SQL Performance Tuning

CSCI-54100 Course Project Report – Fall 2016

Team:
Adithya Morampudi
Akhil Nayabu
Raghavendran Vijayan

Indiana University Purdue University, Indianapolis

Abstract:

SQL tuning is the iterative process of improving SQL statement performance to meet specific, measurable, and achievable goals. In other words, it is the process of ensuring that the SQL statements that an application will issue will run in the fastest possible time.

The aim of our project is to reduce execution time of some queries which plays around huge volumes of data. We want to show considerable reduction in query running time when compared with old and normal projection. In this task, we will identify high-load SQL queries and analyse its root cause and measures to improve its performance.

1. Introduction:

SQL Tuning is the process of improving the performance of SQL by using certain techniques, SQL statements are used to retrieve data from a relational database, these queries can be written in different ways which retrieves the same data. Here in our project we are trying to find out the queries which yield optimum result cost when compared queries which are not written in an optimal way. In current scenario quires for insertions, updates, retrievals are used very frequently, a hundred thousand per second even for a small application hence a large amount of data is being stored into databases every single day, making the process of data retrieval more and more complex when we use the same old SQL statements to retrieve data. If the data which we are working on is huge then a small change in the query can have a high impact on the performance of the database. Data in a relational database is stored in secondary disks and secondary storage can considerably reduce the performance resulting in poor performance of the database.

Tuning can be done in several possible ways, we can reduce full table scans to help the retrieval process efficient, full table scans can be reduced if we use proper WHERE clauses, minimizing the number of sub queries in the main query, if we have more number of sub queries the worse is the performance, using indexes we can improve the performance by using indexes on high selectivity columns such as the column which is a primary key, by avoiding the use of temporary variables, the use of temporary variables increases the complexity of the query, Partitioning a table can heavily impact the performance of a SQL query when the partition is done on a large database which is greater than 2Gb, as partition makes the data retrieval process a lot easier.

2.Necessity of Tuning

As mentioned earlier, we have huge chunks of data which is not stored in a sorted order, and if we want to retrieve a single tuple from the database, it may take a huge amount of time to achieve this simple task, which is not desirable. In order to overcome this we need SQL tuning, when properly implemented can reduce the data retrieval time significantly. By tuning we can reduce the time spent by end user on the system and we can reduce the amount of resources used to do the same work. We can achieve this objective in many ways, few of them are Reducing the workload, balancing the workload, parallelize the workload.

We can change the execution plan of the statement without changing the overall functioning of the query which will reduce the workload and thereby consuming less resources. Consider a commonly executed query which needs to be executed several times in a given amount of time, if we are able to avoid full table scans in this type of query and implement proper indexes and search keys, then we can reduce the amount of resources used. Balancing the workload, to achieve a balance workload on the resources we need to find the time where peak resources are used and at that time we shouldn't schedule not critical reports and batch jobs which will help in reducing the workload.

3. Tuning Approaches

We can tune an SQL statement by following some standard procedures which will in turn increase the performance and reduces the time required to do the same work, with not optimized query.

General procedures which will increase the efficiency of a SQL query:
-Applying Partitioning Concepts
- Indexing
- avoiding usage of triggers as it utilizes more resources.
- use of better joins as to get minimal number of rows than a normal join.

3.1. Partitioning

We can achieve enhanced performance, manageability by using partitions. Partition pruning is one such technique which is being used widely, consider for example a table which is being used to keep the information about Orders placed by customers. The data which gets added into the Orders table daily is huge as a lot of customers who will be placing orders continuously, now if we want to look up an order which has been placed somewhere around 3-4weeks back then if we use the regular look up query it will scan the complete orders table starting from the current day, instead of this we can partition the orders table by weeks, and if we want to look up an order which is placed several weeks back then we can directly go to that particular partition and lookup the value, this method increases the efficiency by a huge amount, if we had two years of data in the single table and if we used partitions to save this data then we can potentially execute the query 100 times faster simply with the use of partitioning pruning.

3.1.1 Single-Level Partitioning

A single-level partitioning can be implemented by using one of following methodologies:

- Range Partitioning
- Hash Partitioning
- List Partitioning

Range Partitioning

Range partitioning is done on range of values of partitioning key. Each partition has a clause less than value, example less than 10 and less than 100 then partition is done for all values below 10 and for values between 10 and 100.

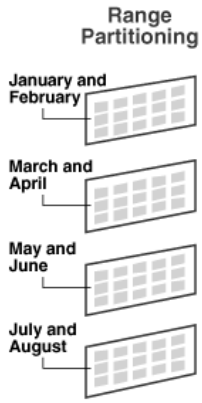


Fig 1: Range Partitioning

List Partitioning:

In list partitioning you can group and order unrelated and unordered rows of data together. When we partition we specify a list of values for partition key in description of each partition. It is generally used for clustering similar data together.

Fig 2: List Partitioning

Hash Partitioning

Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size.

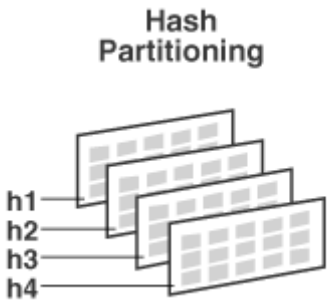
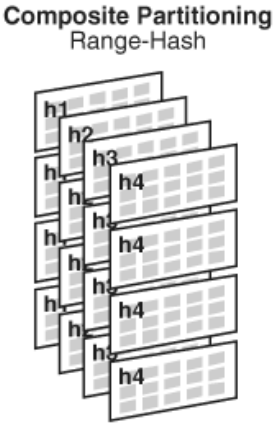


Fig 3: Hash Partitioning

3.1.2 Composite Partitioning

It is combination of any two single-level partitioning methodologies. Initially the table is partitioned by any one of the data distribution method and then each partition is subdivided into sub-partitions using any of data distribution again.



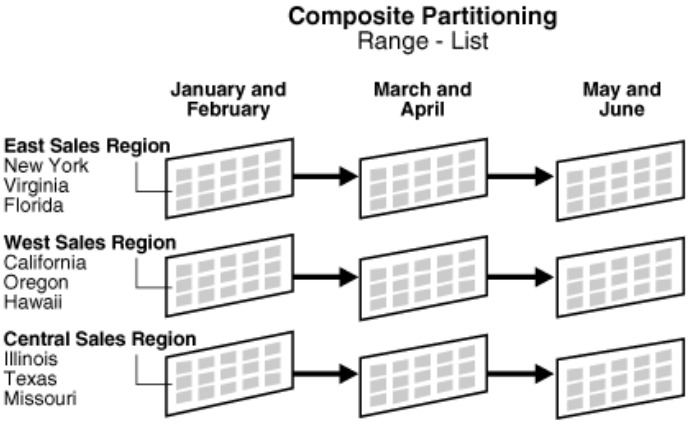


Fig 4: Composite Partitioning

3.2 Indexing

Indexes are data structures which store the values of a column in a table. Indexes help us to speed up the search queries by cutting down the number of records in a table to be that needs to be examined.

Creating an Index in SQL:
CREATE INDEX name_index
ON table_name (column_name);

Creating an Index on multiple columns in SQL:
CREATE INDEX name_index
ON table_name (column_2ame1, column_name2)

Data Structures that can be used for Indexes:

- B-Trees
- Hash Tables

3.2.1 B-Tress

B-Tress are most commonly used data structures for indexing as they are time-efficient. The data which is stored in B-Tress can be sorted which in turn reduces the time required for look-up.

3.2.2 Hash Tables

Hash tables are most effective where we try to retrieve data by using string equalities or comparisons. Hash tables are useless when in certain scenarios like when we want to find out students who have a grade less than 3.5. This can't be indexed this they only suitable for key-value pairs.

3.2.3 How Indexes work?

So how will we retrieve other values in same row of the index found? Index stores pointer values to storage on disk where the corresponding row values are stored. So, an index will basically look like (value, pointer).

Ex: ("group12",0x008c67)

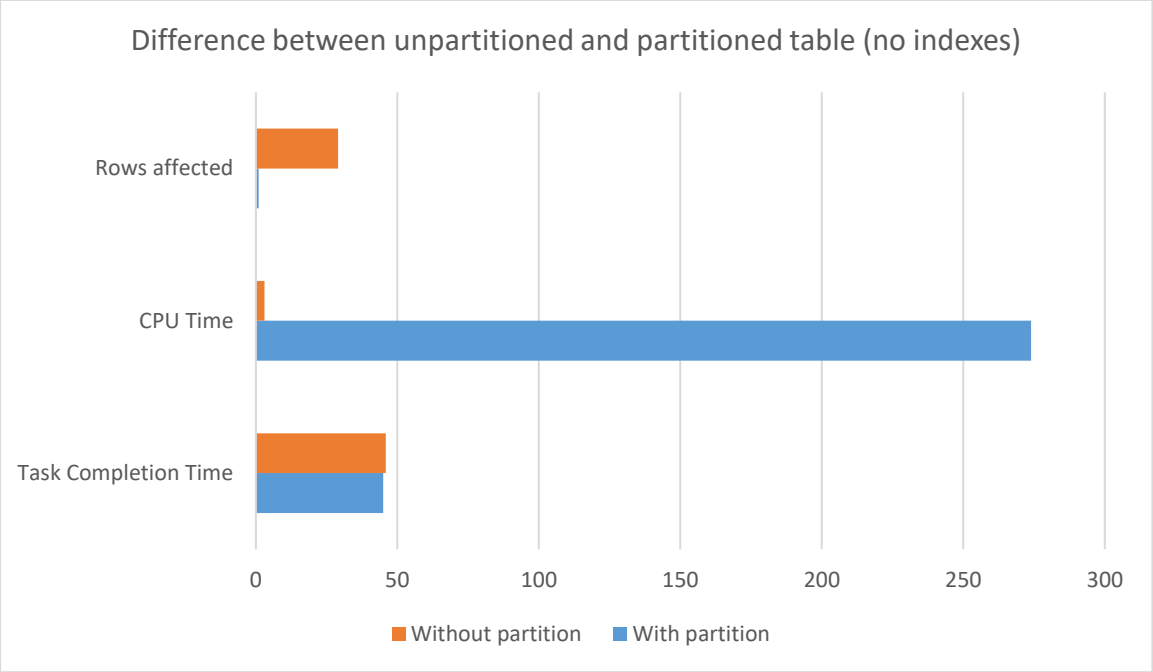
Where:

- group12 is value.
- 0x008c67 is the pointer pointing to memory where corresponding row value are stored in the table.

4. Performance Improvement

Comparison between unpartitioned and partitioned tables without index

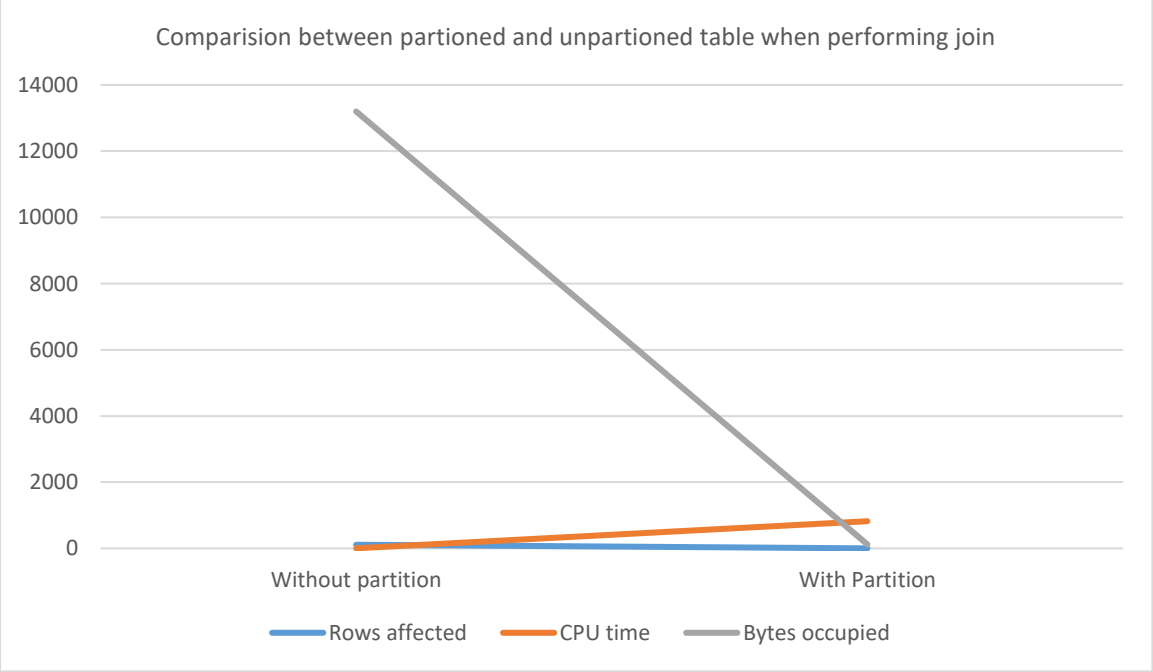
For the query to return rows which have C values less than 20



Analysis:

1. Since we have partitioned a table with very small data, there is a degradation in performance which is clearly plotted in CPU time difference.
2. But, because of the partitions, there was no full scan performed on the table. It helped in searching only in the partitions which are fit

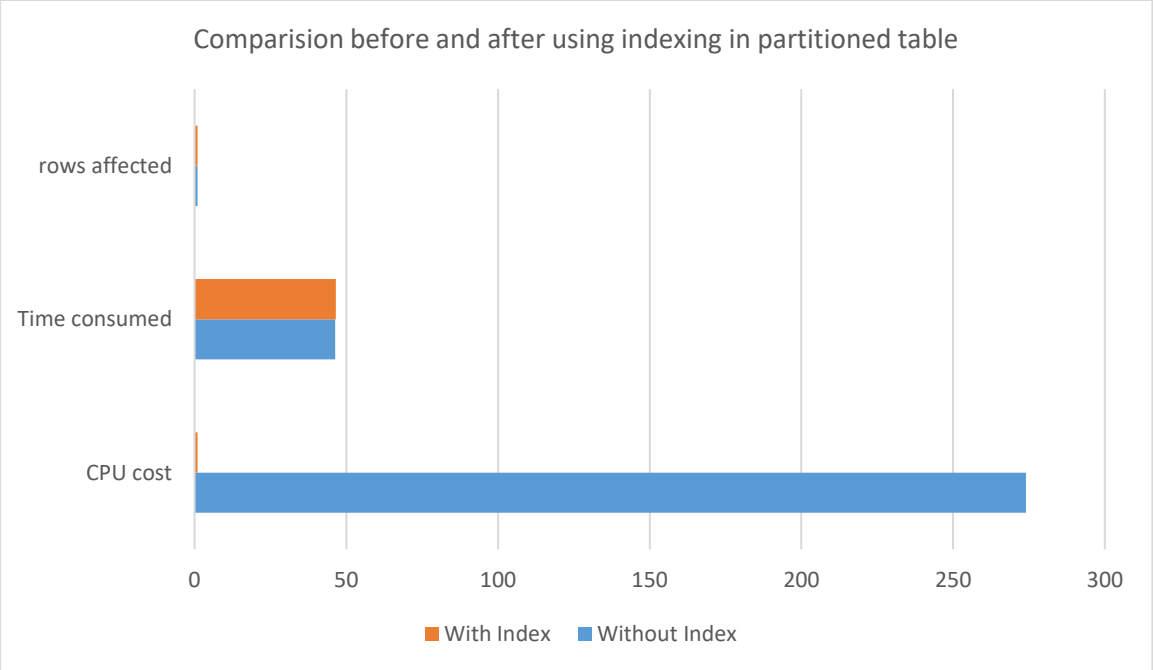
When performing Join operation



Analysis:

1. When an inner join is performed between these two tables, we can able to observe the table without partition have traversed through all the records in the table, whereas the other one has traversed only in the matching partitions.
2. There is a drastic difference in the bytes consumed because there is a need to scan the entire unpartitioned table.
3. The number of rows difference is as same as that of previous case

After Creating index on partitioned table



Analysis:

1. We decided the attribute, that is frequently invoked in the where clause and created index on it.
2. Thus, there is a considerable drop in the CPU cost before and after started using indexes. This is because, the index aid in finding the specific rows faster than the full scan either within the whole table or within the partition.

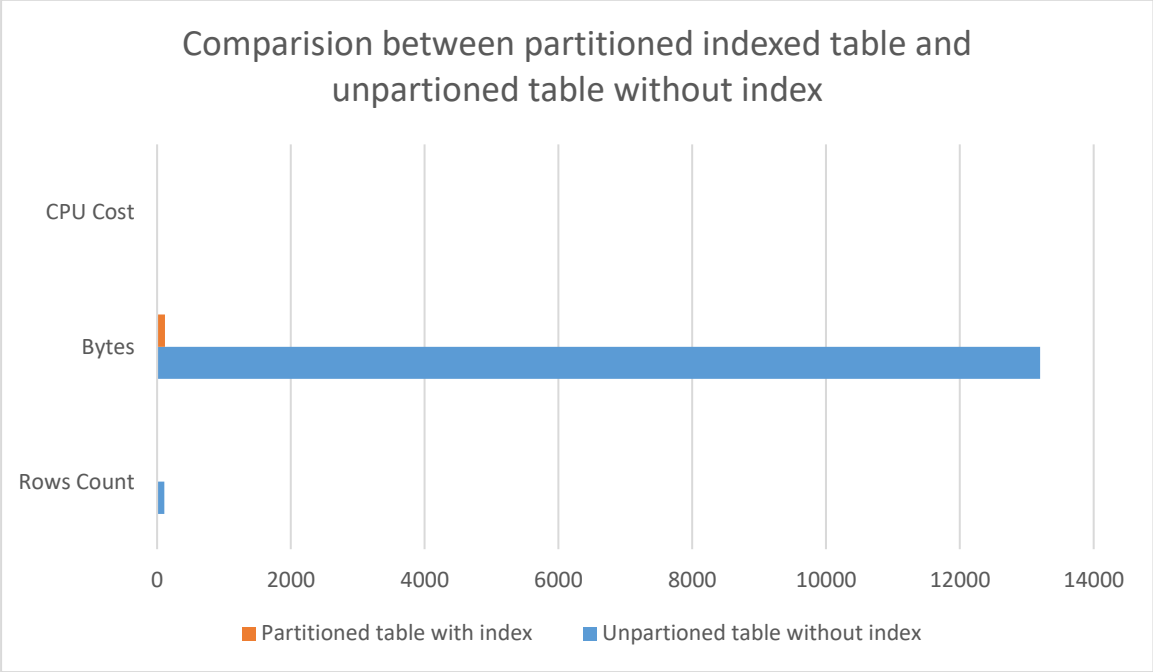
Performing inner join on a table with index and table without index

Here we choose a table with no index, unpartitioned and another table with index and is portioned.

When performing join, we can get better performance than the normal join, that is performed in the second case study.

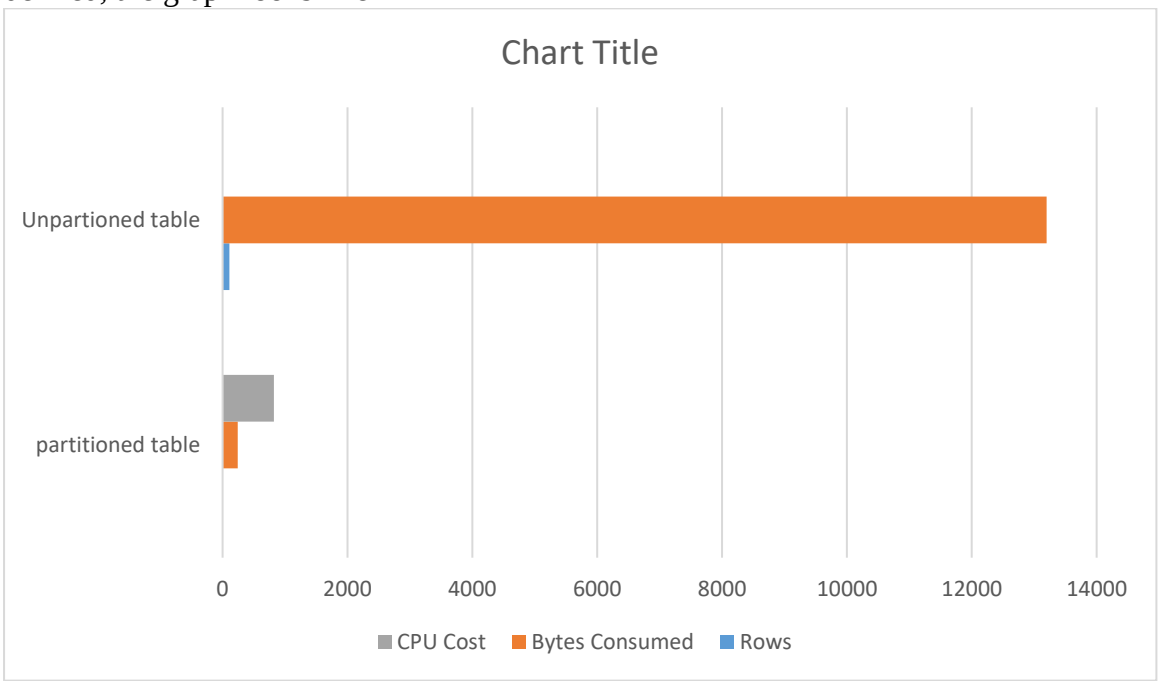
The presence in the differences between them are because of the indexes. They speed up the scan operation and returns the output back.

The following graph will explain the difference in detail.



Performing other queries

When performing join between an unpartitioned table and partitioned table, both with indexes defined, the graph looks like



Analysis:

Since the portioned table has index defined on it, the number of rows it fetches is low. But still it need to traverse all the way and it being a small database, the CPU cost proves to be very high. On the other hand, in the unpartitioned table, the presence of index helps in reducing the records to be fetched.

5. Challenges Faced and Future Enhancements

Challenges faced

1. We thought of using Result cache to store the intermediate results. But since, we are not repeating the same set of operations very often, we dropped the plan of using it.
2. Since we don't have sufficient privileges in our SQL developer account, we cannot gather the exact statistics, cannot insert bulk data, which is the trivial to analyse proper tuning.
3. Since we are taught in theory about some of these concepts, it took time for us to understand them and implement in practical.

Things went well

1. Able to understand the importance of writing a proper query before starting tuning.
2. Several concepts are new to us and we understood those and able to visualize which is the best method to start the tuning.
3. Within our team, several soft skills were also learnt of this project. We can convince each other the ideas/solution we have in mind.

Things that didn't go well as expected

1. Due to shortage of time, we were not able a full-fledged application.
2. Some restrictions associated with our credentials in oracle SQL developer stopped us in trying some other things. For example, due to space limitation on every user, we were not able to create a new tablespace, we cannot work on bulk data and so.

Things in mind (to work in future)

1. We must understand the practical difficulties associated with queries other than straight away diving into tuning.
2. We want to work on a real-time system's database i.e. a pizza shop's database or stock market database and must analyse the tuning measures they perform over there and we must contribute, if there is any room for improvement.

6. Individual Contribution

Since this is not a typical database management system project, we cannot possible to split our works individually. We collectively put our ideas and thoughts to achieve the outcome.

Mainly focus Areas

Member	Area of Work
Akhil Nayabu	Analysed types of Partitioning options
Adithya Morampudi	Analysed different types of Indexing
Raghavendran Vijayan	Implementation of partitioning/indexing.

7. Conclusion

There are many ways to write the query, all of those might fetch you the same results but it only one of them might be optimal. So, it not always about writing query, it is about how optimal the query is especially when we are working databases which have large amount of data where retrieving one value might take hours if query or statements we used are not proper. It is always good to use tuning techniques right from the beginning because it is not feasible to tune later when we have large data inserted in. It is always good to plan on what should we use so that retrieval process will not be effected in future even if large amount of data is inserted in. Though this project we have gone through many tuning concepts which we were not aware of, and we also got to know how important it is to use proper statements from the beginning.

8. References

Database Performance Tuning Guide

- https://docs.oracle.com/cd/B19306_01/server.102/b14211/sql_1016.htm#g42927

Database SQL Tuning Guide

- https://docs.oracle.com/database/121/TGSQL/toc.htm

SQL Database Tuning

- https://www.tutorialspoint.com/sql/sql-database-tuning.htm

Optimizing and Optimizer: Essential SQL Tuning Tips and Techniques, An oracle White paper, September 2005.