International Journal of Computer Applications Technology and Research
Volume 4– Issue 7, 529 - 534, 2015, ISSN:- 2319–8656

A Review of Data Access Optimization Techniques in a Distributed Database Management System

Sadiq Mobolaji Abubakar
Department of Computer Science
University of Port Harcourt
Nigeria

Anyama Oscar Uzoma
Department of Computer Science University of Port Harcourt
Nigeria

Adamade Peter Simon
Department of Computer Science
University of Port Harcourt
Nigeria

Abstract: In today's computing world, accessing and managing data has become one of the most significant elements. Applications as varied as weather satellite feedback to military operation details employ huge databases that store graphics images, texts and other forms of data. The main concern in maintaining this information is to access them in an efficient manner. Database optimization techniques have been derived to address this issue that may otherwise limit the performance of a database to an extent of vulnerability. We therefore discuss the aspects of performance optimization related to data access in distributed databases. We further looked at the effect of these optimization techniques.

Keywords: Data; Distributed database; Performance; TSQLANN

1. INTRODUCTION

It’s a known fact that the amount of data that enterprises are storing and managing is growing rapidly. Industry estimates indicate that data volume is doubling every 2-3 years. The rapid growth of data presents frightening challenges for IT, both in cost and for our study, performance. Although the cost of storage keeps declining, fast-growing data volumes make storage one of the costliest elements of most IT budgets. In addition, the accelerating growth of data makes it difficult to meet performance requirements while staying within budget.

When a database based application performs slowly, there is a 90% probability that, the data access routines of that application are not written in the best possible way or optimized, In this paper we will discuss Data access performance optimization in transactional SQL Server databases and will also consider the performance of a very large database with and without our suggested optimization. Though the optimization techniques are suggested for transactional SQL Server databases but most of the techniques are roughly the same for other database platforms. In oracle 12c, Automatic Data Optimization (ADO) automatically moves and compresses data according to user-defined policies based on the information collected by Heat Map, [1].

1.1 Performance Optimization Techniques

It is worth mentioning here that all forms of optimization actually enhances the performance of the database and below are some of the techniques employed.

Indexing in the table column:

We need to create primary key in every table of the database. When we create a primary key in a table, a clustered index tree is created and all data pages containing the table rows are physically sorted in the file system according to their primary key values. Each data page contains rows which are also sorted within the data page according to their primary key values. [2], pointed that each time any row from the table is asked for; the database server finds the corresponding data page first using the clustered index tree and then finds the desired row within the data page that contains the primary key value.

The intermediate nodes contain range of values and direct the SQL engine where to go while searching for a specific index value in the tree starting from the root node. The leaf nodes are the nodes which contain the actual index values. If this is a clustered index tree, the leaf nodes are the physical data pages. If this is a non-clustered index tree, the leaf nodes contain index values along with clustered index keys (Which the database engine uses to find the corresponding row in the clustered index tree). Usually, finding a desired value in the index tree and jumping to the actual row from there takes an extremely small amount of time for the database engine. So, indexing generally improves the data retrieval operations which a performance enhancement strategy, [3].

Movement from application into the database server of SQL Codes:

Moving the SQLs from application and implementing these using stored procedures/ Views/ Functions/ Triggers will enable us to eliminate any duplicate SQLs in our application. This will also ensure reusability of our TSQL codes. Implementing all TSQLs using the database objects will enable us to analyze the TSQLs more easily to find possible inefficient codes that are responsible for slow performance. Also, this will let us manage our TSQL codes from a central point, [4].

Doing this will also enable us to re-factor our TSQL codes to take advantage of some advanced indexing techniques. Also, this will help us to write more “Set based” SQLs along with eliminating any “Procedural” SQLs that we might have already written in our application. Despite the fact that indexing will let us troubleshoot the performance problems in our application in a quick time, following this step might not give us a real performance boost instantly. But, this will mainly enable us to perform other subsequent optimization steps and apply different other techniques easily to further optimize our data access routines.

1.2 Covering Index:

If we know that our application will be performing the same query over and over on the same table, we should consider creating a covering index on the table. A covering index, which is a form of a composite index, includes all of the columns referenced in SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data we are looking for and SQL Server doesn't have to look up the actual data in the table, reducing logical and/or physical I/O, and boosting performance.

1.3 File Organization in groups and files in the database:

When an SQL Server database is created, the database server internally creates a number of files in the file system. Every database related object that gets created later in the database are actually being stored inside these files. An SQL Server database has the following three kinds of files, [5];

mdf file: This is the primary data file. There could be only one primary data file for each database. All system objects resides in the primary data file and if a secondary data file is not created, all user objects (User created database objects) also takes place in the primary data file.

ndf file: These are the secondary data files, which are optional. These files also contain user created objects.

ldf file: These are the Transaction log files. These files could be one or many in number. It contains transaction logs. Database files are logically grouped for better performance and improvement of administration on large databases. When a new SQL Server database is created, the primary file group is created and the primary data file is included in the primary file group. Also, the primary group is marked as the default group. As a result, every newly created user objects are automatically placed inside the primary file group (More specifically, inside the files in the primary file group). If our database has a tendency to grow larger (Say, over 1000 MB) in size, we can (and should) do a little tweaking in the file/file group organizations in the database to enhance the database performance. Here are some of the best practices we can follow:

The primary file group must be totally separate and should be left to have only system objects and no user defined object should be created on this primary file group. Also, the primary file group should not be set as the default file group. Separating the system objects from other user objects will increase performance and enhance ability to access tables in the case of serious data failures, [8].

If there are N physical disk drives available in the system, then we should try to create N files per file group and put each one in a separate disk. This will allow Distributing disk I/O loads over multiple disks and will increase performance.

For frequently accessed tables containing indexes we should put the tables and the indexes in separate file groups. This would enable to read the index and table data faster.

We should put the transaction log file on a different physical disk that is not used by the data files. The logging operation (Transaction log writing operation) is more write-intensive, and hence, it is important to have the log on the disk that has good I/O performance.

Inefficient TSQLs identification and re-factoring best practices application:

Knowing the best practices is not enough at all. The most important part is we have to make sure that we follow the best practices while writing TSQLs. Some TSQL Best practices are described here, [6]:

We should not use “SELECT *” in SQL Query because then unnecessary columns may get fetched that adds expense to the data retrieval time and the Database engine cannot utilize the benefit of “Covered Index” hence, query performs slowly.

We should not use COUNT() aggregate in a sub query to do an existence check because when we use COUNT(), SQL Server does not know that we are doing an existence check. It counts all matching values, either by doing a table scan or by scanning the smallest nonclustered index. But if we use EXISTS, SQL Server knows you are doing an existence check. When it finds the first matching value, it returns TRUE and stops looking.

We should try to avoid joining between two types of columns because when joining between two columns of different data types, one of the columns must be converted to the type of the other. The column whose type is lower is the one that is converted. If we are joining tables with incompatible types, one of them can use an index, but the query optimizer cannot choose an index on the column that it converts.

We should try to avoid the use of Temporary Tables unless really required. Rather, try to use Table variables. Almost in 99% case, Table variables reside in memory; hence, it is a lot faster. But, Temporary tables reside in “TempDb” database. So, operating on Temporary table requires inter db communication and hence, slower.

We should try to avoid deadlock. We should always access tables in the same order in all our stored procedures and triggers consistently and keep our transactions as short as possible. Also should touch as few data as possible during a transaction and should never, ever wait for user input in the middle of a transaction.

We should write TSQLs using “Set based approach” rather than using “Procedural approach”. The database engine is optimized for set based SQLs. Hence, procedural approach (Use of Cursor, or, UDF to process rows in a result set) should be avoided when large result set has to be processed. By using inline sub queries to replace User Defined Functions and by using correlated sub queries to replace Cursor based codes we can get rid of “Procedural SQLs”

We should use Full Text Search for searching textual data instead of LIKE search as Full text search always outperforms the LIKE search. Full text search will enable us to implement complex search criteria that can’t be implemented using the LIKE search such as searching on a single word or phrase, searching on a word or phrase close to another word or phrase, or searching on synonymous forms of a specific word.

We should try to use “UNION” instead of “OR” in the query. If distinguished result is not required we better use “UNION ALL” because “UNION ALL” is faster than “UNION” as it does not have to sort the result set to find out the distinguished values. Here we worked on millions of data with some complex query and got the results in seconds.

1.4 Partitioning the big fat tables

Table partitioning means nothing but splitting a large table into multiple smaller tables so that, queries has to scan less amount data while retrieving. That is “Divide and conquer”. When we have a large (In fact, very large, possibly having more than millions of rows) table in our database we should consider portioning this table to improve performance, [7].

Suppose we have a table containing 10 millions of rows, let’s assume that, the table has an auto-increment primary key field

(Say, ID). So, we can divide the table's data into 10 separate portioning tables where each partition will contain 1 million rows and the partition will be based upon the value of the ID field. That is, First partition will contain those rows which have a primary key value in the range 1-1000000, and, Second partition will contain those rows which have a primary key value in the range 1000001-2000000 and so on.

2. RELATED WORK

Review of most literature on database optimization pointed to the fact that optimization is always considered in relation to databases but performance is always relegated to the background or not mentioned at all.

[9], emphasized more on optimization techniques without mention of performance.

[7], in Optimization Techniques of Queries with Expensive Methods, studied queries that contain time consuming methods.

He carefully defined a query cost framework that incorporates selectivity and cost estimates for selection. A lot of other details on queries were dealt with but without mention of how performance is further enhanced by optimizing the queries.

In The Principles of Query Optimization in Relational Database Management Systems, Johann Christopher Freytag describe a wide variety of different optimization algorithms for query languages.

3. INFORMATION LIFECYCLE (ILM)

Information Lifecycle Management (ILM) is intended to address challenges of accessing data by storing it in different storage and compression tiers, according to the enterprise's current business and performance needs. This approach offers the possibility of optimizing storage for both cost savings and maximum performance.

In Oracle Database 12c, two new ILM-related features have been added to the Advanced Compression Option. Heat Map automatically tracks modification and query timestamps at the row and segment levels, providing detailed insights into how data is being accessed. Automatic Data Optimization (ADO) automatically moves and compresses data according to user-defined policies based on the information collected by Heat Map.

Heat Map and ADO make it easy to use existing innovations in Oracle Database Compression and Partitioning technologies, which help reduce the cost of managing large amounts of data, while also improving application and database performance. Together these capabilities help to implement first-class Information Lifecycle Management (ILM) in Oracle Database.

3.1 Storage Tiering and Compression Tiering

An enterprise (or even a single application) does not access all its data equally: the most critical or frequently accessed data will need the best available performance and availability. To provide this best access quality to all the data would be costly, inefficient, and is often architecturally impossible. Instead, IT organizations implement storage tiering, by deploying their data on different tiers of storage so that less-accessed ("colder") data are migrated away from the costliest and fastest storage – still available, but at slower speeds, whose effect on the overall application performance is minimal, due to the rarity of accessing those "colder" data. Colder data may also be compressed in storage. We use the term Information Lifecycle Management (ILM)1 to name the managing of data from creation/acquisition to archival or deletion.

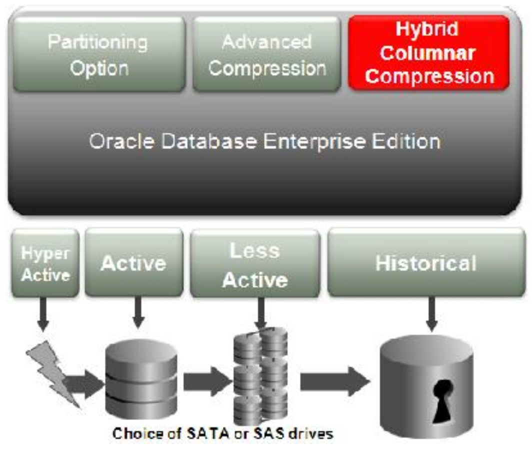
Partitioning Option
Advanced Compression
Hybrid Columnar Compression
Oracle Database Enterprise Edition
Hyper Active
Active
Less Active
Historical
Choice of SATA or SAS drives

Fig1: Partitioning, Advanced Compression and Hybrid Columnar Compression

Figure 2 shows the most active data located on a high performance tier and the less active data/historical data on lower-cost tiers. In this scenario, the business is meeting all of its performance, reliability, and security requirements, but at a significantly lower cost than in a configuration where all data is located on high performance (tier 1) storage. The illustration shows that compression can be applied to the less active and historical storage tiers, further improving the cost savings while also improving performance for queries that scan the less active data.

In addition to storage tiering, it is also possible to use different types of compression to suit different access patterns. For example, colder data may be compressed more at the cost of slower access.

Oracle, even with the right storage and compression capabilities, deciding which data should reside where and when to migrate data from one tier to another remains a serious challenge. Oracle Database 12c addresses this challenge with functionality that automatically discovers data access patterns – Heat Map – and uses Heat Map information to automatically optimize data organization – Automatic Data Optimization. The rest of this document explains the Oracle Database technologies that enable storage and compression tiering, and how to use them to support Information Lifecycle Management.

3.2 Heat Map: Fine-grained Data Usage Tracking

Heat Map is a new feature in Oracle Database 12c that automatically tracks usage information at the row and segment levels.2 Data modification times are tracked at the row level and aggregated to the block level, and modification times, full table scan times, and index lookup times are tracked at the segment level. Heat Map gives you a detailed view of how your data is being accessed, and how access patterns are changing over time. Programmatic access to Heat Map data is available through a set of PL/SQL table functions, as well as through data dictionary views as in figure 3 below

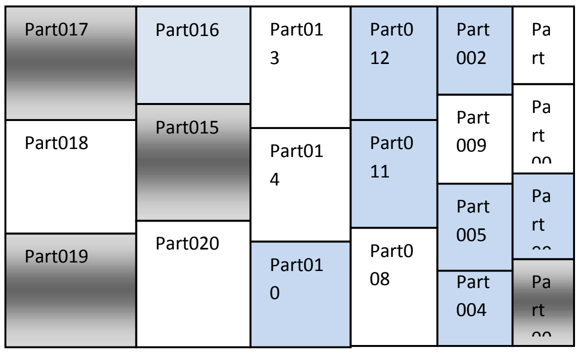

Fig 2. Heat Map data for access patterns to a partitioned table

Database rows are stored in database blocks, which are grouped in extents. A segment is a set of extents that contains all the data for a logical storage structure within a table space, i.e. a table or partition. The colour code is as detailed below for clarity;

GREY	WHITE	SKY BLUE
Part017	Part018	Part016
Part015	Part020	Part010
Part019	Part013	Part012
Part006	Part014	Part011
Part008	Part002	
Part009	Part005	
Part003	Part004	
Part007	Part001	

3.3 Automatic Data Optimization

Automatic Data Optimization (ADO) allows you to create policies for data compression (Smart Compression) and data movement, to implement storage and compression tiering. Smart Compression refers to the ability to utilize Heat Map information to associate compression policies, and compression levels, with actual data usage. Oracle Database periodically evaluates ADO policies, and uses the information collected by Heat Map to determine when to move and / or compress data. All ADO operations are executed automatically and in the background, without user intervention.

ADO policies can be specified at the segment or row level for tables and table partitions. Policies will be evaluated and executed automatically in the background during the maintenance window. ADO policies can also be evaluated and executed anytime by a DBA, manually or via a script.

ADO policies specify what conditions (of data access) will initiate an ADO operation – such as no access, or no modification, or creation time – and when the policy will take effect – for example, after n days or months or years. Conditions in ADO policies are not limited to Heat Map data: you can also create custom conditions using PL/SQL functions, extending the flexibility of ADO to use your own data and logic to determine when to move or compress data.

3.4 Automatic Data Optimization Examples

The following examples assume there is an orders table containing sales orders, and the table is range partitioned by order date.

In the first example, a segment-level ADO policy is created to automatically compress partitions using Advanced Row Compression after there have been no modifications for 30 days. This will automatically reduce storage used by older sales data, as well as improve performance of queries that scan through large numbers of rows in the older partitions of the table.

ALTER TABLE orders ILM ADD POLICY ROW STORE COMPRESS ADVANCED SEGMENT AFTER 30 DAYS OF NO MODIFICATION;

Sometimes it is necessary to load data at the highest possible speed, which requires creating a table without any compression enabled. It would be useful to later compress the data in the table, on a more granular basis than entire partitions. With ADO, you can create a row-level ADO policy to automatically compress blocks in the table (using Advanced Row Compression) after no row in a given block has been modified for at least 3 days. This is an example of OLTP background compression, in which rows are inserted uncompressed, and then later moved to Advanced Row Compression on a per-block basis. Note that this policy uses the ROW keyword instead of the SEGMENT keyword.

ALTER TABLE orders ILM ADD POLICY ROW STORE COMPRESS ADVANCED ROW AFTER 3 DAYS OF NO MODIFICATION;

With the above policy in place, Oracle Database will evaluate blocks in the orders table during the maintenance window, and any blocks that qualify will be compressed in place, freeing up space for new rows as they are inserted. This allows you to achieve the highest possible performance for data loads, but also get the storage savings and performance benefits of compression without having to wait for an entire partition to be ready for compression.

In addition to Smart Compression, ADO policy actions include data movement to other storage tiers, including lower cost storage tiers or storage tiers with other compression capabilities such as Oracle' s Hybrid Columnar Compression (HCC).

In the following example, a tablespace-level ADO policy automatically moves partitions to a different tablespace when the current tablespace runs low on space. The “tier to” keywords indicate that data will be moved to a new tablespace when the current tablespace becomes too full. The user has control over the threshold that triggers storage tiering actions with PL/SQL-based ILM admin functions. The “low_cost_store” tablespace was created on a lower cost storage tier. Note that it is possible to add a custom condition to tiering policies, allowing you to trigger movement of data based on conditions other than how full the tablespace is.

ALTER TABLE orders ILM ADD POLICY tier to low_cost_store;

In the following example, a segment-level ADO policy is created to automatically compress partitions using Hybrid Columnar Compression after there have been no modifications for 30 days. This makes sense when HCC is available, and when the data will no longer be updated, but will continue to be queried; moving to HCC will save a lot of storage AND give a big boost to query performance.

ALTER TABLE orders ILM ADD POLICY COLUMN STORE COMPRESS FOR QUERY HIGH SEGMENT AFTER 30 DAYS OF NO MODIFICATION;

Another option when moving a segment to another tablespace is to set the target tablespace to READ ONLY after the object is moved. This is beneficial for historical data and during backups, since subsequent RMAN full database backups will skip READ ONLY tablespaces.

3.5 Automatic Data Optimization for OLTP.

The previous examples show individual ADO policies that implement one action –compression tiering (Smart Compression) or storage tiering. The following example shows how to combing multiple ADO policies for an OLTP application.

In OLTP applications, you should use Advanced Compression for the most active tables/partitions, to ensure that newly added or updated data will be compressed as DML operations are performed against the active tables/partitions.

For cold or historic data within the OLTP tables, use either Warehouse or Archive Hybrid Columnar Compression. This ensures that data which is infrequently or never changed is compressed to the highest levels – compression ratios of 6x to 15x are typical with Hybrid Columnar Compression, whereas 2x to 4x compression ratios are typical with Advanced Row Compression.

To implement this approach with ADO, use the following policies:

Figure 4. Advanced Row Compression, Hybrid Columnar Compression, and tiering.

ALTER TABLE orders ILM ADD POLICY COLUMN STORE COMPRESS FOR QUERY HIGH SEGMENT AFTER 30 DAYS OF NO MODIFICATION;

ALTER TABLE orders ILM ADD POLICY COLUMN STORE COMPRESS FOR ARCHIVE HIGH SEGMENT AFTER 90 DAYS OF NO MODIFICATION;

ALTER TABLE orders ILM ADD POLICY tier to low_cost_store;

In this example of Smart Compression and storage tiering, we assume that the orders table is defined with Advanced Row Compression enabled, so that rows are compressed at that level when they are first inserted. Oracle Database will automatically evaluate the ADO policies to determine when each partition is eligible to be moved to a higher compression level, and when each partition is eligible to be moved to a lower cost storage tier. As discussed earlier, storage tiering is primarily triggered when the current tablespace becomes too full, but can be customized to occur based on user-defined conditions.

The capabilities of Heat Map and ADO in Oracle Database 12c make it easy for DBAs to implement ILM for OLTP applications, and enable the use of HCC with OLTP data. With HCC, DBAs can significantly reduce the amount of storage space used by OLTP data, while increasing the performance of reports and analytics.

3.5.1 *Automatic Data Optimization and Data Warehousing*

In data warehousing applications on Exadata or on Oracle Storage that supports HCC, Warehouse Compression should be used for heavily queried tables/partitions. For cold or historic data within the data warehousing application, using Archive Compression ensures that data which is infrequently accessed is compressed to the highest level – compression ratios of 15x to 50x are typical with Archive Compression.

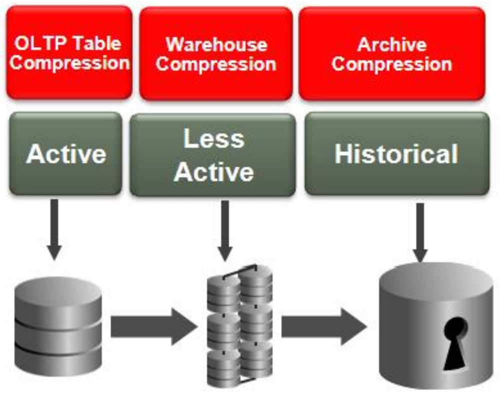

Fig5. Partitioning and Hybrid Columnar Compression.

To implement this approach with ADO, use the following statements:

ALTER TABLE orders ILM ADD POLICY COLUMN STORE COMPRESS FOR ARCHIVE HIGH SEGMENT AFTER 90 DAYS OF NO MODIFICATION;

ALTER TABLE orders ILM ADD POLICY tier to lessactivetbs;

In this example, we assume that the orders table is defined with Warehouse Compression enabled, so that rows are compressed at that level when they are first inserted. Oracle Database will automatically evaluate the ADO policies to determine when each partition is eligible to be moved to a higher compression level, and when each partition is eligible to be moved to a different tablespace. As with the previous Smart Compression example for OLTP, the automatic capabilities of ADO in Oracle Database 12c make it simple and easy for DBAs to implement ILM for Data Warehousing, and significantly reduce the amount of time and effort DBAs need to spend optimizing storage usage and storage performance.

4. CONCLUSION

Information Lifecycle Management (ILM) should enable organizations to understand how their data are accessed over time, and manage the data accordingly. However, most ILM solutions for databases lack two key capabilities – automatic classification of data, and automatic data compression and movement across storage tiers.

The Heat Map and Automatic Data Optimization features of Oracle Database 12c support comprehensive and automated ILM solutions that minimize costs while maximizing performance. In combination with the comprehensive compression features in Oracle Database 12c, Oracle Database 12c provides an ideal platform for implementing ILM.

Furthermore in this paper we have suggested very few other performance optimization techniques in transactional (OLTP) SQL Server databases. Optimization is a “Mindset”, rather than an automatic occurrence. In order to optimize access in our database performance, first we have to believe that, optimization is possible. Then we need to give our best effort and apply knowledge and best practices to optimize. The most important part is, we have to try to prevent any possible performance issue that may take place later, by applying our knowledge before or along with our development activity, rather than trying to recover after the problem occurs.

5. REFERENCES

[1] Churcher, C., 2007. Beginning Database Design: From Novice to Professional, Apress, ISBN-10: 1590597699, ISBN-13:978-1590597699.

[2] Date, C. J., 2003. An Introduction to Database Systems", Addison Wesley, ISBN-10: 0321197844, ISBN-13: 978-0321197849.

[3] Date, C. J., 2000. Foundation for Future Database Systems: The Third Manifesto", Addison-Wesley Professional, ISBN-10:0201709287, ISBN-13: 978-0201709285.

[3] Codd, E. F., 2002. The relational model for database management: version 2", Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, ISBN:0-201-14192-2.

[4] Joseph M. H., 2000. Optimization Techniques of Queries with Expensive Methods.

[5] Freytag J. C., 1989. The Principles of Query Optimization in Relational Database Management Systems.

[6] Jernigan, K. Christman, G. C. Pedregal. Automatic Data Optimiization with Oracle. Database 12c, 2013.

[7] Hernandez, M. J. Database Design for Mere Mortals. A Hands-On Guide to Relational Database Design, Addison-Wesley Professional, ISBN-10: 0201694719, 1996.

[8] Asagba P. O. Distributed Processing and Distributed Database System. Journal of Applied. Science Environmental Management. Vol. 18 (2) 249-253, 2014.

[9] Zhiyuan Chen et al. Query Optimization in Compressed Database Systems. International Journal of Computer Science and Network Security (IJCSNS), VOL.10 No.8, 2010.