The Impact of Partitioning on Performance of Database and Data Warehouse Systems

Dr. Abdallah Alashqur
Applied Science University
Amman, Jordan
alashqur@asu.edu.jo

Abstract

The increase in power and capacity of hardware systems coupled with the decrease of hardware costs made it possible for institutions and corporations to store larger quantities of data in their database and warehouse systems than ever before. Multi-terabyte databases are becoming more widely spread than in the past. This creates a need to improve the performance of data retrieval and data manipulation operations in such large databases. Techniques such as bitmap indexes, materialized views, and partitioning have been incorporated in many state-of-the-art database management systems. In this paper, we provide an overview of the different partitioning techniques that have been introduced in the literature, then we present the results of an analysis that quantitatively demonstrates the positive impact that partitioning can have on query performance in database and data warehouse systems.

Key Words: *database fragmentation, distributed database, disk reads, Oracle, indexing.*

1. INTRODUCTION

Advanced optimization techniques have been proposed to improve query performance in very large databases (VLDB) and data warehouse environments. These techniques include materialized views [1,2,3,11,16], partitioning [6,5,14,17], parallel query processing [12,18], and special indexing methods such as bitmap indexes and join indexes [8,9,10,15,16].

Partitioning is a technique through which a relation (table) is divided into partitions (fragments). Each partition is stored on a different node in a multi-node architecture or in a separate file segment in a single-node system. There are two major types of partitioning as described in [17], namely, vertical partitioning and horizontal partitioning. In a vertical partitioning approach, a relation is divided into subsets of attributes (columns or fields). Each subset of attributes along with its tuples (rows) represents a partition. These subsets of attributes are not necessarily disjoint, since the key attributes may need to be replicated in all partitions. The key attributes are used to relate data values from different partitions, in order to reconstruct an entire tuple of the relation. In a horizontal partitioning approach, on the other hand, a relation is divided into subsets of tuples. Each subset is a partition of the relation. A mix of horizontal and vertical partitioning is desired in some cases when the database is extremely large and geographically distributed.

The advantages of partitioning include (1) increased database manageability and (2) improved performance. Regarding manageability, a Data Base Administrator (DBA) can, for example, gather statistics partition-wise, therefore reducing the time slots needed for administrative tasks. Also, relation re-organization and index re-creation can be done partition-wise, which helps reduce the off-line time needed for database maintenance. With respect to performance, in the case of range and hash partitioning (see section 2 for a description) the database optimizer can benefit from partitioning by performing partition elimination. This results in search being performed only on the partitions that contain the tuples that satisfy the search conditions, which is faster than searching the entire relation. Also, in case of equi-join queries, joins can be performed partition-wise (see section 2 for a description of partition-wise joins), resulting in a considerable improvement of performance. Partitioning can also improve the performance of mass-deletions to remove out-dated data. Instead of deleting one tuple at a time, partitions as a whole can be dropped or archived to tape. Dropping a whole partition substantially outperforms tuple-at-a-time deletions.

In this paper we focus on horizontal partitioning as a technique to achieve high performance levels in very large databases and data warehouses. The objective of this paper is to quantitatively demonstrate the performance gains that can be achieved by applying some horizontal partitioning strategies to large relations in a database. The remainder of this paper is organized as follows. In Section 2, we provide an overview of horizontal partitioning strategies, namely range and hash partitioning. We also describe index partitioning techniques. In Section 3, we describe the database used as a test platform, by describing the schema and set of test queries that we used. We also explain the partitioning strategies that were applied to large relations and indexes in this database. Section 4 provides an analysis of the test results and shows a comparison between the performance after the large relations were partitioned to the performance before partitioning was applied. Conclusions are presented in Section 5.

2. HORIZONTAL PARTITIONING – OVERVIEW AND ADVANTAGES.

In this section we give a brief description of horizontal partitioning of relations. We also describe index partitioning.

2.1 RELATION PARTITIONING

Different horizontal partitioning strategies have been described in the literature [4,5,14,17]. These partitioning strategies are mainly round-robin partitioning, range partitioning, and hash partitioning. A combination of range and hash partitioning is possible and is referred to as mixed partitioning. In round-robin partitioning, tuples of a relation are randomly spread over the partitions following a round-robin algorithm, where each partition resides on a different disk. This balances the loading of data over disks. The performance gain is achieved by enabling the application of data predicates in parallel. The query optimizer does not have knowledge of which partitions (disks) may contain the data satisfying a predicate. Therefore, when an efficient access path based on an index is not found, all partitions have to be scanned. On the other hand, when range or hash partitioning is employed, the optimizer may have enough information to determine the partition where the data satisfying a predicate is located and therefore searches only that partition. To get a feel for the significance of performance improvement, assume there is a large relation that has 500 million tuples. If that relation is partitioned to, say, 500 partitions, with an average of one million tuples per partition, then if the optimizer knows which partition contains the requested data, it will search only that partition. This means a partition of only one million tuples has to be searched as opposed to searching an entire relation of 500 million tuples. This considerably improves query response time.

Below we give a more detailed description of range and hash partitioning since they are used in conducting the performance analysis described in the remainder of this paper. In range and hash partitioning techniques, one or more attributes of the relation are designated as the partitioning attributes. The values of the partitioning attributes of a tuple determine the partition that the tuple will be stored in.

Range Partitioning. If a relation is range-partitioned, then a partition stores all tuples whose partitioning attribute value lies within a given range. The range for each partition has to be specified at relation creation time. For example, if a relation is partitioned on a date attribute, and the range specified is 'monthly,' then all the tuples that share the same month value of the partitioning attribute go to the same partition. An advantage of range partitioning is that the query optimizer knows the partition in which a tuple is stored by examining the partitioning attribute. For example, a predicate on the date partitioning attribute of a relation R of the form *R.date_attr > 'July 10, 2006' and R.date_attr < 'August 25, 2006'* will lead the optimizer to narrow down the search to the July and August partitions, therefore *eliminating* all other partitions before starting the search. *Partition elimination* is one of the powerful optimization techniques that the optimizer can use. A second optimization technique is referred to as *partition-wise joins*. This technique is used when two relations are range-partitioned the same way and there is an equi-join condition between the two relations on the partitioning attribute. For example if relations R1 and R2 are partitioned monthly on date attributes date_attr1 and date_attr2, respectively, and there is a join predicate of the form *R1.date_attr1 = R2.date_attr2*, then tuples of each partition of R1 are compared with tuples of the corresponding partition of R2 and not with all the tuples of R2. Partition-wise joins provide significant performance improvement of join operations. Range partitioning is appropriate when the number of distinct values of the partitioning attribute is small. For example, in a monthly partitioning on a date attribute, there will be only 12 partitions per year. Therefore a relation whose data span a 5-year period contains only 60 partitions.

Hash partitioning. Unlike rage partitioning, hash partitioning is useful when the partitioning attribute has a large number of distinct values. Similar to range-partitioning, hash partitioning needs to be specified at the time of creating a relation in the database. When a tuple is inserted, a hash function is applied to the value of the partitioning attribute. Based on the value returned by the hash function, the tuple is stored in the right partition. The query optimizer can employ partition-elimination and partition-wise join strategies when hash partitioning is used. However, partition-elimination requires that the predicate has to be an equality predicate or a predicate that uses the IN operator, and not a range predicate.

Mixed partitioning. This is a mixture of range and hash partitioning. In this approach, a relation is range-partitioned on a partitioning attribute, then each partition is further hash-sub-partitioned based on another partitioning attribute. When a tuple is inserted in such a relation, the first partitioning attribute is examined to determine the partition, then the second partitioning attribute is examined to determine the sub-partition where the tuple should be stored. So, tuples are not actually stored in the first-level partitions but in their sub-partitions. A first level partition (based on range partitioning), in this case, merely serves as a logical grouping of sub-partitions, and does not physically store any tuples. In a mixed partitioning strategy, the query optimizer can benefit from partition elimination and partition-wise joins to achieve better performance. This strategy is useful for partitioning extremely large relations which require more than one level of partitioning in order to reduce the size of partitions to a reasonable size.

2.2 INDEX PARTITIONING

In addition to partitioning relations, indexes can also be partitioned. Just like relation partitioning, index partitioning can also lead to performance improvements. If an index is partitioned and the execution plan of a query involves scanning the index, the optimizer in this case attempts first to scan only those partitions of the index that satisfy the predicate.

Index partitioning can be performed regardless of whether the underlying relation is partitioned or not. When both a relation and its index are partitioned, the index can be *equi-partitioned* or *non-equi-partitioned* with the underlying relation. An equi-partitioned index is an index that is partitioned on the same partitioning attributes as the underlying relation. On the other hand, a non-equi-partitioned index is an index that is partitioned on different partitioning attributes (see figures 1.a and 1.b).

In the equi-partitioned case, each index partition is used to index tuples residing only in one relation partition. In other words, each partition of the index corresponds to one and only one partition of the underlying relation. In a non-equi-partitioning scenario (Figure 1.b), on the other hand, one index partition is used to index tuples that belong to different relation partitions.

In addition to performance benefits, equi-partitioned indexes provide other benefits. For example, a relation partition can be dropped or archived along with its index partition without having to rebuild the entire index.

3. SCHEMA AND QUERIES USED IN THE PERFORMANCE ANALYSIS

In this section we describe the schema, database, and queries used in conducting this performance comparison analysis. We have created the schema shown in Figure 2, in an Oracle database. This schema represents a financial application.

As shown in Figure 2, a client can have multiple accounts as represented by the Account relation. Each account has multiple related tuples in the Account_History relation. Each Account_History tuple has an Effective_Date that identifies the start date of a period in the life span of an account. Account_Holding relation stores the names and IDs of the financial instruments (stocks, ETFs, mutual funds, etc.) that were held by an account during each period in its history. Some of those Account_Holding tuples have related Account_Holding_Detail tuples that show further detail about the holding. Accounts can be grouped into account groups as represented by the Account_Group relation.

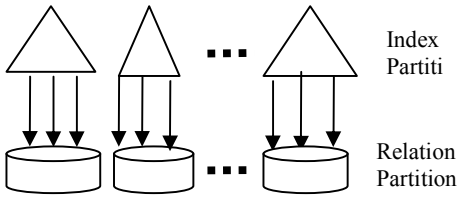

Figure 1.a. Equi-partitioned Index

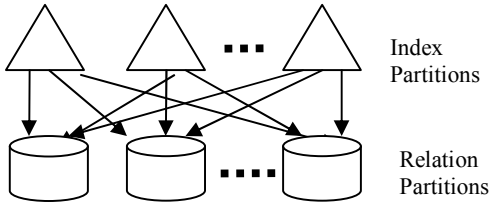

Figure 1.b. Non-Equi-partitioned Index

In the schema of Figure 2, for clarity we show only the key attributes. Each relation, however, contains many other attributes (some of them more than 100 attributes.) We populated the relations with test data. The number of rows and storage size of the largest three relations are shown below.

Relation name	**Num_of_Rows**	**Storage_Size**
Account_holdings	510 Million	114 GB
Account_Holding_Detail	260 Million	12.6 GB
Account_History	127 Million	4.1 GB

The above three relations were partitioned because of their large sizes. We created seven test queries against this database and ran the queries several times before partitioning was applied, then took the averages of the results. Next, we partitioned the largest three relations listed above and ran the same seven queries several times and averaged their results. Section 4 of this paper shows a comparative analysis between the two sets of results.

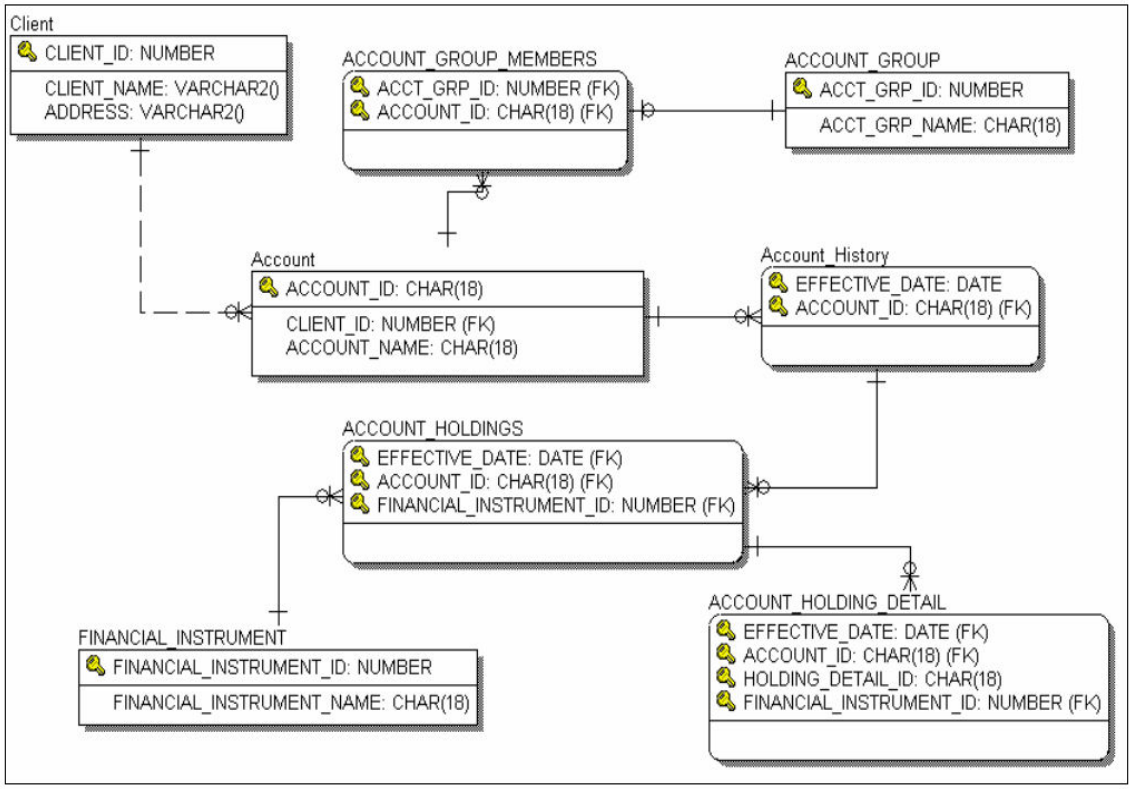

Figure 2. Schema Used in Performance Analysis

The partitioning strategy that was employed for each of these relations is as follows. Account_History was range-partitioned on Effective_Date as the partitioning attribute. Mixed partitioning was applied to the other two relations, where they were both range-partitioned on Effective_Date, then hash-sub-partitioned on Account_ID. The reason for not sub-partitioning the Account_History relation was that it was relatively small compared to the other two relations. Those relations have several indexes. We equi-partitioned the indexes with their underlying relations wherever possible.

The seven test queries we created were named Query_1 through Query_7. The first 5 queries (Query_1 through Query_5) joined the following relations and selected attributes from them: ACCOUNT, ACCT_GRP, ACCOUNT_GROUP_ MEMBERS, ACCOUNT_HISTORY, ACCOUNT_ HOLDINGS, and CLIENT. The join conditions were based on foreign keys.

The five queries differ only in the date range and number of accounts used in WHERE conditions in each query. The following shows the date range of each query and the number of accounts used.

Qeury	**Data Range**	**No. of Accounts**
Query_1	1-day	50
Query_2	30-day	50
Qeury_3	90-day	50
Query_4	1-day	500
Qeury_5	30-day	500

Qeury_6 and Query_7 join the relation ACCOUNT_HOLDING_DETAIL in addition to the above relations used in the first five queries and selects additional attributes from this relation. Here are the date ranges and number of accounts used in these two queries.

Qeury	**Data Range**	**No of Accounts**
Query_6	30-day	50
Query_7	30-day	500

In other words, Query_6 is similar to Query_2 with respect to the date range and number of accounts restriction conditions, while Query_7 is similar to Qeury_5. The purpose of adding ACCOUNT_HOLDING_DETAIL in Query_6 and Query_7 is to further assess the impact of partition-wise-joins, which is an optimization technique described in section 2 of this paper.

4. PERFORMANCE ANALYSIS

The approach we followed to conduct these tests was as follows. First, we executed each of the seven queries five times before partitioning the tables. Each time we took a snapshot of Oracle statistics pertaining to four criteria, namely, Elapsed Time, CPU Time, Disk Reads, and Buffer Gets. In Oracle terms, Disk Reads represent

physical reads while Buffer Gets represent logical reads. Some of the logical reads become physical reads if the block sought is not found in memory. Therefore, the number of physical reads is always a subset of logical reads. Elapsed Time is the total time spent before Oracle returned the query result; while CPU Time is the duration of time the CPU was actually busy processing the query. After executing each of the queries five times, we averaged, for each query, the results of each one of the four criteria. Next, we partitioned the largest three relations as described in section 3 of this paper and re-executed the same queries five times each. We took the averages of statistics for each one of the four criteria. Note that every time a query was re-run, we varied the 'begin' and 'end' date criteria and/or the specific accounts used in the query in order to minimize the impact of caching.

Below we show the results pertaining to each one of the four criteria and provide an analysis describing these results. Each one of the diagrams described below shows the average of the results before and after partitioning for each of the seven queries.

4.1 ELAPSED TIME

This measures the total time in seconds that Oracle took before delivering the result of the query. As Figure 4 shows, the elapsed time was substantially reduced after the relations were partitioned.

We notice that the CPU time of Query_6 is smaller than that of Query_5. This is because Query_5 returns data pertaining to 500 accounts, while Query_6 returns data pertaining to 50 accounts only. However, when comparing Query_5 and Query_7 (both of which have the same date range and number of accounts) we notice that the elapsed time in the case of no partitioning is larger in Query_7 than Query_5, because Query_7 joins one additional table. However, the elapsed time of both queries in the partitioning case is almost the same. This is because Oracle uses partition-wise joins which limits the search space of the join condition to only few partitions regardless of the size of the underlying tables.

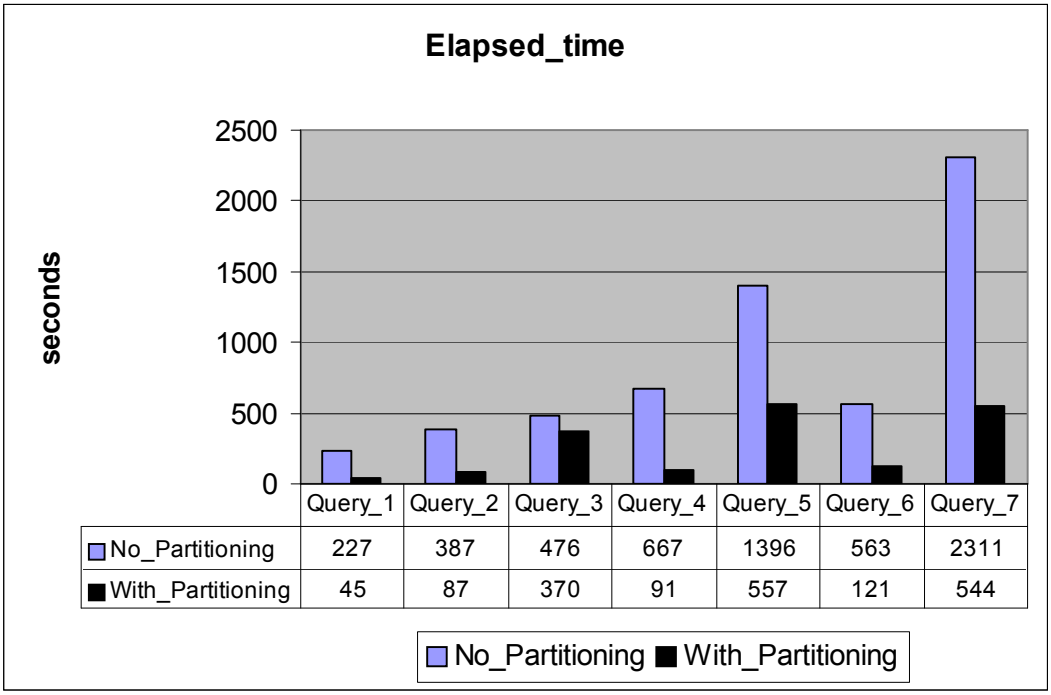

Figure 4 - Elapsed Time

4.2 CPU TIME

The figure below depicts the CPU time for the partitioning and no partitioning cases. Partitioning has resulted in a sharp decline of CPU time for all queries.

Because of partitioning, the amount of data that has to be read from disk into memory is smaller. Therefore the amount of data that the CPU has to process is smaller, resulting in smaller CPU time.

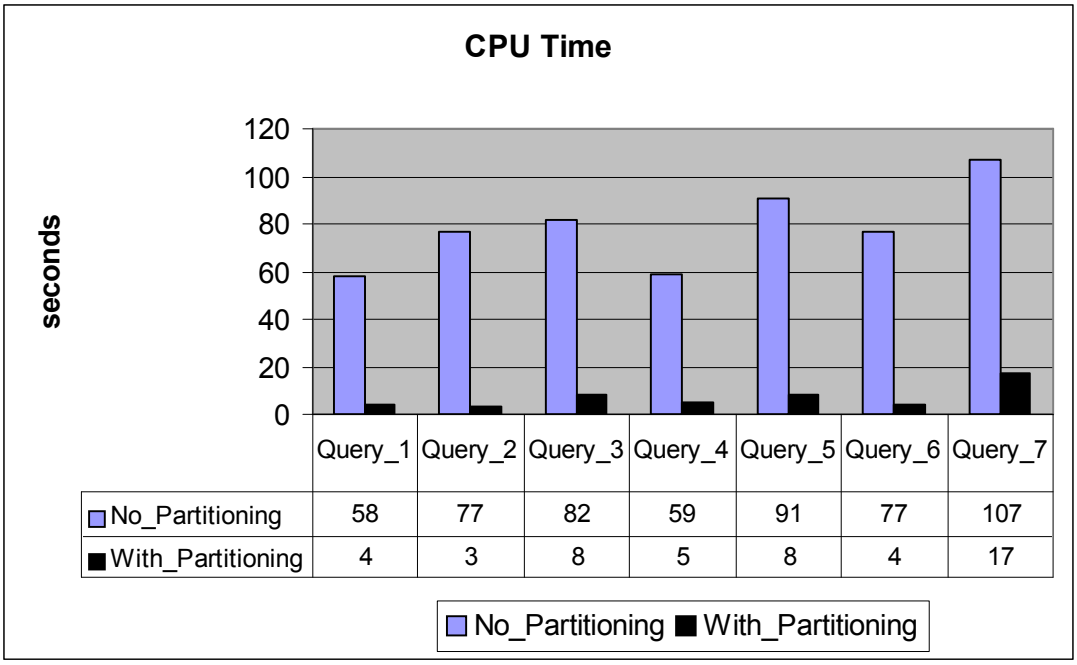

Figure 5 – CPU Time

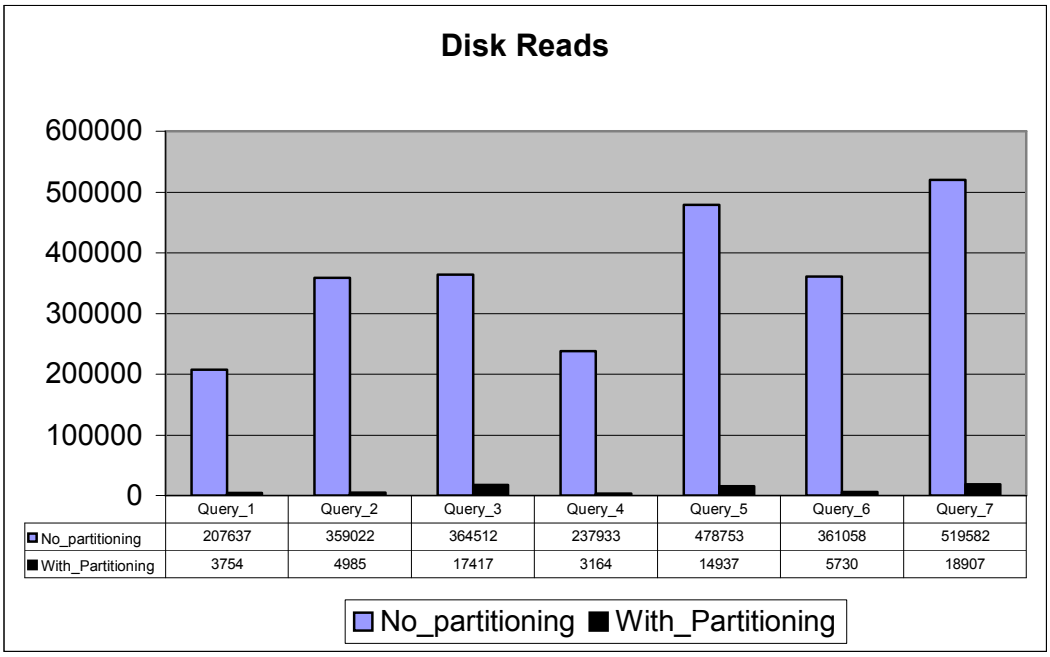

Figure 6. Disk Reads

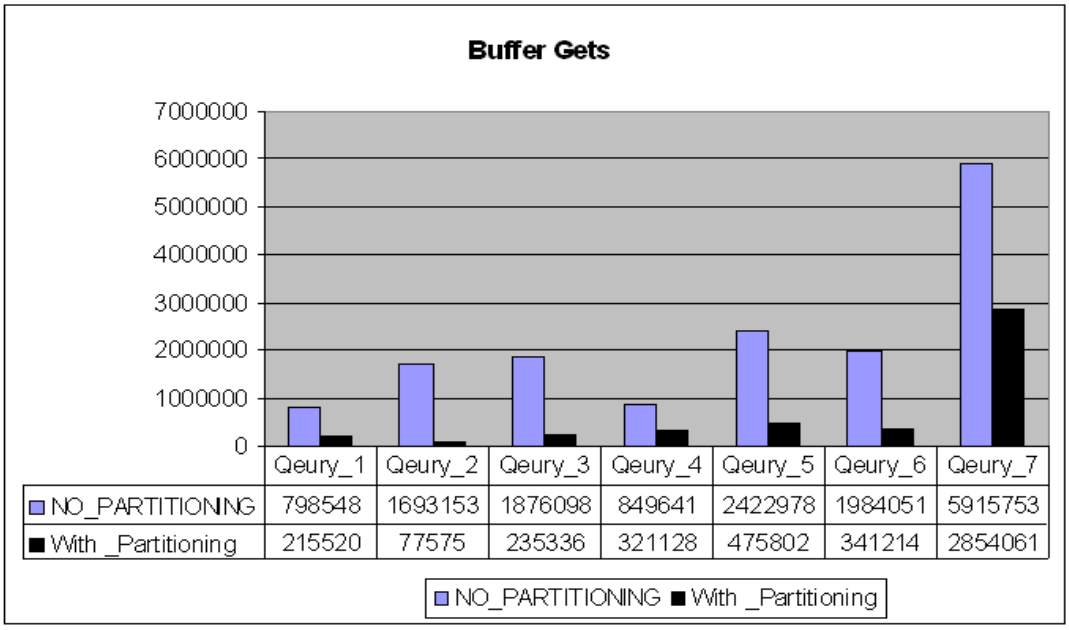

Figure 7. Buffer Gets

4.3 DISK READS

This is the criterion that shows the biggest difference between the partitioning and no partitioning cases. Partitioning causes search to be narrowed down to the specific partitions that contain the data. Oracle preprocesses the query and based on the query's restriction conditions, it creates an execution plan that involves only the needed partitions.

This resulted in a substantial I/O improvement as shown in the figure and alleviated the pressure on the I/O subsystem. In addition, since the amount of data brought to memory is smaller in case of partitioned tables, there is a better chance that this data is cached in memory for a longer period before it is aged-out. This improves performance of other queries that may need the same data since they can find this data via logical reads as opposed to expensive physical reads.

4.4 BUFFER GETS

Buffer Gets represent logical reads, that is reads that were satisfied from memory without having to go to disk. Again, by partitioning the large relations, logical reads have been reduced for every one of the test queries as shown in Figure 7.

5. CONCLUSION

In this paper, we provided a brief classification of partitioning strategies as applied to relational database systems. Both relations and indexes can be partitioned. Partitioning can be horizontal or vertical. Horizontal partitioning can be further classified into range partitioning and hash partitioning. A combination of range and hash partitioning can also be applied, which is referred to as mixed partitioning. Performance gain is achieved because the query optimizer component of a database management system has knowledge of the partitioning criteria applied to each table or index. Therefore, it can use that information to perform partition elimination and partition-wise joins, which are two powerful optimization techniques.

We demonstrated the powerful and positive impact that partitioning has on database performance by conducting a comparison analysis on a relatively large prototype database that was implemented in Oracle. The database was populated with large quantities of data that simulated multi-year financial data entries. Next, we designed seven queries that covered a wide range of restriction conditions and joins. Those queries were executed before partitioning was applied and then after partitioning was applied. Then we compared the 'before' and 'after' results along four dimensions, namely, elapsed time, CPU time, buffer gets (logical reads), and physical reads. The analysis demonstrated that partitioning resulted in a substantial improvement in overall query performance in all four dimensions for every one of those queries.

ACKNOWLEDGMENT

Publishing this research has been supported by the Deanship of Research and Graduate Studies at Applied Science University in Amman, Jordan.

REFERENCES

[1] S. Agrawal, S. Chaudhuri, V.R. Narasayya, "Automated selection of materialized views and indexes in SQL databases", *in Proc. 26th Int. Conf. on Very Large Data Bases (VLDB)*, pp. 496-505, 2000.

[2] E. Baralis, S. Paraboschi, and E. Teniente, "Materialized view selection in a multidimensional database," *in Proc. 23rd Int. Conf. on Very Large Data Base (VLDB)*, pp. 156-165, 1997.

[3] L. Bellatreche, K. Karlapalem, and Q. Li, "Evaluation of indexing materialized views in data warehousing environments", *in Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DAWAK)*, pp. 57-66, 2000.

[4] L. Bellatreche, K. Karlapalem, M. Schneider and M. Mohania, "What can partitioning do for your data warehouses and data marts", *in Proc. Int. Database Engineering and Application Symposium (IDEAS)*, pp. 437-445, September 2000.

[5] L. Bellatreche, M. Schneider, M. Mohania, and B. Bhargava, "Partjoin : an efficient storage and query execution design strategy for data warehousing", *Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DAWAK),* pp. 296-306, 2002.

[6] L. Bellatreche, M. Schneider, H. Lorinquer, and M. Mohania. "Bringing together partitioning, materialized views and indexes to optimize performance of relational data warehouses." *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2004)*, pages 15–25, September 2004.

[7] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis. "A genetic algorithm for database query optimization." *in Proceedings of the 4th International Conference on Genetic Algorithms*, pages 400–407, July 1991.

[8] S. Chaudhuri and V. Narasayya., "An efficient cost-driven index selection tool for Microsoft sql server", *in Proc. Int. Conf. on Very Large Databases (VLDB),* 1997, pp. 146-155.

[9] C. Chee-Yong, "Indexing techniques in decision support Systems", *Ph.D. Thesis*, University of Wisconsin, Madison, 1999.

[10 H. Gupta et al., "Index selection for olap," *in Proc. Int. Conf. on Data Engineering (ICDE),* pp. 208-219, 1997.

[11] H. Gupta and I. S. Mumick, "Selection of views to materialize under a maintenance cost constraint," in Proc. 8th Int. Conf. on Database Theory (ICDT), pp. 453-470, 1999.

[12] A. Gounaris, R. Sakellariou, N. Paton, and A. Fernandes. "Resource scheduling for parallel query processing on grids."*5th IEEE/ACM International Workshop on Grid Computing (GRID 2004),* Pages 396-401, November 8, 2004, Pittsburgh, USA.

[13] T. Loukopoulos and I. Ahmad. "Static and adaptive distributed data replication using genetic algorithms." *in Journal of Parallel and Distributed Computing,* 64(11):1270–1285, November 2004.

[14] Nicola, M. ''Storage Layout and I/O Performance Tuning for IBM Red Brick Data Warehouse'', *IBM DB2 Developer Domain*, Informix Zone, October 2002.

[15] P. O'Neil and D. Quass., "Improved query performance with variant indexes", *in Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 38-49, 1997.

[16] A. Sanjay, G. Surajit, and V. R. Narasayya, "Automated selection of materialized views and indexes in microsoft sql server", *in Proc. Int. Conf. on Very Large Databases (VLDB),* pp. 496-505, September 2000.

[17] A. Sanjay, V. R. Narasayya, and B. Yang. "Integrating vertical and horizontal partitioning into automated physical database design." *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pages 359–370, June 2004.

[18] Vijayshankar Raman, Wei Han, and Inderpal, "Narang Parallel Querying with Non-Dedicated Computers", *Proceedings of the 31st international conference on Very Large Databases.* Trondheim, Norway from August 30 to September 2, 2005. pages 61-72.