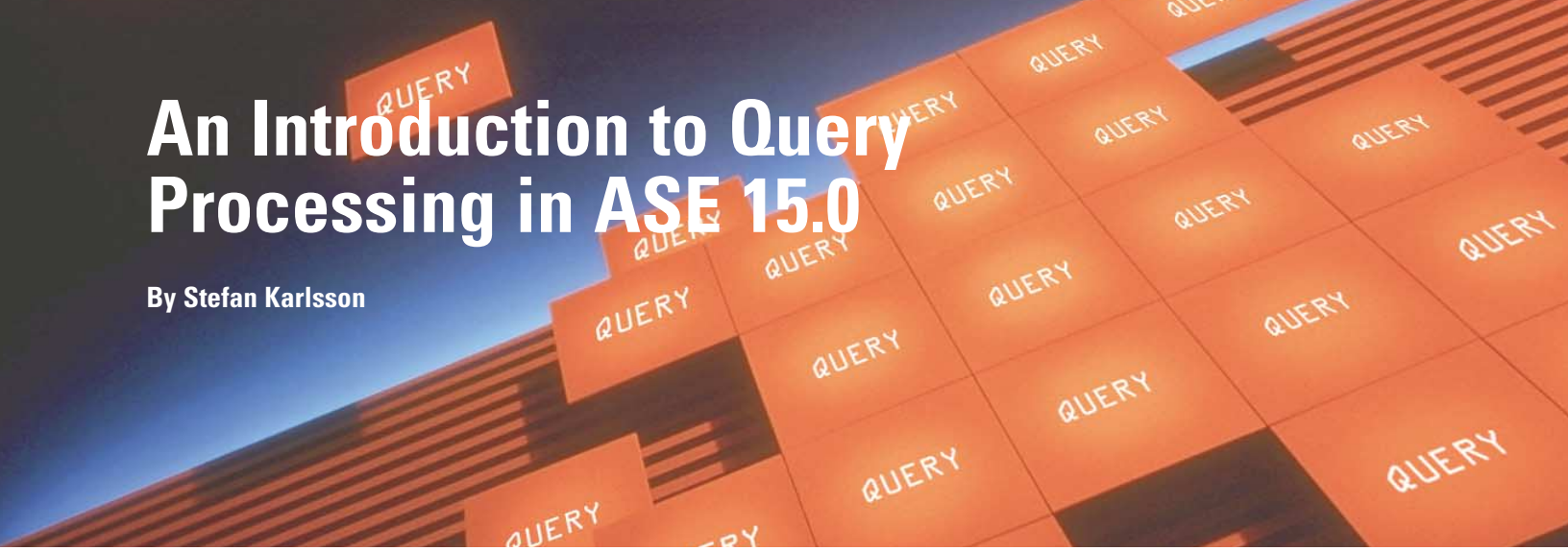

An Introduction to Query Processing in ASE 15.0

By Stefan Karlsson

The second article in a series describing the new ASE 15 release

The previous article in this series on the forthcoming release of ASE 15 in discussed the new semantic partitioning feature within the access and store layers of ASE. In that article we touched on the subject of this article, namely the optimizer and execution engine. The next article will go into further details on these components and present query tuning implications in using this state-of-the-art machinery, as well as information on the new toolset for query tuning.

This article will first introduce the optimizer and the execution engine, and show how they facilitate the query processing of ASE. Then we'll look at some of the new capabilities such as hash-based functionality and new join methods in the building blocks section. Lastly, we will tie the building blocks together and show how the different features interact to achieve the design goals of the query processing improvements in ASE 15.

What are the Optimizer and Execution Engine?

The optimizer is much talked about, but it goes hand-in-hand with the execution engine. The execution engine does what the optimizer tells it, while the optimizer models the execution engine.

The only way to find the optimal plan for a SQL statement is to execute all the possible plans for a query and choose the cheapest. Hence a complete optimization takes longer than executing the worst possible plan. On the other hand, as we all know, even trivial statements can be executed in many ways, and there are typically considerable performance differences between them. So the term "optimizer" doesn't refer to the result, but to the process of finding the most efficient plan for a given SQL statement.

SQL is a high-level language, completely independent of how the statement should be executed. Before execution it must be turned into steps that describe just how it should be executed to produce the correct result as efficiently as possible. The optimizer chooses these steps and the execution engine carries them out. The leaf nodes of the execution engine's steps are the access methods, e.g., index scan.

The purpose of the optimizer is that given a SQL statement (or specifically, a parsed and pre-processed tree representation of the statement), the optimizer evaluates the different ways of executing the statement and then generates the steps needed to perform them. The execution engine, on the other hand, is a virtual machine implementing relational algebra broken down into operators. These operators are the physical implementation, such as N-ary Nested Loop Join or Hash Vector Aggregate, to the logical operators such as join and sort. The plans are tree-shaped, where each node (or step) is an operator and the leaf nodes are the access methods.

Stefan Karlsson is a technical evangelist with ASE Performance Engineering and can be reached at stefan.karlsson@sybase.com.

```
1> SET SHOWPLAN ON
2> go
1> SELECT title, au_fname, au_lname
2> FROM titles T, titleauthor TA, authors A
3> WHERE T.title_id = TA.title_id
4> AND TA.au_id = A.au_id
5> AND T.title_id = 'BU2075'
6> go

QUERY PLAN FOR STATEMENT 1 (at line 1).

  STEP 1
      The type of query is SELECT.
      FROM TABLE
        titles
        T
      Nested iteration.
      Index : title_id_idx
      Forward scan.
      Positioning by key.
      Keys are:
        title_id  ASC
      Using I/O Size 4 Kbytes for index leaf pages.
      With LRU Buffer Replacement Strategy for index leaf pages.
      Using I/O Size 4 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.

      FROM TABLE
        titleauthor
        TA
      Nested iteration.
      Index : titleidind
      Forward scan.
      Positioning by key.
      Keys are:
        title_id  ASC
      Using I/O Size 4 Kbytes for index leaf pages.
      With LRU Buffer Replacement Strategy for index leaf pages.
      Using I/O Size 4 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.

      FROM TABLE
        authors
        A
      Nested iteration.
      Index : au_id_idx
      Forward scan.
      Positioning by key.
      Keys are:
        au_id  ASC
      Using I/O Size 4 Kbytes for index leaf pages.
      With LRU Buffer Replacement Strategy for index leaf pages.
      Using I/O Size 4 Kbytes for data pages.
      With LRU Buffer Replacement Strategy for data pages.
...
```

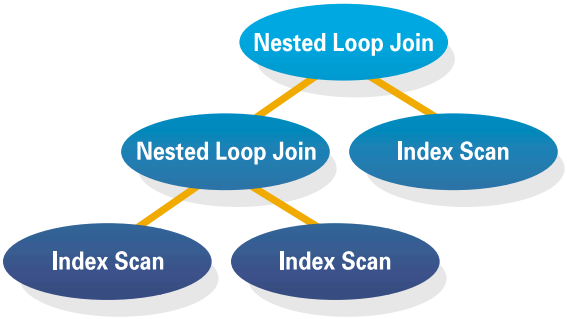

Figure 1 - *Query Tree for Three-Table Join*

Both this code and Figure 1 present the same plan. Omitted in the tree in Figure 1 is the node (or step) that returns the data to the client. The tree shape is called "left-deep," which refers to the emphasis on the left branch.

Now, the optimizer models the execution engine and access methods, as well as the data. The latter is directly applicable to us as users and administrators, since we affect the statistics the optimizer can use. So the optimizer generates all the possible plans for a particular statement and assesses the cost of each plan or permutation. The optimizer, or more specifically the search engine, chooses the most efficient plan with regards to its modeled system resources: logical IO, physical IO and CPU. In the model, the relative costs of these different resources are weighed together, with each dimension given its relative weight. With the new features of the 15.0 execution engine, the importance of PIO and CPU increases, such as the cost of a hash join, must include the building and probing of the hash table.

But the model needs to be translated into absolute metrics! This is particularly obvious when taking a parallel plan into account. Here a higher resource use is exchanged for a decreased execution time, so that while the relative costs between the different resources are key, the model needs to incorporate time. One could say that there's a specific optimization goal in effect when parallelism is enabled: to optimize for response time over minimizing resource use. As we shall see later in this article, there are settable configuration goals for the optimizer in ASE 15.

New Features in ASE 15 Optimizer and Execution Engine

Hash-Based Techniques

Hashing is an indexing technique, a way of translating a logical key to its physical location. In fact, it has been used as an alternative to B-tree indexes to maintain uniqueness in

relational database management systems. In ASE, hashing has been employed internally to map lock object to lock, locating data pages in caches, etc.

Hashing works by applying a function on the logical key, and the result is the physical location for the object identified by that logical key. The physical location is called a hash bucket. An example of how this would work:

Assume that we have an integer identifier and that we have chosen a fixed 20 hash buckets, e.g., entries in an array pointing to memory locations. As a hash function we use *modulo*, or, more specifically, (mod 20). An object with identifier value 125 would be stored in Bucket 5 (125 % 20 = 5) while a value 338 would correspond to Bucket 18. The hash value is really the offset into the hash table.
This example demonstrates two inherent properties of hashing: First, that the same identifier will always be hashed to the same bucket, and second, that many identifiers may be hashed to the same bucket. The first property can be used to manage uniqueness and/or remove duplicates, as the value 125 always will be hashed to the same bucket. The second property has implications for choosing the hash function in that the fewer the objects hashed to the same bucket, the higher the performance.

Applications of hash-based techniques will be discussed in more details later in this article, but it's worth listing them here:

- **Joins:** A hash-table is built from a result set and then used to probe another result set.
- **Group by:** Hash-tables are built on the projection used for grouping
- **Distinct:** If a value is already in the hash-table, it can be discarded.
- **Union:** Similar to distinct.

There are design decisions to make when implementing hash-based mechanisms. The cost of hashing is not only the processing, i.e., hashing, but the investment in memory for the hash-table. Early implementations relied on the hash-table fitting in memory; if not, performance would degrade dramatically. ASE query processing implements a hybrid approach, which strives to keep the hash tables in memory but behaves gracefully and maintains high performance if hash table partitions must be swapped out to disk.

Join Methods

ASE 15.0 supports the two join methods from pre-15.0 versions, Nested Loop Join (NLJ) and Merge Join (MJ). While simple, NLJ is very fast and resource-effective, especially when the outer result set is small with relatively few rows. A key point when the optimizer is costing NLJ is to minimize the number of rows in the outer table(s).

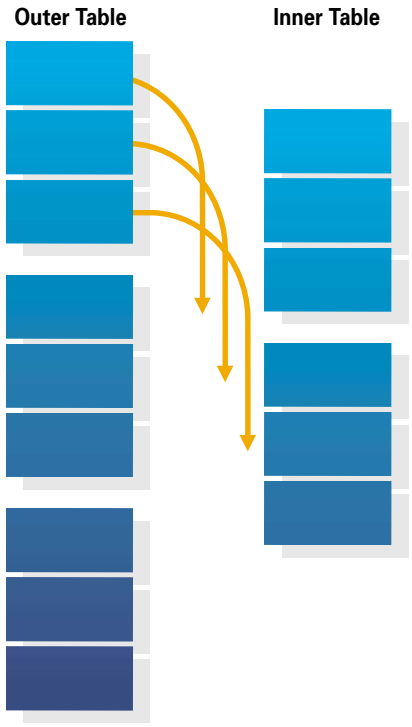

Figure 2 - *A note on the graphics: All illustrations of joins show only data pages and data rows. Obviously any access may be an index scan.*

The Execution Engine always translates an NLJ into an N-ary Nested Loop Join (N-ary NLJ), a technique patented by Sybase. An NLJ has two child nodes, while an N-ary NLJ can have an arbitrary number of child nodes (hence the "N-ary") and performs as well or better than an NLJ. The performance gain comes with joins between three or more tables, and conditions between the outer and an inner table besides the one being joined with the outer. In other words, the N-ary NLJ shortcuts the scan, saving IOs on the inner tables.

In ASE 12, NLJ is complemented with merge join. The big difference between NLJ and MJ is that an MJ always has a scan count of one for both sides of the join, while NLJ implies that the inner result set is scanned once per row in the outer. The key differentiator means that MJ works well also when the outer result set has many rows.

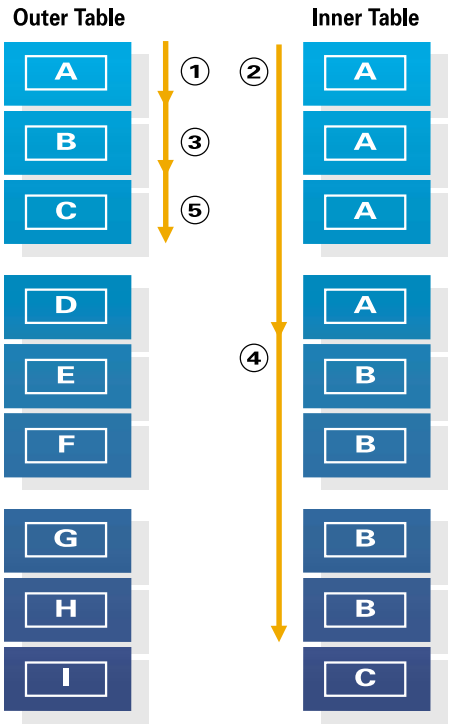

Figure 3

A merge join is really a merge of two tables, where the tables are scanned in alternate steps. For efficiency, the positioning in the inner table is done by applying all possible search arguments (SARG). Caching of inner table rows means that duplicates in the outer table don't cost any more logical IO.

Merge join is often called "Sort-merge join," implying that a sort is required. This is not the case, but there's a point in that term. Merge join requires that both sides are sorted on the join columns, typically through indexes on the join columns; if not, a sort is needed to produce the proper ordering.

New in ASE 15 is *hash-based join (HJ)*. In this case, a hash-table is built from one side, then the other side is scanned and every row is probed again against the hash-table (see Figure 4).

The hash value "column" is really the offset from the hash- table's start, so it's not stored in the hash-table. The real content of the hash-table is the projection from qualified rows from the outer table.

As with the merge join, the tables are scanned once and only once, so a hash join works really well when the result sets are large. AS opposed to the MJ, there is no requirement on ordering, which, in turn, makes indexing easier and HJ more versatile. The overhead is the cost of building the hash-table and the investment in memory it poses.

Besides Nested Loop Join and N-ary Nested Loop Join, Merge Join and Hash Join, there are two adjacent techniques ASE 15 employs to speed joins in certain cases. The first is the well-known reformatting, building a missing index for a query that is dropped after the query finishes. This is a feature that ASE has had as a safety net from the start. It should be noted that ASE 15 QP has an increased ability to assess and use reformatting.

The other technique is the RID-join (Row ID join), which is new for ASE 15.0. RID-join is not a join method, nor is the technique used only for joins. However, it should be introduced here because of the implications it has for joins.

Basically, RID-joins decouple the index scan from the accesses to data pages, so a node above the scan node gets the RIDs instead of the full data. This allows the execution engine to sort the RID before accessing the data, improving performance in several ways. One benefit is that index inter-

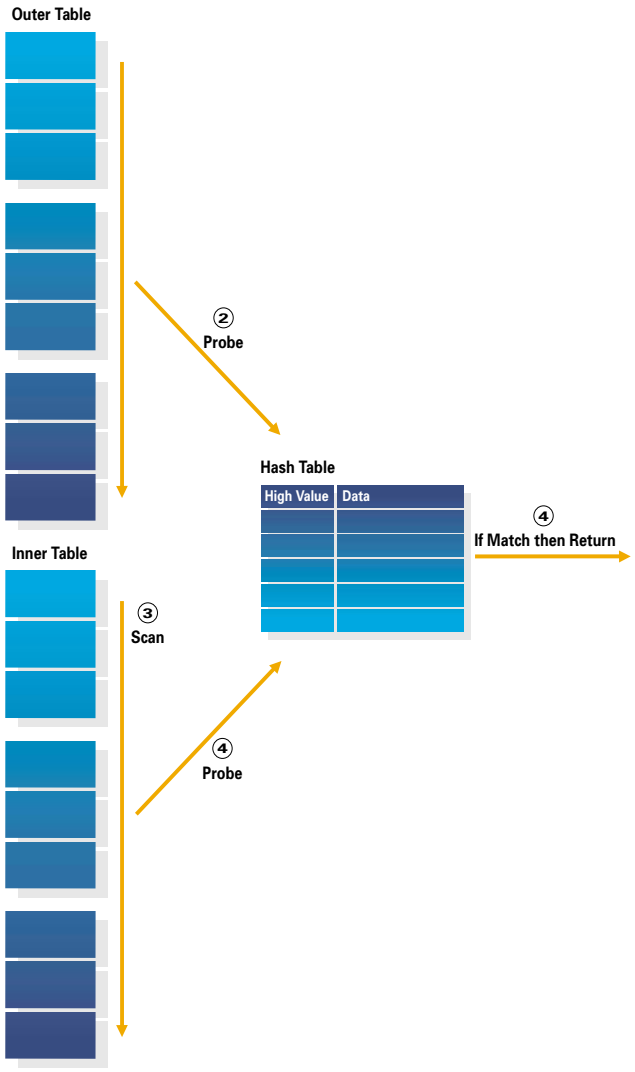

Figure 4

section (using more than one index to access a table) can be further optimized, since the RIDs can be sorted to remove duplicates and then efficiently accessed. Another benefit is that RIDs can be sorted to increase the cache hits in ASE and fully utilize the SAN disk controller's pre-fetch mechanisms.

Computer Columns and Functional Indexes

When there are functions or expressions in the **where** clause, query tuners immediately realize that no index will be useful for that condition. Not so anymore, since with the 15 execution engine comes the ability to create indexes on expressions or even to include them in a table definition.

As an introductory example, let's look at a case insensitive search in an ASE that uses a case sensitive sort order. By defining an index on the function of **upper(last_name)**, a **select** statement using that function can use the index to execute the query.

```
CREATE TABLE customer (
        last_name VARCHAR(50)           NOT NULL
        , ...
)
CREATE INDEX u_lname_idx ON customer( UPPER( last_name ) )

SELECT ... FROM customer WHERE UPPER( last_name ) = 'SMITH'
```

Another use is to include computed columns in the table. If needed it can be indexed:

```
CREATE TABLE orderline (
        ...
        , price       NUMERIC( 10, 2 ) NOT NULL
        , quantity    INTEGER   NOT NULL
        , tot_sum     NUMERIC( 10, 2)       COMPUTE price * quantity
        , ...
)
```

In this case, the table **orderline** has a column **tot_sum** which is not physically stored. It is calculated as the column is referenced. By adding the keyword MATERIALIZED, the column is physically stored on disk. Among the possible applications for this feature are the ability to index attributes in columns defined as Java ADTs and to index functions and attributes in XML documents. This list of application areas is quite long.

Streaming Operations

The execution engine and optimizer for ASE 15 maintains properties for all nodes. An index scan always returns data in a sorted order, and the optimizer uses this and other physical properties. In certain cases, this presents a significant performance boost. Operations such as **merge join** and **order by** require sorted data. Another function is **merge union**, when the queries return the same sort order and the result is kept streaming back to the client. In other words, when costing a plan, the sort of each node is taken into account. A plan that has the data sorted according to the **order by** clause will be cheaper than a plan that requires a sort.

ASE 15 will use sorts to a lesser extent than previously. Furthermore, given how ASE now can do **distincts**, **unions**, and **group bys**, there's a decreased use of worktables and improved performance. Consider a statement that uses a distinct. Instead of holding the rows until all are sorted, the result set can be streamed back after each row is checked against a hash-table. If the row has a match hashed, it is discarded. Otherwise it is returned and entered into the hash-table.

Optimization Goals

In general, the goal of the optimization phase is to minimize resource use, and the resources modeled are CPU and logical and physical IO. The goal, in turn, translates into the best throughput and response times.

As mentioned above, for parallel plans the resource use must be considered in absolute terms, translated in time. There is also another situation where this general assumption does not hold: In applications where users or processes can start working immediately when the first row is returned, and potentially not even need all the rows. Cursor-based applications and some web applications are common examples. The optimization goal **fastfirstrow** meets this requirement. The principal behind **fastfirstrow** is to stream data back to the application as quickly as possible, by using only streaming operations, and completely avoid any operation that would dam the rows, such as a sort.

The default optimization goal is **allrows**, which is the standard purpose optimization goal and uses all available strategies. As the name suggests, the optimization goal **allrows_oltp** is particularly aimed at OLTP-type application. To do so, the optimization goal specializes in particular tree shapes and uses only the nested loop join. Also, it generates only serial plans, not parallel plans. Besides parallelism, this optimization goal is the one that is closest to pre-15 query processing.

New Plans and Trees in the Search Engine

Even for a fairly simple statement, there are usually many different execution plans, all producing the same result set. What differentiates them is the performance characteristics and use of system resources. Generating these plans involves general algorithms as well as logic to identify special cases.

The more general aspect of generating plans is to have plans that include various indexes on each participating table and all possible join orders. An example of a special case is the star join, where counter-intuitively a Cartesian product between the dimension tables before joining with the large fact table can prove the most efficient.

ASE 15 considers many different tree shapes, including what's called "bushy trees." Here's a four-table join as a left-deep tree and as a bushy tree, both from the ASE 15 optimizer:

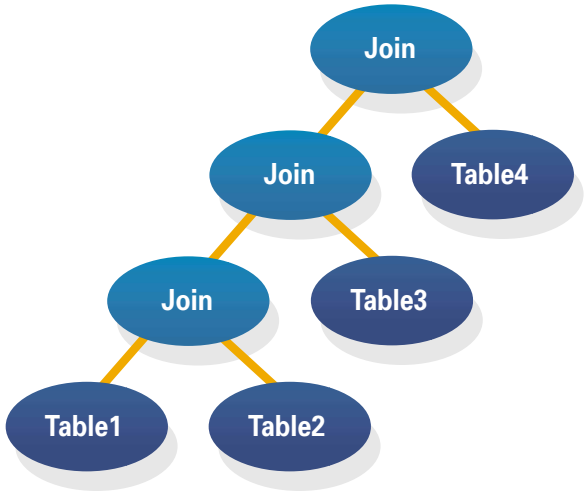

Figure 5 - *Left-deep Tree*

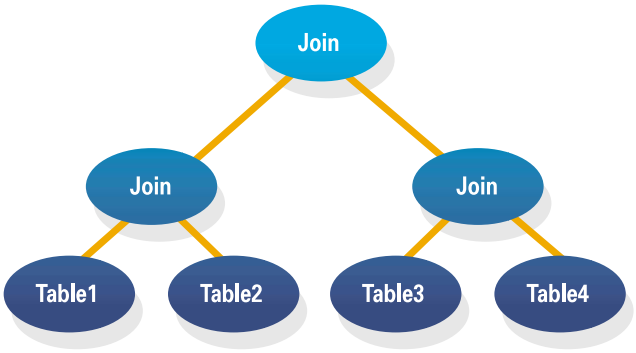

Figure 6 - *Bushy Tree*

Once each plan is generated, it's a matter of assessing the individual costs. Each plan's cost is computed bottom-up, starting with the individual scans (or leaf nodes), moving on to include the second, third table, etc., so that each partial plan's cost is based on available statistics. This means that the total cost is based on all statistics available for all potentially participating columns, indexes, and tables. It also means that unpromising sub-plans can be discarded, and, as a consequence, all tree permutations that build on these sub-plans.

The last optimization is one of many built into the ASE 15 optimizer to intelligently manage the search space. Since there may be a very large number of different plans, an exhaustive search can consume considerable time. ASE 15 has several features to efficiently manage the search space.

Lifted Limitations

A requirement of ASE 15.0 and the new query processing components is to remove or increase limitations. This goes for very specific and visible limitations, like the number of literals in an IN-predicate to aspects like the tree shapes we saw above. Some removed/increased limits are:

- Number of columns in a GROUP BY clause removed (limited by max row width of approx. 16kb).
- Total number of tables in UNION query increased to 512.
- Number of worktables per query increased to 46.

The streaming operations discussed above also decreases the need for worktables.

ASE 15 also introduces unsigned integer data types, as well as 64-bit integer data types. This is convenient for data modeling as well as application development. The unsigned "bigint" data type holds integers between 0 and about 184 billion.

Another feature designers and developers are likely to appreciate is the large identifier support. An identifier can now be 255 characters and this includes names for tables, columns, indexes, views, triggers, defaults, rules, constraints, procedures, local variables, jars, databases, devices, segments, caches, functions and others. We also now have full mixed data type support. ASE 15 fully supports SARGs and joins where the data types are implicitly convertible, regardless of data type precedence.

OLTP Performance

Another key requirement for the ASE 15 query processing components is to ensure OLTP performance while thoroughly exploring more alternative plans and methods than have previously been available to the optimizer. New and smarter algorithms maximize the optimizer's work. One algorithm is that as the costs of sub-plans are computed as too expensive, the search engine no longer considers plans with this combination. Another is that as a tree is generated with the Tables 1 through 4 join order, the tree is not re-generated for the 4-1 join order, but the mirror of that tree has its cost computed instead.

Users can control the optimizer's work in several ways, such as timeouts and optimization goals. But how do timeouts really work? It's one thing to stop executing, but an entirely

other matter to determine how usable that plan really is.

Besides the main search engine, a secondary search engine takes a greedy approach and only considers certain types of plans. The greedy search engine very quickly arrives at a good plan and runs before the main search engine. In other words, while the full search engine is working, two optimizations are employed. The full search engine provides a good plan to compare with, and as soon as the cost of a sub-plan exceeds that of the good plan from the greedy engine, it can be ignored.

Certainly using timeouts can be a double-edged sword, to say the least. But since the full search engine already has a good plan starting off, timing out is now a viable option.

Optimization goals also control the optimizer by providing the priority for the optimization, and the optimization goal **allrows_oltp** is specifically aimed at OLTP-type applications.

DSS/Reporting Performance

Decision-support type queries, be they operational DSS, warehousing applications, or the ever-present reporting batch jobs, have similar performance requirements but different workload characteristics. For complex schemas and larger amounts of data, the typical complex queries scan large amounts of data over many tables, usually to aggregate key data points.

Based on these requirements, ASE 15 employs the latest techniques to quickly and efficiently execute queries, such as scanning only once, maintaining cache hit rates, and taking advantage of properties including the data's sort orders. The optimizer has very few restrictions in the type of plans it considers. More plans are evaluated and in more detail while keeping optimizations in place to go through the search space efficiently.

The newer types of plans (or trees) have no inherent restrictions on how parallelism can be employed. Some table accesses and joins may be largely parallel while other joins are serial, so that the resulting parallel join can be joined to the result from a serial join and vice versa. Parallel access can be further optimized through the semantic partitioning feature discussed in the first article in this series.

A key property of the new join methods, be that the new hash join or the merge join, is that a table is scanned only once. When a query scan returns a large amount of rows in an outer table, the inner tables in a join will nevertheless be scanned only once—a huge gain for DSS-type queries.

With the larger result sets from each scan—table or index—the new join methods are a performance boost, as is the approach to grouping. Materializing a result set in a worktable does not scale gracefully as the result set grows in size. This is inherent in this algorithm. Hash-based grouping, or hash vector aggregates, does scale, and without the non-linear growth in resource consumption. The same performance and resource consumption gains goes for **distincts** and **unions**, both also benefiting from hash-based techniques.

Not having worktables for groups leads us to sorts. While the hash vector aggregates don't require sorted data, it's very common for DSS-type queries to require sorted results through the **order by** clause. Here the optimizer cleverly chooses between maintaining the sort order returned from the index scans and using another index which would require a sort.

Besides data growth, schemas have continued to evolve. More details are included in existing schemas; more attributes are modeled into them. Subject areas are now larger as business and IT support for business grows. Third, new modeling techniques have been introduced such as more object-oriented modeling of database schemas and specialized data models such as Star schema and their cousins, Snowflake and Snowstorm.

Taking the Star schemas as a good example, it is up to the query processing components to employ specific strategies for best performance. Here it's a matter of applying as many conditions as possible before accessing the very large fact table. A second strategy used by ASE 15 is to provide efficient access to the fact table by either using composite index or index intersection. The latter means using more than one index for a single table and, in a manner very close to the existing OR strategy, use the **union** of the RIDs identified through the different indexes when accessing the actual data pages.

Conclusion

The query processing technology in ASE 15 is the next generation response to emerging requirements for mixed workload and DSS applications on the operational data store. It addresses needs without compromising OLTP application performance through innovative use of a variation of techniques and features. The new plans and search engine capabilities are coupled with refreshes of existing technology, such as the nested loop join and merge join, and new techniques, including streaming and hash based operations.

Apart from its performance features, ASE decreases the need for query tuning, and at the same time presents a very productive toolset for tuning activities. The topic for the next article is a deeper look into these features and the query tuning toolset. ASE 15 is currently in beta testing and is planned for release later this year. ■