



Why SeeMoreData is Faster

This document addresses the performance enhancing features of SeeMoreData version 2.5

Native SQL

Native SQL is the SQL standard together with the rdbms-specific enhancements to that SQL, examples of this are using Oracle's DECODE functionality, SQL Server's 'Top 10' SQL variance, Oracle's analytic SQL (partition over, lead, lag, rollup, cube, etc) functionality, TeraData enhanced analytic SQL, etc .

To make the matter even clearer, let us assume an application (be it Siebel, SAP [with transparent non-clustered SAP tables], PeopleSoft or whatever) would like to show the summary of budgeted versus actual costs by division within a corporation. Let us also assume that there is a 6 level hierarchy within the company that goes something like:
Group → department → branch → operating arm → region → corporate

Using a Business Component approach to summarize the budgeted and actual costs, one would have to bring back every single expense and budget record to the application, so that it could get the relevant currency for the region and translate it back to the corporate HO currency and add up each cost all the way up the hierarchy. If there are millions and millions of expense accounts, this will generate a huge amount of network traffic and also on the database, as the database will be required to retrieve each expense and budget record and send it back to the application, where it will then perform the calculations. This type of grunt work is best done by the database either at real time or by using pre-summarized tables or materialized views (Oracle). Also, with Oracle, we can easily display hierarchies using the 'connect by PRIOR' optional predicates of a SQL statement.

By using SeeMoreData, we could effectively put this query into the database where it belongs, or have the top part of the hierarchy execute in a data warehouse or via pre-summarized tables / materialized views, using Oracle dimensions to define the levels and relationships between the hierarchy, thus making use of the native RDBMS engine power.

The other thing that native SLQ allows us to do is to tune the query. Often what will happen is that when base tables are initially loaded, the indexing is balanced, but over time as data is inserted and updated (unfortunately archival is still only found in text books and in university classes these days) the indexing becomes skewed, so that a particular access path that was suitable when a table had 1 million records is not suitable anymore when a table has 40 million rows. In this case, one can over-ride the optimizer decision with a hint. This is something that a Business Object type approach (Siebel & Actuate) does not allow, not to mention just how slow these hierarchical queries would run for.

One could simply insert a hint into Oracle as follows:

```
Select /*+ ORDERED INDEX (C CUSTOMER_NEW_IDX2) */ rest of statement .....
```

The above statement allows one to order the query in the order that the tables appear in the FROM clause and also to specify that a particular index be used for one of the tables accessed in the statement. This becomes very powerful.

Parameters as bind variables.

Most reporting packages have the ability to parameterize a query (select the division for which you wish to see budgeted vs. actual expenses). Other reporting packages and dynamic SQL will then generate the query: `SELECT FROM WHERE DIV='SALES' ;`

If another person runs the same report and wants to see the same information but for the MARKETING division and chooses 'MARKETING' from the list of parameters, the query generated might look something like:
`SELECT FROM WHERE DIV='MARKETING' ;`

This leads to inefficiencies as the shared pool will see 2 statements and then generate 2 separate execution paths for the statements, and not lend itself to sharing of cursors. It is also a very quick way to see where the scalability limits of the reporting application are and in an environment where there are thousands of users generating reports every few minutes, the underlying database will choke rather quickly.

The use of Bind Variables, which are also called Prepared Statements, allows cursor sharing to happen.

SeeMoreData uses prepared statements and for the above query, would generate the statement:
`SELECT FROM WHERE DIV=? ;`

And then pass the rdbms engine the values of 'SALES' to substitute for the question mark. When the next person wants to execute the same query for the 'MARKETING' division, the same statement would appear, thus avoiding the need to parse the statement in the database again, leading to cursor sharing between sessions. The only parsing that would happen would be the 'soft' parsing, which is the final piece of parsing before the statement is actually executed, thus cutting out more than 90% of the work, leading to greater database efficiency.

Where multiple values from a parameter list are chosen, SeeMoreData will create a prepared placeholder for each of these and rewrite the query as:
"SELECT FROM WHERE DIV in (?, ?, ?) "

By doing this, the next time that anyone wants to run the same parameterized report and pick 3 values from the pop-up list of parameter values, the exact same statement will be generated and sent to the database, regardless of which values were chosen, as the actual values will be 'bound' to the prepared statement placeholders.

According to the technical research done, no other reporting package provides for this, which is just yet another reason why most other reporting packages rely on the data first being extracted and dumped into a separate reservoir, like a Datamart, cube or proprietary temporary data storage structure.

Threads

SeeMoreData uses threads to establish connections to databases and SeeMoreData will manage the threads within a session.

Session establishment is a costly exercise, and with a lot of reporting packages, to extract the data, the session is established inside the database, and the query is generated, without any regard to threads.

On an Oracle database system with 3,000 concurrent users, without any pre-spawning of sessions, it may take up to a minute to establish a session, whereas a thread within an existing session will take under a second typically.

Queries written by Developers, not end users

SeeMoreData has taken a very deliberate approach as to who writes statements and queries.

Considering that the ratio of developers to end users in a system is usually 1:20 or thereabouts, it makes very little sense for a system with 200 users to send 200 users on detailed comprehensive training on how to use a reporting package and how to write their own queries. Some people just have difficulty thinking in terms of datasets and trying to understand what exactly the engine is doing or why it is doing it.

Analogy: If you teach a cat to swim like no other cat has ever swum before, in a fast flowing river, it does not matter because the cat will never be a fish. This is similar in some ways to teaching end users how to write reports. The cat can cavort in the water, can play around in a puddle, but in a river with swollen banks after a monsoon rain, the cat is in trouble. So too are end users who have been trained to write queries that extract data from a test system with a few hundred records, when in the real world, they may be facing several hundred million records, and unless they understand relational databases, query optimizing techniques and data transmission, they are going to find themselves in way over their head. A senior accountant at a corporation has great value to that corporation as an accountant. By frustrating themselves for hours on end trying to write a complex report, their effectiveness and usefulness to the corporation is diminished, whereas an experienced reports developer could get the same task done more efficiently and in a way that does not impede the performance of the online application for the other 1,999 users logged on. For instance, a senior accountant who is brilliant at their accounting tasks might take 4 hours to create a report that runs daily for 30 minutes. The same report when created by someone who understand the data model clearly and is experienced in database optimization and understands the reporting package very well, could write the same report in 15 minutes and it might run in 5 seconds. If this is a frequently run report, the system resource savings become rather significant.

Note to Lobbyists: No cats or fish were harmed in any way in conducting the above research.

SeeMoreData Reporter is very intuitive, and would take the average user between 10 and 15 minutes to master how to run the reports, without ever having to know a single thing about how to create it. SeeMoreData Developer training is 1 day, in extreme cases 2 days (depending on the incumbent's exposure to and experience with SQL and understanding of their data model). What we mean is that development of complex queries is better left to those whose nature and work it is to think that way and to those who understand data sets, volumes of data and network transmission limitations.

Network Compression

SeeMoreData conserves network bandwidth by compressing information. There are no other report vendors boasting about this capability.

Assume a report is run in 5 seconds and is 2 MB in size and it is being run by a remote rural office, where 10 people share an ISDN line and a maximum bandwidth of 128 KBPS. Assuming that the report user has the full access of the ISDN line available to them, the report would take

about 140 seconds to transfer across the network to its destination and then render it (assume 5 seconds for rendering). Approximate total response time: 2 ½ minutes.

SeeMoreData however, would compress this 2 MB of data to about 1/10th of its size in approximately 500 milliseconds (current measured compression/decompression throughput is about 4 MB / second), transmit the compressed packet of 200 KB across the network in about 14 seconds, and then uncompress it at the client side and render it (assume 5 seconds for rendering). Approximate total response time: 25 seconds.

Let us extrapolate this same scenario to a point where 2 users are sharing a 64 Kbps line and both users are making heavy use of the network, reducing each user's throughput to 32 Kbps:

2 MB report using uncompressed mode:		2 MB report using compressed mode:	
Database response time:	5 seconds	Database response time:	5 seconds
Network transmission time:	576 seconds	Network transmission time:	57 seconds
Rendering:	5 seconds	Rendering:	5 seconds
	=====		=====
Total Elapsed time	586 seconds	Total Elapsed time	67 seconds
Total Elapsed time in mins:	9 min, 46 sec	Total Elapsed time in mins:	1 min, 7 sec

Now that we have this clear, imagine for a minute that there is a customer waiting in front of the reports user, wanting the details of all the orders they have placed with you over the last 2 years (this is the report being generated) and the importance of conserving network resources starts to become very, very clear.

Typical compression ratios vary from 1:8 to 1:20, depending on the kind of data. The average rule of thumb is 1:10.

Report Caching

SeeMoreData allows the caching of reports either at a generic (all users) level or at a per user level. This is in order to significantly enhance performance.

Take the example of 100 Call Centre employees at a company that need to access the report "Last week's service requests – FAULTS". As long as we remain in 'this week', this data should not change, also due to SeeMoreData's built-in functionality to determine exactly when 'last week' started and ended. Let us also assume this report runs for **15 minutes** every time it runs, and the report needs to be generated at the start of each new day by all 100 of the above employees and the report generated contains 3,000 lines (about 50 pages).

Thus in an ordinary situation, if each employee ran this 5 days a week, this report would consume 100 * 15 minutes of processing time * 5 days per week, thus **7,500 minutes** of processing that the database would have to service. Thus **125 CPU hours** or just over 5 days' worth of CPU power is needed to process this one report for 100 users.

SeeMoreData provides the option for the Scheduler to run the report once on Monday morning at 3 a.m. and then cache the report result set. When the user requests the report, SeeMoreData will see that there is a cached report dataset available whose cache retention will expire the following Monday @ 3:00 when the report is next run by the scheduler.

Thus instead of going to the database to process the report, SeeMoreData interrogates its repository to see where the cached report data is stored on its reports cache directory and then it reads this file in as a byte stream, processing the report in about 300 milliseconds, saving the

database from doing any work at all, returning the report to the user **in < 2 seconds**, instead of the 5 minutes. All the work is carried out on the application server, never touching the database.

The user will see that the background colour for the report title has changed, to signify that the report was obtained from a cached dataset, and not directly against the database, giving the user the option to refresh the report (and the cache) from the database to get an up-to-date report.

Cache retention periods can be set anywhere from seconds, to 99 days.

A report can also optionally be cached for a single user, thus if the same user has a need to generate the same report frequently and the data does not change (typical of drilldown top-level reports, where the drilldown children are really the focus of the report), then this too can be set. An example would be 'My Pending Opportunities' type of report, and once again, the retention period can be set from ridiculous (a few seconds) to extreme (99 days).

SeeMoreData also allows the **caching of a complex Data Object**. Imagine the situation where a complex joined dataset provided all of the new customers for **last month**, together with their credit histories & ratings, their stated purchasing intentions and any referral information, which could be the result of a complex join between 3 or more disparate systems. There may be a total of 1,000 new customers for last month and this report may have taken 20 minutes to generate, but the information contained therein is frequently needed by many other reports.

We could write reports to select from the data object rather than having to recode the system join again, and if there are various reports that access this data object, we would not want to incur the 20 minute cost of getting the data each time, since a 1,000 row dataset takes in the order of 100 milliseconds to load from a cached dataset.

Thus if there are potentially 50 reports that access the data objects and an unknown number of users running those reports, the information can suddenly be made available **VERY** fast, as opposed to having to wait 20 minutes for each execution of the data object recursively, thus adding to overall **speed, efficiency** and **user satisfaction**.