Paper # 3018

Allbase by Design

Jeff Townsend
ASI Market Research, Inc.
101 N. Brand Blvd., suite 1700
Glendale, CA 91203
(818) 637-5600

Introduction

Database performance will be an important and often crucial goal for client-server and server-driven OLTP and OLAP development. The logical and physical design of an HP Allbase/SQL database have a profound impact on data throughput. With the vast amount of technical documentation on relational databases, it is difficult to separate the nonessential from the truly important, performance-critical information. Design, creation and administration tasks in Allbase/SQL that especially impact resources and have performance implications will be identified. Recommendations will be made on DBA tasks and configuration preferences.

[A great source for an introduction to relational database terms and basic concepts is HP's mini-guide: ✌Up and Running with Allbase/SQL.✍ Practicing with the provided sample DBEnvironment, PartsDBE, will be beneficial.]

DESIGN

Normalization

A careful and thorough list of entities and their related attributes will help you arrange the data into, respectively, tables and columns. Relational databases perform best when the data is normalized, as this process reduces data redundancy and facilitates efficient retrievals and updates. The goal is to get to third normal form (3NF), or even Boyce-Codd normal form (BCNF). BCNF can also be stated as: ✌Every Attribute must be determined by the Key, the whole Key, and nothing but the Key, so help me Codd!✍

There are drawbacks, however, to overnormalization such as more joins and additional referential integrity indexes. You may consider denormalizing two columns

always reported together or fast access columns. If two rather static tables are constantly being joined, consider combining tables or duplicating some columns. Not all joins are expensive and there is a cost to the alternative denormalization. Consider the effect of denormalizing on other processes and tune to the most used or most important process. Follow another computing adage,

✌*Normalize 'till it hurts, then optimize 'till it works!*✍

Partitioning the data is another trick used to improve performance. Horizontal partitioning divides a large table containing historical data into two: a current data table and an archive table with a "shadow" of its parent table's column structure. Vertical partitioning removes less frequently accessed columns to their own table. Be sure the primary key values are copied so a 1:1 join can be performed if necessary. This trick will reduce the size of the parent table's rows making it more efficient to process.

Data Selection

Understanding what the data is, where it comes from and how it will be used is crucial. Table design should be modified to accommodate planned data selection, multi-table joins and views. Anticipating how data will be selected and used helps in defining relationships between tables, keys and attribute columns. Consider separate views for restricted access to one or more tables. Views have the added benefit of presenting a consistent and simplified representation of a related set of data.

Constraints

Ensure integrity by designing unique or referential constraints on columns. Duplicate key values are eliminated via the unique constraint. Referential integrity assures a dependency relationship where a key value exists before a row with that key value is inserted in a referencing table.

Other Issues

Other design issues, such as application characteristics (on-line vs. batch, static vs. volatile data, etc.), should be well thought out prior to implementing your logical design into a physical one.

Index Types

The most common cause of bad database performance is misused and poorly designed indexes. A relational index can be defined on one or more columns in a

table. Its primary objective is to speed data retrievals, but with the unfortunate side effect of slowing modifications due to the overhead of index maintenance for data inserts, updates and deletes. Understand also that all queries against Allbase/SQL pass through its Query Optimizer, whose task is finding the optimal access path. You cannot explicitly request the optimizer to use a particular index. You can, however, implicitly improve the chances of index use by ensuring an index exists for queries where certain values for table columns are to be selected. Design indexes according to your application's most executed and most important processes.

Allbase/SQL currently provides for four types of indexes: unique, clustered, normal and hashed. A unique-and-clustering combined index type can be defined as well. The first three types store values in a B-Tree (or B+ -Tree actually, as the leaf pages are doubly linked). A hash structure, to be precise, is not a physical object like the other index types, but a way of arranging data in the DBEFile pages.

A unique index, as the name implies, ensures no duplicate key values exist. A clustering index attempts to place data rows with similar key values physically close together on the same or consecutive pages. Performance degrades on a clustering index as these pages become full, when new rows must be inserted wherever a page with room can be found. Utilize a clustering index only where an application needs to search large volumes of data to retrieve rows in sequential key value order, and where the table is either static or the number of insertions are nearly equal to the deletions on the table. A normal index is neither unique nor clustered, and used solely for its B-Tree retrieval performance. Hashing is best used with uniformly distributed, unique key values and direct per-value access ("equal" in the query predicates, or "where" clauses) is the overriding concern. Hashed tables are preferred to B-Trees if rows are frequently inserted and/or deleted and key values themselves are not changed during updates, but they require manual maintenance by the DBA to continually optimize its performance.

Indexes do not have to be permanent objects, but can be created when needed then dropped. This provides improved performance for the duration of the index'es existence without the continual index maintenance overhead. This strategy is particularly suited to monthly and other reporting or archiving cycles.

CREATION

A thorough and well thought out logical design must now translate into an equally solid physical design and implementation. Allbase/SQL performance is also closely tied to DBEnvironment configuration and object creation options.

DBE

A DBEnvironment is newly created with the "start dbe new" command. You supply startup configuration parameters with this statement. These parameters set environment operating limits such as: single or multi user mode, autostart mode, archive mode, user timeouts, language, DDL enabling, DBEFile0, log file(s), maximum transactions, data and log buffers, and runtime control block pages. Most parameter values, though important, have little impact on performance. Shared memory buffers, however, have a large effect on I/O performance.

For every multi-user DBE session started, a block of shared memory is reserved. All users and programs accessing the DBEnvironment share this allocated memory, which is composed of three types of buffers: runtime control blocks, data buffers, and log buffers. Control blocks are needed for such DBCore services as database access control, locking and buffer pages. Data buffers hold data pages currently in use. Log buffers hold changes made to data pages during a transaction until they are written to log files on disc.

Lock management is the greatest user of runtime control block buffer pages. Consequently, depleting this resource is less likely with shorter transactions, coarser locking strategy (table *vs*. page *vs*. row), or more efficient concurrency practices. Formulating the number of runtime control block buffer pages is based on several site- and situation-specific factors. An HP manual states:

> ✌*The number and type needed at any one time depends on such factors as the number, duration, and type of concurrent transactions, the amount of page level locking, and the amount of update activity occurring.*✍
>
> - Allbase/SQL Database Administration Guide

I recommend trying an initial setting between the default 37 and, say, 100 pages. The first time you receive a "shared memory lock allocation failed" (DBERR 4008), pass on HP's solution of re-executing the transaction with a 20% buffer size increase and, instead, minimally double your current value. The maximum setting allowed is 2000 pages.

For the data buffer, composed of 4096-byte pages, set the parameter to about 12 times the number of active transactions or the default (100) whichever is greater. With too few pages, there will be excessive page swapping. With too many, performance is decreased as all data buffer pages are searched to see if the needed page is already in memory. For a more quantitative approach, examine the tables accessed in performance-crucial transactions. The right number of buffer pages is approximately the number needed to hold the small tables and the aged, least used pages of the large

tables. An excellent discussion, with examples, can be found in the ✌DBE Guidelines on System Administration✍ section of the Allbase/SQL Performance Guidelines manual.

Provide for enough 512-byte log buffer pages (independent of data buffers) to give all active transactions enough log space. Additional buffers will not help performance, but neither will it degrade it noticeably. The maximum is 1024 pages.

Do not exceed the amount of shared memory available on your system as you adjust the shared memory (buffer) parameters. Excessive shared memory may cause page faults. A formula to approximate total shared memory used (SM, in Kbytes) is:

SM = (4.2 * Data Buffer Pages)
+ (0.5 * Log Buffer Pages)
+ (0.16 * Number of Concurrent Transactions)
+ (4.1 * Control Block Pages)

Increasing data buffer pages and control block pages increases total shared memory much more than log buffer pages or number of concurrent transactions. Total shared memory should not exceed free real memory available on the system at any given time, as you never get a performance benefit by defining more page space than will stay in real memory. Too many data buffers may force paging of virtual space and will degrade performance.

Database Objects

Physical design is primarily implemented by arranging data in tables, which reside in DBEFiles, and putting DBEFiles in DBEFileSets. DBEFileSets are also the logical home of tables, which are always created in a particular DBEFileSet.

Avoid putting any user-defined tables in the System DBEFileSet, as it is accessed frequently and its tables locked differently than in other DBEFileSets. Place tables larger than 1000 pages or in their own DBEFileSets. This makes sequential scans and maintenance, such as "update statistics", perform faster. Group smaller tables together into DBEFiles of 1000 pages or less, and place in their own DBEFileSet. This reduces the number of physical files the system has to find and open. It utilizes discspace efficiently and minimizes real I/O.

DBEFiles are the physical files used to store table and index data. Create them in multiples of 253 pages (1 page table page followed by 252 data pages). Use dynamic DBEFile expansion, setting initial, maximum, and increment page sizes to avoid creating unused extra space yet also avoid running out of DBEFile space at run time. Create separate "table" and "index" DBEFiles, especially for large tables, and consider placing on separate discs to improve I/O performance. Create a "mixed"

DBEFile for a small table, especially with a single index, where data and index rows will be on alternating pages resulting in faster processing and better space utilization.

In creating the database object ✌table,✍ not to be confused with the ✌table✍ DBEFile, the designer implicitly defines the access and locking mode. This has a tremendous effect on performance! The four table types, in order of increasing consistency and decreasing concurrency, are: publicrow, public, publicread and private. The first two are for maximum concurrency, holding many locks of finer granularity but for shorter periods. "publicread" can improve performance by using fewer table locks for writes, but locking the table means it can only be modified by one transaction at a time. Tables that have a lot of read activity and very little updating, except in batch at off-hours, are ideally suited to this type. "private," the default (!), holds exclusive table locks for read and write and should only be used for special tables accessed by only one user at a time.

A table must have at least one named column and its datatype. Avoid nulls which may cost an additional 5% overhead. Also avoid variable length datatypes that may waste more space than they save, and may change the row's position or data page when the column is updated later. If null or varchar/varbinary columns are unavoidable, place them as the trailing columns in the table schema to minimize the row shifting. Integer values may be 5-15% more efficient than packed decimal. Since SQLCore only performs 4-byte arithmetic, all smallint values are automatically converted to integer. For better performance, use datatypes appropriate to the programming language and application, and avoid data conversions. If space savings is paramount, smallints only require half the storage space as integers.

Indexes and Constraints

Indexes should always use "not null" columns as keys. If a column is constantly used in a "where" clause, it should have an index. Contrary to some HP documentation which states ✌*indexes are always created explicitly*,✍ certain DDL (data definition language) commands implicitly cause indexes to be created. Buried deep in the documentation is a non-trivial piece of information on implicitly created indexes:

> ✌*If you are using unique or primary key constraints, Allbase/SQL automatically creates unique indexes which can be used for data retrieval. If you are using referential constraints, Allbase/SQL creates a PCR (parent-child relationship), which is an index on the two tables in the referential relationship.*✍
>
> - Allbase/SQL Performance Guidelines

If you have already applied a unique or primary key constraint on a column, do not execute a "create unique index" command or you will likely double the resource and

performance hit of duplicate indexes. The Query Optimizer already has the option to use for retrievals the index implicitly created by the constraint.

Integer and char index columns are preferred to decimal or date, and keep the length to 20 bytes or less. Create a compound index if the "where" clause contains "=" (equal) comparators along with "and"s. Place the most used column, and outer sort sequence, in the first position. Create separate indexes on columns from a single table used extensively with the "or" predicate. Do not bother creating indexes on columns with very few values (e.g. yes/no, year), especially when retrieval is expected to return 30% or more of the rows. On a side note, I strongly suggest naming each constraint.

Check constraints are essentially search conditions placed on a particular column to validate values as they are inserted. The performance is marginally better compared with doing the check programmatically. Check constraints and default column values benefit a system by standardizing actions that cannot be circumvented by any application using the database.

Views, Rules, Procedures, Sections

A view is essentially a "select" command stored in the system catalog and not a physical copy of the data. Allbase/SQL performs the data retrieval when the view is used. It is a strategy to present a consistent view of selected data to authorized users. The column names could be simplified and the selection criteria made transparent for a more intuitive and understandable display of information to the user. Views, *per se*, have little or no impact on performance compared with similar programmatic selection of the data.

Rules and procedures provide a mechanism for the database itself to enforce data relationships and data integrity, but far more flexible than the simple integrity constraint. A procedure is created, then called by a rule triggered under certain conditions. The "where" condition is checked before the procedure is loaded, so triggering on or after the "where" clause can avoid overhead if the clause is false. More control and better performance is gained by developing several rules with separate conditions and procedures rather than a single rule with no condition and one big procedure that makes decision checks before executing certain steps.

Procedures as well as preprocessed, SQL-embedded programs are stored as sections. A number of actions against the database will invalidate some or all sections, such as "update statistics." To avoid the overhead incurred with automatic revalidation when the first transaction attempts to execute an invalidated section, the DBE should use the "validate" command. Preferably, use both of these statements during periods of low activity.

ADMINISTRATION

Sqlutil

Sqlutil is a DBA tool to look at and change DBECon parameters. It also performs many tasks on log files, and is used to back up and restore DBEnvironments. DBEFiles may be purged with this utility, but be sure to drop them from the DBEnvironment in ISQL first. One particular performance task it can perform is moving DBEFiles and log files among your disc drives. This is known as "load balancing." HP recommends improving performance by using "movefile" to place table and index DBEFiles for the same table on different discs. Also recommended is putting log files on a separate disc drive than the DBECon file.

Monitoring

The database administrator (DBA) should keep close tabs on table capacities and index efficiency using SQLMON, a diagnostic utility, or by accessing and reporting from the system tables. This information should be examined only after an "update statistics" has been performed on the tables of interest. DBEFile usage and percent of capacity can be found in SQLMON's Static DBEFile screen. The following SQL statement will report essentially the same information:

```
select dbefname, dbefupages, dbefnpages,
  dbefupages*100/dbefnpages from system.dbefile;
```

Indexes on tables that have undergone multiple updates and deletes may use unnecessary space by creating sparsely populated pages. From a performance standpoint, maintaining efficient, compact indexes is very important. The cluster count of an index indicates how many times a different page must be accessed to retrieve the next row during an indexed retrieval. The greater the cluster count, the greater the I/O overhead. Cluster counts (ccount's) for indexes are reported in SQLMON's Static Cluster screen. To report index ccount's directly from the system tables, use the SQL statement:

```
select tx.indexname, tx.tablename, tx.ccount, tt.nrows, tt.npages
  from system.index tx, system.table tt
  where tx.owner = tt.owner and tx.tablename = tt.name;
```

and for constraints:

```
select tc.constraintname, tc.tablename, tc.ccount, tt.nrows, tt.npages
  from system.constraintindex tc, system.table tt
  where tc.owner = tt.owner and tc.tablename = tt.name;
```

When the cluster count about equals the number of pages used, then the table is optimally clustered for that index. If the cluster count nearly equals the number of rows, it indicates little or no clustering and the index is quite useless. When the cluster count exceeds half the value of nrows, the DBA should consider dropping and re-creating the index. The method should make use of an "unload internal" command using a "select" with an "order by" on the index key, then a "load internal." For very large tables, it may be 300 to 500% faster to reload programmatically utilizing the "bulk insert" command.

CONCLUSIONS

While there is a vast amount of Allbase and SQL information found in various publications, the crucial performance-related aspects are not clearly differentiated by degree of impact. The one HP manual that comes closest is ✌Allbase/SQL Performance and Monitoring Guidelines,✍ but this is better suited as a reference than as a guide. Certain design, creation and administration tasks in Allbase/SQL have a much greater impact on resources and performance than others.

Allbase/SQL is a powerful relational database that approaches industry-leading SQL performance provided the designer starts with a solid, normalized relational model. This must be followed by creating appropriate database objects with optimal parameters. The physical design must stem from a thorough understanding of the application and utilization of data by users and processes of the database. Lastly, as the database is changed dynamically by its use, the DBA should monitor, correct and optimize inefficiencies as they appear.