

Data warehouse Advance Tuning

Author: Kevin Windt

Organization: Evaltech, Inc.

**Evaltech Research Group,
Data Warehousing Practice.**

Date: 04/08/04

Email: erg@evaltech.com



Abstract:

Tuning a data warehouse is more difficult than tuning an OLTP system because of the ad hoc and unpredictable nature of the load. A data warehouse, of its very nature, evolves over time. As a result, many aspects of the warehouse environment changes as the profile and the usage of the data change.

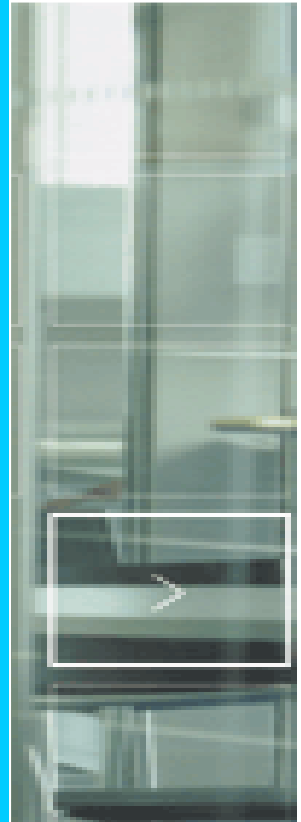
Change in user profile and data access pattern clearly mean changes in the queries that are being run. These changes can also have knock-on effects on the aggregations that need to be generated or maintained, perhaps even affecting what data needs to be loaded, and how long data needs to be maintained.

It is all too possible in a data warehouse environment for a single query to monopolize all of a system's resources, particularly if it is running with high degree of parallelism. Often the correct tuning choice for such eventualities will be to allow an infrequently used index or aggregation to exist to catch just that sort of queries.

These practices are put in place for tuning the: Tuning Queries, data load, Operating System, Database Physical Structures, and Network Traffic.

Intellectual Property / Copyright Material

All text and graphics found in this article are the property of the Evaltech, Inc. and cannot be used or duplicated without the express written permission of the corporation through the Office of Evaltech, Inc.



Data Warehouse Advanced Tuning

INTRODUCTION

Tuning a data warehouse is more difficult than tuning an OLTP system because of the ad hoc and unpredictable nature of the load.

A data warehouse, of its very nature, evolves over time.. As a result, many aspects of the warehouse environment changes as the profile and the usage of the data change.

Change in user profile and data access pattern clearly mean changes in the queries that are being run. These changes can also have knock-on effects on the aggregations that need to be generated or maintained, perhaps even affecting what data needs to be loaded, and how long data needs to be maintained.

It is all too possible in a data warehouse environment for a single query to monopolize all of a system's resources, particularly if it is running with high degree of parallelism. Often the correct tuning choice for such eventualities will be to allow an infrequently used index or aggregation to exist to catch just that sort of queries.

These practices are put in place for tuning the: Tuning Queries, data load, Operating System, Database Physical Structures, and Network Traffic

TUNNING QUERIES

The data warehouse contains two types of query

FIXED/STATIC QUERIES

Tuning the fixed queries is no different than the traditional tuning of a relational database. Not a lot can be done to the fact data to improve performance. However, when dealing with dimension data or the aggregation, the usual collection of sql tweaking, storage mechanisms and access methods can be used to tune these queries. New aggregations can be created, or extra index added to the dimension data, b-tree indexes or bit-mapped indexes if the database supports them. Different storage mechanism, such as clustered tables, hash clusters or index-only tables, could be used. These tuning must not affect the performance of ad hoc queries.

AD HOC QUERIES

The number of users of the data warehouse will have a profound effect on the performance, in particular if they are the ad hoc users. For each users or group of users we need to know the following:

- 1) the number of users in the group
- 2) whether they use ad hoc queries frequently
- 3) average size of query they tend to run
- 4) maximum size of query they tend to run
- 5) peak time of daily usage
- 6) queries run per peak hour
- 7) whether they require drill-down access to the base data.

The more unpredictable the load, the larger the queries, or the greater the number of users the bigger the tuning task.

A typical data warehouse will have some where 10 to 50 users, although the first signs of data warehouses with large user populations have started to appear.

The more ad hoc the nature of the query mix, the more difficult the job of tuning the data warehouse. The trick with tuning this sort of environment is to move the query mix from ad hoc to predictable. To do this, it is necessary to identify any similar ad hoc queries that are frequently run. If such queries exist, the database can be changed to make their running more efficient. New indexes can be added, or new aggregations created and maintained specifically for those queries.

The way to turn an unpredictable query load into a predictable load is to control the queries via batch queues.

No users should be able to submit a query directly, and all queries should be controlled via the query manager. It is equally important

that no query can monopolize the system. Such queries should be detected and killed. If there is a mix of small and large queries to be run, then allowance will need to be made for the smaller queries. Multiple queues can be used, with a separate queue for smaller queries. If a query submitted to the small queue exceeds its quotas it should be killed and submitted to the large queue.

TUNING THE DATA LOAD

Data load is the entry point into the system; it provides the first opportunity to improve performance. Database loading software can work in a number of different ways. The more normal way is for the data to be inserted into the database using sql layer. This approach means that all the checks and constraints of a normal insert have to be performed. These checks take time to perform and they are costly in CPU. The purpose of these checks is to try to make maximal use of the space, packing the data as tightly as possible. Some data load programs allow many of these checks to be bypassed, and the data to be placed directly into preformatted blocks. This can lead to some space wastage which is quite small, but the time gain can be quite significant.

Another costly part of any data load is the maintenance of any indexes on that data. The options are either to maintain the indexes as the data is being loaded, or to drop the indexes and re-create them when the load is complete. Experience shows that the latter option is likely to be quicker, particularly if the RDBMS supports the re-creation of the indexes in parallel.

It is also worth noting that even if the approach of dropping the indexes, loading the data, and re-creating the indexes is slower as a whole than loading with the indexes active, this will mean that the data will be loaded to the table quicker. Any integrity checking that needs to be performed will make a significant difference to the performance of the load. If possible, these integrity checks should be applied on the source systems.

Table Structures

1) Minimize the size (width) of the target table- Often the width of the targets is very large. This can double, triple, quadruple (or more) the I/O necessary to read/write a record. Particularly if there is a clustered key. To reduce the size of the target table, group the fields according to functionality, and break the target up between 2 and 4 tables, all with the same primary key. If they each have a cluster, then have the DBA move each resulting target table on to its own DISK. (Effective partitioning without really partitioning the table). The problem is: think about each wide row in the original table, it over-flows the single block size of the database, forcing a read/write of 2 or more blocks for each rows. Because of this, your I/O is multiplied: (1 write) * (# of blocks for 1 row) - substitute (write) for (read) as well. If you have a clustered key and you need to insert data in a clustered area that's been filled up your formula changes: ((1 write) * (# of blocks for 1 row)) + ((# of rows to move to make room) * (# of blocks for 1 row)) - Thus increasing your I/O activity by multiples. The first step is to break the table apart, so that 1 row fits in 1 database block. That's the objective. The next step (if necessary) is to increase the block size of the database structure itself (often requiring database instance restart), and an empty database

Henceforth the overall result is a speed improvement for concurrent operations. Splitting a wide table also significantly reduces I/O read contention in the same manner - allowing concurrent / parallel queries to operate much faster.

2) Add clustered keys- Adding a clustered key will assist in keeping the data sequenced properly. However, if you have a clustered key in your target table, and your loads are going slowly, attempt at any cost to pre-sort the data in the order of the clustered key. Thus the loads will double or triple in speed, because they no longer have to "move" blocks in order to insert in the middle of a physical location. The load of sorted data (sorted in the order of the clustered key) allows the load to append all of it's data to the end of the clustered key location. Clustered keys also help read speed. Now: if you have a single load

of a huge file, and it's a complete refresh of the target table, then have the Data Architect re-create the table with a fill-factor of 100% (empty space / free space 0%). This will keep the clustered rows tightly packed, and reduce over-all data storage requirements - this ONLY works if

- 1) you have a single load file to refresh the table with,
- 2) you've pre-sorted it in order according to clustered keys.

3) DROP all other indexes prior to load, rebuild them after the load- If the indexes are not dropped and rebuilt they will only slow the load down, usually by some unreasonable factor of 4 or more. However, if the source file is relatively small (< 5000 rows), and relatively narrow, then the load could proceed with the indexes attached. Should your load proceed in this fashion, it becomes imperative that the STATISTICS for the table / Indexes are updated immediately following the load. If you're re-creating the indexes, the database by nature builds the statistics for the index during creation - so update stats is not necessary. Clustered Keys are always correct in their statistics (unless otherwise stated by Database Documentation).

Database Physical Layout

The database can be an important part to the success or failure of the data warehouse which is being built. In order to satisfy user needs, it's nearly always important to tune the database. we feel the need to state that this should be thought about and architected up front, right from the start when the warehouse is being defined. If these things are not planned for, it becomes difficult if not impossible to properly tune the data warehouse. All too often contention for resources, be it CPU, DISK, RAM, or combination thereof can make a huge impact on the speeds of load, processing, and retrieval. In order to address the contention issues, it may be necessary to employ a series of techniques ranging from Performance Monitoring all the way down to discussing the scheduled timing of processing jobs - maybe even consider tuning the SQL statements. However, all of that said, the focus of this contention discussion is to help you arrive at a physical layout that will assist you in solving some of these issues - we also assume that you've already found out that at least one roadblock is the physical nature of the data storage (tables / indexes etc..)

Below are a series of hints, as to what you might want to look at, and why.

1. Separate Tables From Indexes - except Clustered Indexes. Separate: On to two different DISKS, if possible, separate on to two different controllers on two different disks. Why? Because it increases the parallelism of the SQL (inserts, updates, deletes, and selects). It allows the indexes to be read while the table data is being processed. Mostly - it keeps a single head from "skipping" and "thrashing" between disk sectors, one with indexes, then back to the data, then back to the indexes... With it all on the same disk - EACH read takes an equivalent SEEK... Thus doubling your I/O. There it is, separate the indexes, and data , you've just cut the I/O potentially in half, one seek and continuous reads for Indexes, with one seek and continuous reads for data (best possible outcome). Separating further across two different controllers allows the parallel queries to operate in conjunctive fashion. It helps to minimize the I/O through a single controller on-board cache.
2. Cluster the primary key if possible. Clustering the index by primary key (hopefully included with some time continuum) helps to reduce the disk space that each table takes up, and doesn't require any additional storage for the index (which is normally separated from the data anyhow). Clustering the index by business keys works well for data that doesn't utilize surrogate keys. The cluster allows B* tree index seeks to be "continuous" reads with data and index being read in one seek, and one read - thus potentially reducing your I/O again (follow the above scenario).
3. Separate Staging Tables from NDB (Normalized Database) tables. Moving these tables on to separate segments (stripes / disks) - will reduce contention. Allowing the reads from the staging table to not interfere with the writes to the NDB. Again, keep the indexes and the data separated. Don't place the staging indexes on the same disk as the NDB indexes, same goes for the data sets. Organize your "sets" of tables carefully - this will help eliminate contention.
4. Place the TEMP / ROLLBACK / LOG segments on Internal Mirrored Disk. Placing these segments on internal disk makes them FAST!! Super Fast in most cases. Internally mounted disk is always faster than external RAID arrays. Placing the internal disk on a mirror keeps the database from

- having a recovery problem if it crashes. Yes - this is risky, but in times of need, there may be a need for speed.... Sorry for the pun. Also - removing these items to internal disk reduces the possibility that they will contend with the data and indexes that all the other SQL is accessing.
5. Separate the NDB from the Star Schema / Snowflake tables. Again, the disk - keeping contention down means not having a "spike" in any certain area. Don't go for "best contention" level on one single device, try to keep it evenly spread - thus maximizing the throughput between the CPU / RAM and controller on the RAID device (external disk). Separating the NDB from the Star / Snow will keep user access HOT. Typically users are not accessing the warehouse while loads are occurring - in this case it may be safe to combine the data and indexes from the Star / Snow with the data / indexes in the Staging area. But again, keep it out of the NDB area. During the load of the Star / Snow the NDB must remain fast - hence the separation.
 6. Separate the Star Schema / Snow Flake tables from the report collections. Typically at this point, the report collections are placed on a completely different machine - allowing them to operate independently. The report collections are highly indexed, super wide tables. Typically the primary key is TIME - and generally is clustered - for super fast access.

If your database is Oracle, you may have Striping setup on your RAID array - this can win big if it's done right. Raw partitioning is Sybase's requirement, but can be done in Oracle as well. The RAW partitioning works particularly well if you as an architect already have an idea about the data set - how it will grow, how it will be used, and when

Operating System Basics

CPU's can play a huge role in the speed of the machine, particularly when the machine is underpowered. If however, the machine has adequate CPU - there must also be equivalent RAM. In the UNIX world - the rule of thumb is to have 512 MB per CPU available - this ensures that your never running CPU's at 86% or higher. Typically people forget that the CPU and RAM has to do OS work as well as ETL.

once the NDB or Staging area begins to deal with 5+ million rows, say there are 15 staging tables each at 5+ million... Well now - maybe the situation is like this: 1 Million rows, 15 staging tables (add up to 1 M), but you only have 2 hours to process this information in to the NDB, and out to the Stars / Snowflake. This could be a problem. Whatever the case, upgrading this machine again to maybe 8 CPU's and 8 GIG's RAM might not be feasible, then again if it is feasible do it.

OS Doesn't typically require a ton of resources, other than if the CPU's get overloaded, and swap / memory thrashing begins to occur. ETL tools such as Informatica can be CPU bound - if the maps are complex, or contain particular calculations that need to be performed such as aggregation, or lookup matching across multiple keys. Reporting tools are generally fast - don't cause too much trouble until they begin to render the report image. If this is done on the server in memory, then it's both CPU and RAM intensive. It has to create the image in a RAM buffer in order to format it for the "page" views that occur on the client - or are sent to printers. All of this takes place before the report is distributed. Under this guise, in production with nearly 100 users - we typically separate the report server as well. Leaving only the ETL and the DBMS on the Data Warehouse Server.

Data warehousing, regardless of machine or Database is RAM, CPU, and I/O intensive - and if any one of these components are underpowered, the database performance can suffer by orders of magnitude. In the DW world it isn't always the application - something we learn with years of experience. The DW Architect must isolate a single large firing query (long running query) and repeat these tests over and over again, or it could be a troubled load. These are the indicators that help prove the point.

Network Traffic Minimization

In nearly all situations, the data warehouses have reporting tools pulling data from the database. Almost all of these tools aggregate data on the client side to present results. This can not only slow down the network, but over-burden it. There are several simple things an Architect can do to achieve lower network traffic.

1. Place the most commonly used SQL selects in a VIEW in the database. What this does: allows the database to re-use the SQL from the database source itself with multiple users. Also ensures that the results areas in RAM on the server are re-used. Abstracts the reports one layer from the database itself - allowing greater tuning control over the result sets returned.
2. Add Aggregation to the newly created views. Prepare the data on the server in an aggregate form for the reports that use it in that fashion. In this manner - the work is completed on the server, and then only the necessary aggregated data is shipped off to the client. This helps reduce the network traffic.
3. Customize the views to contain only the elements that exist in their particular "series" of reports. Don't be afraid to create views that exist for each "type" or aggregate level of report. Thus again, increasing re-usability by the server and the clients.
4. In a three tier system, attempt to co-locate the machines. Backbone them if you can, otherwise at least put them on the same hub. This will keep outgoing packets to a minimum until they need to be shipped to a client. In a two tier system - all traffic is between the server and the client. Use the techniques above to help minimize.
5. If possible - schedule downloadable files through the reporting tool. Have these files then FTP'd to a web server, and allow the user to pull to their desktop. Or enclose them in an email. These files may allow the user to examine the data in a reasonable fashion using Excel or similar program.

I/O Contention, Resource Monitoring - What it all means

What needs to be known here is that Informatica's products: PM / PC up through 4.7x are NOT built for parallel processing. In other words, the internal core doesn't put the aggregators on threads, nor does it put the I/O on threads - therefore being a single strung process it becomes easy for a part of the session/map to become a "blocked" process by I/O factors. Thus inhibiting faster movement. If I/O is the culprit - it doesn't necessarily show on the performance monitors that are being utilized to gauge machine performance.

Resource Monitoring doesn't always tell the truth about what's going on with system load. Particularly when blocked I/O comes on the scene. It doesn't matter what the cause of the blocked I/O is, it could be a CPU swap, RAM swap, or process requesting data just to name a few. Whatever the cause - the fact remains that I/O is blocked. Which keeps the CPU from processing in most cases? What is reported by the resource monitors is an average load. The average load drops from 80% to 60% for example, when blocked I/O eats time. The CPU is left spinning in idle waiting for the blocked I/O process to return with information before it can continue. This isn't to say that the CPU won't swap the process out - it may do just that - but then again, if resources are near full, a swap might involve disk, so what's the point? Blocked I/O causes the swap of the process off a CPU to wait.

Whenever the CPU is waiting, it's considered 'idle' this in turn reports to the resource monitoring utilities that full capacity is not being utilized. Hence the problem - the average across the CPU's is not true to form. You may be at a 100% capacity for running processes, but you may not be utilizing the CPU at a 100% speed ratios. Take this scenario and average it across the # of CPU's installed in the box, and soon - you end up with an average load being reported of 80%, disk will spike / peak at 80%, and the management might say: what problem? The system isn't showing 100% max utilization, so how can their be a problem. While at the same time, in a perfect world - adding more RAM, installing another controller or attaching a faster disk device would potentially double throughput speeds.... Interesting justification in the way of Hardware Resources.

My generic rule of thumb for this is: if the machine resource monitors hit between 75% and 85% or above 85% on a regular basis, you've potentially got a performance bottleneck. This of course is applying the average machine load across all available resources (network, disk, CPU, RAM, etc). A cool-running system which has room to grow reports utilization less than 40%. An average system with a little room to handle spikes is running between 50% and 70% max, average load at 60% or less.

These are just my estimations based on what I've seen in the field. Of course, I've been able (for other reasons) to justify upgrades to machines where processing appeared to be "slow" according to the business. These machines were running at or above 75% load according to resource monitors, and by upgrading we dropped resource load to 35%, and the processes simply doubled in speed. In some cases, it was that easy.