, and therefore no more than 28 user-defined segments.

The Page and Extent Allocation Algorithm

A common occurrence of page allocation happens during row insertion, when the row cannot fit into the page where it belongs. This page is called the *target page*. ASE uses an allocation algorithm which is designed to allocate the new page as close as possible to the target page while minimizing the logical I/Os, i.e., processing time to do so. It performs a sequence of up to six sequential steps to find which page to allocate (and in the process of doing it may also need to allocate a new extent):

1. It looks for a free page on the target page's extent.
2. If there is no such free page, it looks in the same allocation unit for a free page on any other extent which already belongs to this object.
3. Otherwise it searches for a free page in other extents (i.e., in other allocation units) which belong to the object. First it makes use of "allocation hints" stored in the first OAM page of the object. This is a short list of OAM pages which may have references to such extents. (See more on OAM pages below.)
4. If no free page is found this way, ASE may scan all OAM entries for a reference to an extent which is not full and belongs to the object. ASE will do this only if the page utilization, the ratio of used to reserved pages for the object, is low. If the page utilization is high, it may take a significant number of I/Os to scan OAM pages before a free page is found. In that case, Step 4 is skipped. You can configure the PAGE UTILIZATION PERCENT PARAMETER to control where the cut-off is.
5. If a free page has still not been found, an entire extent is allocated to the object and its first page is allocated. Note that this extent is from the segment the object is bound to. As we have seen, the functional meaning of object-segment binding is to point to only those fragments from which extents may be allocated.
6. This last step should not fail if proper capacity planning and monitoring is performed on the object's segment. However, what happens if there are no more free extents available anywhere on this segment? ASE's algorithm makes one last effort to find a free page to allocate. It scans the maintenance information of all other extents that belong to the object. Essentially, it performs the scan of Step 4, this time regardless of page utilization. Usually this only happens when Step 4 was skipped. If not, the scan of step 4 is performed again in the off-chance that a page was deallocated in the meantime. It is only when even step 6 has failed that an out-of-space error message is raised for this segment. The meaning of error 1105 is twofold: There are no free extents left and this particular object has no free pages on any of its own extents.

Allocation, OAM, and GAM Pages

The allocation, OAM, and GAM pages exist primarily for database consistency checking, for corruption repair, and to support the efficiency of the page and extent allocation and deallocation algorithms. The first three steps of the allocation algorithm examine information on the allocation page corresponding to the target page. Steps 3, 4, and 6 make use of the object's OAM page(s), and Step 5 makes use of the database's GAM page(s).

Allocation Pages and Extent Structures

The first page of each allocation unit is called the allocation page and is used to maintain storage information about the allocation unit. It contains primarily a reference bitmap (which we will discuss in the dbcc commands section), allocation hints (see Step 3 above), and 32 extent structures.

Each extent structure corresponds to one of the 32 extents of the allocation unit. The extent structure contains a bitmap of 8 allocation bits, one for each page of the corresponding extent. When a page is allocated this bit is changed to a "1". When it is deallocated, it is changed to a "0". A "0" means that the page is free and can be allocated when the extent-owning object needs another page. The other part of the

extent structure stores the primary key for the extent-owning object: it is identical—with one subtle exception in the case of APL clustered indexes—to the primary key of a segment-bound object. The latter consists of the *id* and *indid* columns of the object's row in the *sysindexes* table, i.e., the object's table id and index id.

Note that all large column values of a table collectively form a single object and share the same extents; we have already seen that they are bound to a single segment. The index id for this object is always "255".

Allocation units and extent structures help save I/Os. Step 1 of the allocation algorithm looks for a "0" allocation bit in the extent structure of the target page; Step 2 does the same for any other extent structures of its allocation unit that belong to the same object. Step 1 requires a single logical I/O. If extent structures did not exist, this step would require the consistent maintenance of an allocation status in each page and the direct investigation of possibly all other pages in the extent, requiring up to seven logical I/Os. Step 2 can quickly find the other extents of the allocation unit belonging to the object by scanning the other 31 extent structures. This prevents the need to directly investigate up to 254 other pages.

Extent structures have other uses, such as extent deallocation and consistency checking. When a table or an index is dropped, each of its extents can be deallocated with minimal work by writing a "0" byte into its extent structure.

OAM Pages and Object Statistics

Maintenance information for each extent-owning object is stored in a single OAM page chain, which consists of one or as many OAM pages as it takes. There is an exception. There are a few tables which have no OAM pages: *syslogs*, *sysgams* (the GAM table), and the dynamic tables. Dynamic tables are tables such as *sysprocesses*, *syscurconfigs*, and *syslogshold* that are maintained in memory only and are not permanent data structures. OAM pages do not have dedicated extents: They are stored in the extents of their object.

An OAM page stores up to 250 OAM entries except for an object's first OAM page which can store only up to 240 OAM entries. It uses the equivalent space of ten such entries to also store several types of object-level information beyond what is stored in the *sysobjects* and *sysindexes* tables. This includes allocation hints and some basic object statistics such as the number of rows, the total number of pages, and the number of allocated and unallocated pages. Prior to release 11.9.2, these object statistics were used by the optimizer and the *sp_spaceused* command; now this information, and much more, is maintained in the new *sysstatistics* and *systabstatistics* tables. The *systabstatistics* table has superseded the distribution page, which used to be part of an index. This type of page no longer exists except during an upgrade process from a release prior to ASE 11.9.2.

OAM Entries

The explanation behind the name "Object Allocation Map" is that the OAM entries of an OAM page chain make up a concise map of all reserved and allocated (used) pages. An extent-owning object has an OAM entry for each allocation unit in which it has at least one extent. Conversely, the object has no OAM entries for any allocation units in which it currently has no extents. An OAM entry stores the address of the allocation unit (the logical page number of the allocation page), the number of allocated pages on it for the object, and the number of unallocated pages. If an object currently has five allocated extents in a given allocation unit, there should be an OAM entry for this allocation unit which indicates that it has a total of 40 *reserved* pages and, for example, 33 *allocated* pages and seven *unallocated* pages.

Step 3 checks the OAM pages referred to in the allocation hints for OAM entries ultimately pointing to a free page. The scans of Steps 4 and 6 potentially scan all of the object's OAM pages for such entries.

OAM entries are also used to calculate the total number of allocated, unallocated and reserved pages for the object. As of release 11.9.2, these entries are also used, in the case of DOL tables, for OAM scans. As the page linkage is not maintained for the datapages of a DOL table, a regular table scan cannot be performed. Instead, the table is traversed via all its extent structures which are found by scanning its OAM entries.

GAM Pages

When an extent is allocated, this means that at least one of its pages is allocated and the extent currently belongs to a given object. As soon as the last used page of an extent is deallocated, the entire extent is also deallocated. Imagine that most extents of an entire database were allocated. In this case, the overhead might be large, given the data structures we have discussed so far, to find a free extent in Step 5 of the page and extent allocation algorithm. ASE would have to examine one allocation page after another, and in each one examine each extent structure, until it could find one free extent.

To make Step 5 very fast, ASE maintains a large bitmap called the Global Allocation Map or GAM. Each GAM bit corresponds to an allocation unit of the database. A "1" value

means that the allocation unit has no free (i.e., unallocated) extents. This does not mean that it has no unallocated pages, but that all extents are currently allocated to objects. In Step 5, ASE simply needs to scan for a "0" GAM bit for the appropriate segment, and then examine the extent structures of the corresponding allocation unit. Because of the "0" GAM bit value, there should be at least one extent structure representing an unallocated extent.

Each GAM page stores 8 x 2016 = 16,128 GAM bits, each of which stores free extent information for 256 pages. In a server with a 2K logical page size, this corresponds to a little short of eight gigabytes of database space. When a database grows beyond this size, more GAM pages are allocated and together they form a GAM page chain. Of course, not all GAM bits are actively used in the last or only GAM page of a database but only as many as the number of allocation units in the allocated database fragments. An entire extent (at least one) is allocated to the GAM table, even if there is only one GAM page.

Database Consistency Checking

As mentioned earlier, the allocation, OAM and GAM pages are also used for consistency checking. So are various fields in the headers of all database pages, their row number fields, and their row offset table when it exists (it does not exist in APL indexes). The combination all of this maintenance information is sufficiently redundant to detect database corruption. A lot of checking happens automatically as ASE traverses dynamic and permanent data structures in processing regular queries and system functions.

For example, as page chains or index trees are traversed, ASE checks that each accessed page has the expected object id, index id, and index level values in its header. Another example involves the access of a data row via a nonclustered index. The row id found in the leaf level is used to find the start of the data row via the row offset table. This row id is then compared to the value stored in the row id field of the data row. If they do not match, error message 2509 is raised.

In the vast majority of cases, database corruption is caused by factors outside of ASE: for example, reliability problems with the power supply, hardware, or O/S, and administrator errors made in the creation of raw partitions or the allocation of devices. If one of these causes corrupts a lot of information in various locations, the automatic consistency checking may report a problem soon. When the corruptions are few, it is more likely that the problem will remain undetected for a longer time, and yet the business consequences may be very costly. It is therefore necessary to have a well-thought-out schedule for running DBCC commands, and adhere to it, just as is the case for database backups.

As long as this is practical, it is recommended to also check database consistency prior to each backup. Incidentally, in releases prior to 12.0, the Backup Server backs up only the allocated pages of a database; since, this is configurable with Sybase Central or with the *sp_dumpoptimize* procedure.

DBCC Commands

There are many DBA commands under the rubric DBCC or "Database Consistency Checker." A few of these are used for regular maintenance consistency checking. *dbcc checktable* checks, among other things, the match between rows and the row offset table, that index keys in the index pages are in the correct order, that index pointers are consistent, and that various maintenance fields have reasonable values. *dbcc checkdb* performs the corresponding checks for all the tables in a database. *dbcc checkcatalog* performs several checks for consistency within and between a database's system tables, e.g., it checks that for every table in *sysobjects* there is at least one row in *syscolumns*. The *dbcc tablealloc*, *indexalloc* and *checkalloc* commands check the consistency of OAM pages, OAM entries, page chain linkages and page allocations. They also come with a fix option which allows for the automatic fixing of some errors which are readily remedied by ASE. Note that *dbcc checkalloc* with the *fix* option turned on requires *single-user* mode for the database.

Page allocation checking is speeded up by the use of the 256 page reference bits in each allocation page. While one of the allocation DBCC commands scans all the pages of an index, table or set of large columns, the page reference bit corresponding to each scanned page is changed to "1". When the scan is completed, the page allocation bits of all

the object's extent structures are compared against the corresponding reference bits, and their values should all match. (Remember that each time a page is allocated or deallocated its allocation bit in the corresponding extent structure is set to "1" or "0", respectively.) If a page is found to be referenced but not allocated, error message 2521 is raised; if it is found to be allocated but not referenced, error message 2540 is raised.

Should you find consistency errors, please consult the *Troubleshooting and Error Messages Guide* and follow its instructions. They may refer you to Sybase Technical Support. In a number of instances, you will need a recent (not corrupted) database backup. And of course you will want to find out what was the original cause of the corruptions and address it, especially if they recur.

Unfortunately the completion of the regular dbcc commands requires a lot of processing and they may run for a long time when the table or database is large. This performance issue was addressed in release 11.5 by the introduction of the *dbcc checkstorage* command, which is also called "parallel dbcc." It achieves a much greater efficiency by making use of parallel table and index scans—which were also introduced in that release—and of dedicated data caches, user-defined segments, and a performance-improved checking algorithm which, among other things, scans any page of the database only once and skips some less critical checks.

Data Corruption vs. Database Corruption

Note that there can also be data corruption when there is no database corruption. All internal maintenance information is correct, but some of the values in the data columns are not. These might be caused by errors in the client software logic or the data itself prior to input into ASE. This type of corruption is typically not detected by the dbcc commands—which have no knowledge of the business meaning of the data field values—with minor exceptions, such as when the data corruption changes a clustered index key value and puts it out of clustered order. Data corruption may be caught by primary key, unique, check and reference constraints, and by business rules checking logic in triggers, stored procedures, middleware, and/or the client application.

Logical Page Sizes in ASE 12.5

Prior to release 12.5, the logical and virtual page sizes were both 2K and not configurable (with a now obsolete 4K exception for certain platforms). Now the virtual page size is still 2K and not configurable. However, the logical page size for the entire server can be chosen once and for all at server installation time and can be 2K, 4K, 8K, or 16K. This release allows for greater maximum sizes for various datatypes such as *char* and *varchar*, even in 2K servers.

Servers with larger logical sizes support even larger limits for various datatypes, as well as longer row lengths. In either case, the benefits include the fact that sometimes the need for a text column can be eliminated. The larger logical page sizes can make large I/Os more efficient (e.g., a single I/O can retrieve up to 128K in a 16K server) but result in more disk overhead for most objects, because of significant amounts of unused space in many of the database's pages, including system table pages. For example, in an 8K server, a table with just 1.5K of data will have an entire extent allocated: most of the first 8K page will be empty, and the remaining seven pages (56K) will be unusable by other objects.

In a 12.5 server with a 4K logical page size, there are still 8 pages per extent and 256 pages per allocation unit, but these structures are now 32K and 1MB in size, respectively.

What is a virtual page, whose size is always 2K, when the logical page size is larger than 2K? It is still the same as it always was: A concept related to the 4-byte virtual page number, which is unique across the entire server. The first byte is the logical device's virtual device number or *vdevno*. The other three bytes (24 bits) represent a unique page number within a device. There are still at most 2 to the power 24 = 16,777,216 virtual pages per device, which translates into a maximum logical device size of 32 GB.

How do virtual pages correspond to logical pages in an 8K server? You can observe this by examining the *sysusages.vstart* and *sysdevices.low* columns: They will always be multiples of four. For every logical page there are, in this addressing scheme, four virtual pages; the page number of the first one serves as the virtual address of the 8K page.

Maximum Server Size Limits

An ASE server may have up to 256 database devices. The maximum size for a logical device is as large as the platform will support, up to 32 GB. The maximum possible size for an ASE server is therefore 8 TB. These limits are based on the 4 byte addressing scheme encoded in virtual page numbers. A single database may have up to 128 database fragments. The maximum size of database fragments is limited only by the maximum size for a logical device. The maximum possible size for a single database is therefore 4 TB. None of these maximum numbers have changed in release 12.5, even when a larger logical page size is chosen. However, these limits are very high with respect to most current databases. ■