An Introduction to Semantic Partitioning in ASE 15.0

By Stefan Karlsson and Dhimant Chokshi

The first of a series of articles outlining the features of the latest ASE release

ASE 15.0 is the next major version of Sybase's flagship product Adaptive Server Enterprise, first announced at TechWave 2004. There are many major and minor features planned for this version. To introduce these features a series of articles is planned, beginning with this discussion on Semantic Partitioning.

What are Partitions?

Partitioning is a divide-and-conquer approach to improving query performance, operational scalability, and managing ever-increasing amounts of data. A large table can be broken up into smaller chunks or partitions. The data in each partition can then be processed and managed separately. Maintenance becomes faster, either by operating in parallel on all partitions concurrently or by restricting the operation to one partition.

Partitioning was introduced in Sybase ASE in 1995 as one of many features of 11.0. At that time it served the purpose of allowing multiple concurrent inserters on a heap table. Another aspect of the feature was to support the upcoming parallelism introduced in 11.5. Operations can be done in parallel with one worker thread per partition; hence a four-way partitioned table implies that queries and maintenance operations are four times faster.

The big difference between the pre-15.0 partitioning and ASE 15.0's *semantic partitioning* is that the latter not only gives the users full control over data placement, but also offers different schemes for how the data should be partitioned. ASE 15.0 partitions are fully manageable objects; they are named entities, designed and controlled by users. Control is exercised through specifying conditions on an assigned *partition key*, comprised of one or more columns in the table. The most intuitive scheme or strategy is to partition by range, where the table is split into parts by the range of values in the specified partition key.

Benefits of Semantic Data Partitioning

With semantic data partitioning, each partition is managed separately by the DBMS as well as the DBA. Its benefits include:

- Increased data availability
- Making large data sets easier to manage
- Increased database scalability and performance

Data availability is enhanced because maintenance windows for individual partitions are smaller, and, should a partition become unavailable, the remaining partitions would be unaffected. With control designed into semantic partitioning and smaller chunks of data to manage, DBAs have a more productive tool as well as facilities that reduce the consequences of errors. Maintenance can be done on a partition level, greatly speeding up operations such as **update statistics**.

Scalability is also improved because inter-partition parallelism facilitates developing systems with more users, transactions and data volumes by spreading I/O usage. This means faster operations and shorter response times.

Stefan Karlsson is a technical evangelist with ASE performance engineering and can be reached at stefan.karlsson@sybase.com. Dhimant Chokshi is a technical architect with Sybase engineering and can be reached at dhimant.chokshi@sybase.com.

Increased Availability

Partitions do two things to increase data availability. First, because partitions are independent, they can be maintained separately. This gives us smaller maintenance windows, meaning less downtime to work on any one partition. Second, when unplanned events occur, such as device failure, only the partitions on the affected device are rendered unavailable—the other partitions remain available as before.

The increased availability of data can for many businesses be translated into cost-savings. Shorter downtimes in case of storage problems cut the costs. There's also the business requirement to reduce maintenance windows so that data is available for users and applications. With user control over data placement comes the knowledge of what data is actively used. Commands and operations have been extended to allow analysis and manipulation of individual partitions. Since users are dealing with a smaller amount of data, operations on a partition are faster than operations on entire tables.

Both applications and maintenance operations all benefit from the use of partitioned tables. In one case, it may mean that the OLTP system predominantly uses the partition with recent and current data, while a DSS-type application uses other partitions with historical data. Another scenario may include ongoing data loads into separate partitions and storage during ongoing querying and transactions.

Making Large Data Sets Easier to Manage

Data volumes are ever increasing. There are numerous reasons for this, such as data models that cover additional subject areas, more details about the subject area, or more stringent business and legal requirements. Larger data volumes can often require more and more system resources until the system is saturated. At the same time, the DBAs who manage these systems are being asked to increase productivity.

Raw power helps, but only so far, and it may not be cost-effective. Here a partitioning strategy makes good sense. Compared to unwieldy tables and the costly consequences of user errors, managing small, well-defined sets of data is far more intuitive and far less risky.

Of course, the larger the amounts of data, the more time and resources a maintenance task consumes. Performing a certain task on a partition is faster than on a table. Range-based partitioning can help further by separating historical data from current and more volatile information. More static data requires less—if any—maintenance, while operations on the active data are significantly faster.

Increased Scalability and Performance

As with manageability, partitioning leads to business-level benefits through improved scalability and performance. Parallelism, inherent in semantic partitioning, not only improves scaling of operational functions but also query performance. In addition, decreased footprint and increased precision lead to significant improvements in performance. Let's start with a look at the decreased footprint.

Since partitions are smaller than the entire table, fewer resources are used. Local indexes (discussed later) span data in one partition and are leaner structures than the table-level indexes. Leaner indexes mean fewer resources to maintain and lower access costs; the latter is of special importance in joins. When tables are partitioned using the same columns and partitioning scheme, the directed joins between these tables are very efficient.

Partitioning greatly improves the degree of parallelism for operations in ASE. Examples of enhanced parallelism include: multiple threads working on different partitions in parallel queries, resulting in faster query execution times; concurrent administrative operations working on different partitions in parallel resulting in faster maintenance operations; and parallel I/O streams to access data stored on different partitions enabling faster I/O operations.

In ASE 15.0, the optimizer is partition-aware, leading to increased precision in decisions and hence better query performance. Statistics describe the physical layout of the partition itself—such as pages, cluster ratios, etc., as well as data distribution statistics kept per partition—all supporting increased precision in the optimizer's evaluation of different plans. In short, where statistics for non-partitioned tables are kept per table and index, the statistics for partitioned tables are kept per partition (and index partition). Partition statistics means that the optimizer will more accurately cost statements that access partitioned tables.

Partitioning Concepts and Details

As discussed above, partitions are named objects controlled and managed by users. The different partitioning schemes include:

- **Round-robin partitioning:** This scheme corresponds to the pre-15.0 partitioning.
- **Range partitioning:** Each partition holds rows that have partition key values within a user-defined range.
- **Hash-based partitioning:** A semantic alternative to round-robin partitioning to achieve even data distribution.
- **List partitioning:** The partition condition explicitly defines the value list for each partition.

Defining Partitions

In the example below, a table is created, range-partitioned on the partition key **c_custkey**. When data rows are loaded or inserted into the table, the rows are placed in the partition given by their respective values for the column **c_custkey**:

```
CREATE TABLE customer (
    c_custkey        INTEGER            NOT NULL
    , c_name         VARCHAR(25)        NOT NULL
    , c_address      VARCHAR(40)        NOT NULL
    , c_nationkey    INTEGER            NOT NULL
    , c_phone        CHAR(15)           NOT NULL
    , c_acctbal      DOUBLE PRECISION   NOT NULL
    , c_mktsegment   CHAR(10)           NOT NULL
    , c_comment      VARCHAR(117)       NOT NULL
)
PARTITION BY RANGE ( c_custkey ) (
    cust_ptn1 VALUES <= (20000)         ON segment1
    , cust_ptn2 VALUES <= (40000)       ON segment2
    , cust_ptn3 VALUES <= (60000)       ON segment3
)
```

An existing table can be partitioned using the **alter table... partition** command. This includes the same **partition** clause as the **create table** command. If there is data in the table when it is partitioned, the rows will be moved to the right partitions. To add another partition to our sample customer table:

```
ALTER TABLE customer
ADD PARTITION (
        cust_ptn4 VALUES <= (80000)     ON segment4
)
```

Note that partitions can only be added as new upper boundary to range partitioned tables. For other scenarios, use the **alter table... partition by** command.

The system procedure **sp_helppartition** provides information on partitioned tables:

```
1> EXEC sp_helppartition customer
2> go
name      type        partition_type partitions   partition_keys
--------  ----------  -----------------------------  -----------------
customer  base table  range                        4 c_custkey

(1 row affected)

partition_name  partition_id  pages  segment    create_date
--------------  ------------  -----  ---------  ------------------
cust_ptn1       1093746769    1      segment1   Sep 27 2004  6:53PM
cust_ptn2       1109746826    1      segment2   Sep 27 2004  6:53PM
cust_ptn3       1125746883    1      segment3   Sep 27 2004  6:53PM
cust_ptn4       1269747396    1      segment4   Sep 27 2004  6:53PM

Partition_Conditions
------------------------
VALUES <= (20000)
VALUES <= (40000)
VALUES <= (60000)
VALUES <= (80000)

Avg_pages  Max_pages  Min_pages  Ratio(Max/Avg)  Ratio(Min/Avg)
---------  ---------  ---------  --------------  --------------
1          1          1          1.000000        1.000000
(return status = 0)
```

Here we can clearly see the partitioning scheme as well as the partition conditions. But there's another useful piece of information here—the sizes per partition. Obviously, this empty table is perfectly balanced, but for tables already holding data this is very useful.

Data Placement

One important part of the management and control is the *data placement*, here illustrated by the **ON segment_name** clause in the **create** and **alter table** commands. The "segment" is the concept employed by ASE to explicitly place an object—table, index, LOB column—on designated disks. It has now been extended to include the management of partitions.

With JBOD storage (Just a Bunch Of Disks, storage subsystem with independently accessible disks), the purpose of segments is fairly clear: to separate IOs over several disks. But what about segments role in SAN configurations then, especially in the case when DBAs have limited control over how LUNs span physical disks?

First and foremost, the improved manageability segments provide, such as thresholds, is fully valid in SAN environments, as is the more efficient allocation. There's potential cost savings too! The cost of LUNs typically varies with performance requirements, and by employing the cheaper LUNs for more historical data and high performance LUNs only for the active partition(s) storage costs are decreased.

Indexing

Indexes continue to be a key physical structure for performance, whether the table is partitioned or not. In the case of a partitioned table, an index can be global or local. A *global index* spans all data rows in all partitions, while a *local index* is built on the rows in a partition. The benefit of local indexes is that they are smaller and have fewer levels. Hence, they are faster for access—another performance benefit.

Clustered indexes are always local on range, hash and list partitioned tables, while they always are global on round-robin partitioned tables. Non-clustered indexes can be local or global. See the following examples:

```
1> CREATE UNIQUE CLUSTERED INDEX custkey_cidx ON
   customer ( c_custkey )
2> LOCAL INDEX
3>     custkey_cidx_ptn1 ON segment1
4>     , custkey_cidx_ptn2 ON segment2
5>     , custkey_cidx_ptn3 ON segment3
6>     , custkey_cidx_ptn4 ON segment4
7> go
1> CREATE UNIQUE INDEX custkey_ncidx ON customer( c_custkey )
2> go
1> CREATE NONCLUSTERED INDEX name_ncidx ON customer ( c_name )
2> LOCAL INDEX
3> go
1> EXEC sp_helpindex customer
2> go
Object has the following indexes
```

index_name	index_keys	index_description	index_max_rows_per_page	index_fillfactor	index_reservepagegap	index_created	index_local
custkey_ncidx	c_custkey	nonclustered, unique	0	0	0	Sep 27 2004 6:54PM	Global Index
custkey_cidx	c_custkey	clustered, unique	0	0	0	Sep 27 2004 6:54PM	Local Index
name_ncidx	c_name	nonclustered	0	0	0	Sep 27 2004 6:54PM	Local Index

(3 rows affected)

index_ptn_name	index_ptn_seg
custkey_cidx_ptn1	segment1
custkey_cidx_ptn2	segment2
custkey_cidx_ptn3	segment3
custkey_cidx_ptn4	segment4
custkey_ncidx_194267567	default
name_ncidx_210267624	default
name_ncidx_226267681	default
name_ncidx_242267738	default
name_ncidx_258267795	default

(9 rows affected)
(return status = 0)

Here the clustered index, **custkey_idx**, is created as a local index. The segment specification is required to keep the assigned data placement, as creating a clustered index moves the table to that segment whether the table is partitioned or not. If the **local index** clause had been left out, the index would have been local, and the table and index placed on the default segment.

The non-clustered index **custkey_ncidx** is created as global. Since the DDL statement for the index **name_ncidx** includes the **local** clause, it's created as a local index with four partitions. Note that having two indexes on the same column without any other columns is of very limited use. It's included here for illustrative purposes only.

The Partition-Aware Optimizer

At the core of query performance is the optimizer. In ASE 15.0, the optimizer is *partition-aware*. First of all, there are statistics describing the physical layout of the partition itself, such as pages, cluster ratios etc. There are also data distribution statistics kept per partition for increased precision in the optimizer's evaluation of different plans. In short, where statistics for non-partitioned tables are kept per table and index, the statistics for partitioned tables are kept per partition (and index partition). *Partition statistics* help the optimizer more accurately cost statements that access partitioned tables.

Based on this data, the optimizer not only has increased knowledge about a partition's physical and logical layout, but also uses the concept of partition in evaluating plans, whether serial or parallel plans. Given predicates on partition columns, which can be indirect through join transitive closure, the optimizer will eliminate all partitions but the ones directly involved in the optimization and execution. As described above, the optimizer can eliminate partitions during the optimization phase, or *compile time elimination*.

Furthermore, the execution engine is fully aware of partitioning and uses this to the fullest. If partitions can't be eliminated during compile time, for instance, in the case of an unknown value such as a local variable, partitions will be eliminated while executing the statement, or *run-time elimination*.

Joins benefit from partitioning too. Instead of joins across large tables and indexes, partitions can be joined—directed joins—which helps query performance.

Besides partition elimination and local indexes, the optimizer will use semantic partitioning to the fullest when parallelism is enabled. Since partitioning is a founding block of parallelism, it will help both speed up as well as scale up

the overall system. ASE 15.0's optimizer is also more flexible when it comes to leveraging parallelism and partitions. But we'll save that for a future article.

Operational Scalability

As we have already discussed, partitioning truly helps operational scalability for large databases. A key to this is that various maintenance operations are either parallelized or can be run on different partitions concurrently. Some maintenance operations benefit from *server-side parallelism*. **create index** can, as in pre-15.0, run in parallel to speed up the creation of indexes. Given the increased control over data placement, this can greatly speed up the operation. In ASE 15.0, **reorg rebuild** is parallel as well, another performance boost.

Other maintenance operations can be done on single partitions: other flavors of **reorg** (including **reorg rebuild <table name> <index name>**), **update statistics**, and **truncate table**. This means less time needed for service windows since only one partition is accessed. Also, **update statistics** can be done on several partitions in parallel, with one invocation per partition. This is referred to as *client-side parallelism*.

Range Partitioning

Range partitioning means that a partition holds rows where the value(s) of the partition key is within a certain range, e.g., an order table can be partitioned over order-date and have one partition per month or quarter. Queries that specify—directly or indirectly—a date or date range only touch the concerned partitions.

Following is an example in which the orders table is range-partitioned on the order date. The table has one partition per quarter in the current year and a partition to store all data from previous years.

```
CREATE TABLE orders (
        o_orderkey          INTEGER             NOT NULL
        , o_custkey         INTEGER             NOT NULL
        , o_orderstatus     CHAR(1)             NOT NULL
        , o_totalprice      DOUBLE PRECISION    NOT NULL
        , o_orderdate       SMALLDATETIME       NOT NULL
        , o_orderpriority   CHAR(15)            NOT NULL
        , o_clerk           CHAR(15)            NOT NULL
        , o_shippriority    INTEGER             NOT NULL
        , o_comment         VARCHAR(79)         NOT NULL
)
PARTITION BY RANGE (o_orderdate ) (
    hist_ptn VALUES <= ('December 31, 2003')    ON hist_seg
    , q1_ptn VALUES <= ('March 31, 2004')       ON q1_seg
    , q2_ptn VALUES <= ('June 30, 2004')        ON q2_seg
    , q3_ptn VALUES <= ('September 30, 2004')   ON q3_seg
)
```

The **hist_ptn** points out a really appealing aspect of range partitioning: the separation of current and active data from historic, and the separation of volatile data from static data. If data in this partition isn't modified to any extent there is lessened need of maintenance, and therefore there no need to update that statistics on that partition. When the need arises, we can add a **q4_ptn** for the rest of the year, but before that, all rows with a partition key value greater than September 30, 2004 will be rejected.

List-Based Partitioning

Instead of specifying a range that the partition key is evaluated against, *list-based partitioning* is basically an IN-list per partition. One partition may hold rows with one particular value for the partition key, while another holds rows matching a list of values.

To illustrate list partitioning, we use the *nation* table. It's a good example for textbook illustration purposes, but in real life it's too small to make a performance difference.

```
CREATE TABLE nation (
        n_nationkey         INTEGER             NOT NULL
        , n_name            CHAR(25)            NOT NULL
        , n_regionkey       INTEGER             NOT NULL
        , n_comment         VARCHAR(152)        NOT NULL
)

PARTITION BY LIST ( n_regionkey ) (
        region_ptn1 VALUES ( 1 ) -- 1 is AMERICAS
        , region_ptn2 VALUES ( 2 ) -- 2 is ASIA
        , region_ptn3 VALUES ( 3 ) -- 3 is EUROPE
        , region_ptn4 VALUES ( 0, 4, 5 ) -- All other regions
)
```

As with range-based partitioning scheme, the partition condition plays the role of a check constraint preventing rows with values for **n_regionkey** outside the range from 0 to 5.

Hash-Based Partitioning

The *round-robin* data distribution from pre-ASE 15.0 is still supported. It is not a semantic partitioning scheme. However, one useful aspect of the round-robin partitioning scheme is that it can provide an even distribution of the rows over partitions.

A semantic alternative to achieve an even distribution is the 15.0 *hash-based* partitioning. This means that an ASE hash function is applied to control which partition a data row should reside in. With this scheme transaction, lengths and other factors does not play a role in balancing data distribution over partitions. The *hash-based* partitioning scheme is most useful to achieve an even distribution of data over partitions, especially for large tables that don't have obvious partition key candidates. Added to this is the benefit of directed joins between tables that are hash-based partitioned on the same set of columns.

The example below is one of two tables in a pair that are commonly accessed together. To help performance—for operational commands and queries/transactions—these two tables are both hash partitioned on the column **p_partkey**.

```
CREATE TABLE part (
        p_partkey               INTEGER             NOT NULL
        , p_name                VARCHAR(55)         NOT NULL
        , p_mfgr                CHAR(25)            NOT NULL
        , p_brand               CHAR(10)            NOT NULL
        , p_type                VARCHAR(25)         NOT NULL
        , p_size                INTEGER             NOT NULL
        , p_container           CHAR(10)            NOT NULL
        , p_retailprice         DOUBLE PRECISION    NOT NULL
        , p_comment             VARCHAR(23)         NOT NULL
)

PARTITION BY HASH ( p_partkey ) (
        part_ptn1               ON                  segment1
        , part_ptn2             ON                  segment2
        , part_ptn3             ON                  segment3
        , part_ptn4             ON                  segment4
)

CREATE TABLE partsupp (
        ps_partkey              INTEGER             NOT NULL
        , ps_suppkey            INTEGER             NOT NULL
        , ps_availqty           INTEGER             NOT NULL
        , ps_supplycost         DOUBLE PRECISION    NOT NULL
        , ps_comment            VARCHAR(199)        NOT NULL
)

PARTITION BY HASH ( ps_partkey ) (
        partsupp_ptn1           ON                  segment1
        , partsupp_ptn2         ON                  segment2
        , partsupp_ptn3         ON                  segment3
        , partsupp_ptn4         ON                  segment4
)
```

A Query Performance Analysis

To exemplify the use of partitioning, the *customer* and *orders* tables were chosen. The partitioned copies of the tables were suffixed with **_partitioned**. The *orders* table has 30 million rows and uses 5.7 GB of disk space, and the *customer* table has 3 million rows and uses 590 MB of disk space. This test doesn't illustrate the case where current data is separated from historical data, but since the partition key defined—the column **custkey** in both cases—is used both for range conditions as well as for joins, it's a good choice.

```
CREATE TABLE customer_partitioned (
        c_custkey               INTEGER             NOT NULL
        , c_name                VARCHAR(25)         NOT NULL
        , c_address             VARCHAR(40)         NOT NULL
        , c_nationkey           INTEGER             NOT NULL
        , c_phone               CHAR(15)            NOT NULL
        , c_acctbal             DOUBLE PRECISION    NOT NULL

        , c_mktsegment          CHAR(10)            NOT NULL
        , c_comment             VARCHAR(117)        NOT NULL
)

PARTITION BY RANGE ( c_custkey ) (
        cust_ptn1 VALUES <= (375000)
        , cust_ptn2 VALUES <= (750000)
        , cust_ptn3 VALUES <= (1125000)
        , cust_ptn4 VALUES <= (1500000)
        , cust_ptn5 VALUES <= (1875000)
        , cust_ptn6 VALUES <= (2250000)
        , cust_ptn7 VALUES <= (2625000)
        , cust_ptn8 VALUES <= (3000000)
)

CREATE TABLE orders_partitioned (
        o_orderkey              INTEGER             NOT NULL
        , o_custkey             INTEGER             NOT NULL
        , o_orderstatus         CHAR(1)             NOT NULL
        , o_totalprice          DOUBLE PRECISION    NOT NULL
        , o_orderdate           SMALLDATETIME       NOT NULL
        , o_orderpriority       CHAR(15)            NOT NULL

        , o_clerk               CHAR(15)            NOT NULL
        , o_shippriority        INTEGER             NOT NULL
        , o_comment             VARCHAR(79)         NOT NULL
)

PARTITION BY RANGE ( o_custkey ) (
        ord_ptn1 VALUES <= (375000)
        , ord_ptn2 VALUES <= (750000)
```

```
        , ord_ptn3 VALUES <= (1125000)
        , ord_ptn4 VALUES <= (1500000)
        , ord_ptn5 VALUES <= (1875000)
        , ord_ptn6 VALUES <= (2250000)
        , ord_ptn7 VALUES <= (2625000)
        , ord_ptn8 VALUES <= (3000000)
)
```

Sample Tests

Test 1

Following is a demonstration of partition elimination on the range-partitioned *order* table. While an index scan would be the realistic access method chosen, for illustration purposes a table scan is forced. But since there's a partition condition on **o_custkey**, ASE will only scan that partition for the *orders_partitioned* table.

First, the non-partitioned *orders*:

```
1> SELECT * FROM orders ( INDEX 0 ) WHERE o_custkey = 2999999
2> go
.
.
.
Table: orders scan count 1, logical reads: (regular=1723232 apf=0
total=1723232), physical reads: (regular=8 apf=215396 total=215404),
apf Ios used=215396
Total writes for this command: 0
```

Next up is the same statement, this time on the eight-way partitioned *orders_partitioned*:

```
1> SELECT * FROM orders_partitioned ( INDEX 0 ) WHERE o_custkey =
2999999
2> go
QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root
The type of query is SELECT.
ROOT:EMIT Operator

  |SCAN Operator
  |   FROM TABLE
  |   orders_partitioned
  |   [ Eliminated Partitions : 1 2 3 4 5 6 7 ]
  |   Table Scan.
  |   Forward Scan.
  |   Positioning at start of table.
  |   Using I/O Size 16 Kbytes for data pages.
  |   With MRU Buffer Replacement Strategy for data pages.
.
.
.
Table: orders_partitioned scan count 1, logical reads: (regular=215269
apf=0 total=215269), physical reads: (regular=8 apf=26905
total=26913), apf IOs used=26901
Total writes for this command: 0
```

We can see in the SHOWPLAN output that ASE optimizer recognizes the compile-time partition elimination opportunity and eliminates all partitions but the one holding rows with the value 2999999. In addition, we see that the gain in logical IOs is impressive and predictable. *orders_partitioned* is a copy of the *orders* table, but has eight partitions. The logical IOs for the partitioned table is 1/8 of the non-partitioned case.

The difference in response times varies with cache configurations, disk layouts, and overall system performance, but it's safe to conclude that partitioned-based access is much faster than non-partitioned. In a case where certain partitions are frequently accessed, the smaller footprint means more efficient use of caches and potentially an even higher gain than the eight times we saw here.

Test 2

Test 2 is a more interesting scenario. The query is to get all customer and order data for a specified set of customers that placed an order on August 1, 1992. Clearly the optimizer will do partition elimination from the condition on **c_custkey**. The first piece of useful data is that partition elimination occurs on the customer as well as on orders, the latter through join transitive closure. Secondly, the optimizer chooses different joins depending on whether the tables are partitioned or not. First, the non-partitioned case:

```
1> SELECT * FROM customer, orders
2> WHERE c_custkey = o_custkey
3> AND o_orderdate = "Aug  1 1992"
4> AND c_custkey >= 2950000
5> go
.
.
.
  |MERGE JOIN Operator (Join Type: Inner Join)
  |   Key Count: 1
  |   Key Ordering: ASC
  |
  |      |SCAN Operator
  |      |   FROM TABLE
  |      |   customer
  |      |   Using Clustered Index.
  |      |   Index : customer_x
  |      |   Forward Scan.
```

```
    |    |    Positioning by key.
    |    |    Keys are:
    |    |         c_custkey  ASC
    |    |    Using I/O Size 16 Kbytes for data pages.
    |    |    With LRU Buffer Replacement Strategy for data pages.
    |
    |    | SCAN Operator
    |    |    FROM TABLE
    |    |     orders
    |    |    Index : oi_ckey_fat
    |    |    Forward Scan.
    |    |    Positioning by key.
    |    |    Keys are:
    |    |         o_custkey  ASC
    |    |    Using I/O Size 16 Kbytes for index leaf pages.
    |    |    With LRU Buffer Replacement Strategy for index leaf pages.
    |    |    Using I/O Size 16 Kbytes for data pages.
    |    |    With MRU Buffer Replacement Strategy for data pages.
.
.
.
Table: customer scan count 1, logical reads: (regular=4072 apf=0
total=4072), physical reads: (regular=8 apf=509 total=517), apf IOs
used=505
Table: orders scan count 1, logical reads: (regular=5977 apf=0
total=5977), physical reads: (regular=221 apf=736 total=957), apf IOs
used=736
Total writes for this command: 0
```

Since both tables have indexes on **custkey**, the optimizer uses the known ordering to do a merge join. This especially helps the performance in the case of non-partitioned tables by keeping the number of scans of *customer* down to 1, which is a huge gain! The end result is the same number of logical IOs on the *orders* table as in the case of the partitioned plan below. Let's see what this becomes when the tables are partitioned:

```
1> SELECT * FROM customer_partitioned, orders_partitioned
2> WHERE c_custkey = o_custkey
3> AND o_orderdate = "Aug  1 1992"
4> AND c_custkey >= 2950000
5> go
.
.
.
  |NEST LOOP JOIN Operator (Join Type: Inner Join)
  |
  |    | SCAN Operator
  |    |    FROM TABLE
  |    |    orders_partitioned
  |    |    [ Eliminated Partitions : 1 2 3 4 5 6 7 ]
  |    |    Index : oi_ckey_fat
  |    |    Forward Scan.
  |    |    Positioning by key.
  |    |    Keys are:
  |    |         o_custkey  ASC
  |    |    Using I/O Size 2 Kbytes for index leaf pages.
  |    |    With LRU Buffer Replacement Strategy for index leaf pages.
  |    |    Using I/O Size 2 Kbytes for data pages.
  |    |    With LRU Buffer Replacement Strategy for data pages.
  |
  |    |RESTRICT Operator
  |    |
  |    |  |SCAN Operator
  |    |  |    FROM TABLE
  |    |  |    customer_partitioned
  |    |  |    [ Eliminated Partitions : 1 2 3 4 5 6 7 ]
  |    |  |    Using Clustered Index.
  |    |  |    Index : customer_x
  |    |  |    Forward Scan.
  |    |  |    Positioning by key.
  |    |  |    Keys are:
  |    |  |         c_custkey  ASC
  |    |  |    Using I/O Size 2 Kbytes for data pages.
  |    |  |    With LRU Buffer Replacement Strategy for data pages.
.
.
.
Table: orders_partitioned scan count 1, logical reads: (regular=5977
apf=0 total=5977), physical reads: (regular=8 apf=2485 total=2493),
apf IOs used=2485
Table: customer_partitioned scan count 224, logical reads: (regular=672
apf=0 total=672), physical reads: (regular=0 apf=0 total=0), apf IOs
used=0
Total writes for this command: 0
```

ASE does a nested loop join with *orders* as the outer table. This makes sense because the condition on **o_orderdate** further limits the result set. It's the same number of logical IOs on *orders* as in the unpartitioned case, which makes sense even though all but one partition has been eliminated. The big gain comes for the access to the customer table. Here two mechanisms play: partition elimination and local indexes. The latter means a leaner index with fewer levels. While the table is only accessed 224 times during the NL join, it still proves the value of local indexes.

Conclusion

ASE 15.0's new partitioning feature allows users to benefit from shorter maintenance windows and increased data availability, better control, and greater parallelism for more precise optimizer decisions and improved data placement. To see a version of this article including more tests that demonstrate the full benefits of partitioning, go to the ISUG website at www.isug.com/journalonline. ■