Daniel Kidane

Oracle Tables Partitioning

Database performance using table partitioning

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Software

25.11.2015

Metropolia

Author(s) Title Number of Pages Date	Daniel Kidane Oracle Tables Partitioning 38 pages + 3 appendices Wednesday 25th November, 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software
Instructor(s)	Patrick Ausderau, Research Engineer (M.Sc.)

The purpose of this thesis is to evaluate how Oracle table partitioning affects the performance of a database. In this thesis, the partitioning techniques are applied to a table varying in size from 30,000 to 18 million rows.

Different techniques and applications are used to support this thesis. On the client side a personal computer with an Oracle database version 12c is installed. At the back end, a free version of Oracle database is used to send data back to the client side whenever requested.
In order to access the database, the application used is Toad, which stands for Tools for Oracle Application Development. Toad is used for application development, database development, measuring database performance and other purposes.

As a result, the tests conducted in this thesis show that there is not any performance difference when the partitioning techniques are applied or not on tables with less than 300,000 (22.9 MB) records.However, a clear gain in performance is observed when the number of rows increases to 3 million (323 MB) worth of data and beyond for partitioned tables compared to non-partitioned ones.

In conclusion, it is not necessary to partition a table if its size is below 22.9 MB. But, based on the findings in this thesis, if the size of the table grows up to 323 MB and beyond, it is recommended to partition the table for manageability, availability and performance purposes.

Keywords	Oracle Database, PL/SQL Scripts, SQL Queries, Tables Partitioning for Performance, Tables Partitioning for Availability, Tables Partitioning for Manageability

Metropolia

Contents

1 Introduction 1

2 Partitioning Fundamentals 2

2.1 Advantages of Partitioning 2
2.2 Disadvantages of Partitioning 3
2.3 Partitioning Concepts 3
2.4 Partitioning Indexes 4
2.4.1 Local Indexes 5
2.4.2 Global Indexes 6
2.5 Partitioning for Performance 7
2.5.1 Online Transaction Processing (OLTP) Systems 8
2.5.2 Data Warehouse Systems 9
2.6 Partitioning for Manageability 9
2.7 Partitioning for Availability 11

3 Oracle Database Partitioning Strategies 11

3.1 Single-Level Partitioning 12
3.1.1 Range Partitioning 12
3.1.2 Hash Partitioning 18
3.1.3 List Partitioning 23
3.2 Composite Partitioning 28
3.2.1 Composite Range-Range Partitioning 28
3.2.2 Composite Range-Hash Partitioning 31

4 Methods and Materials 32

5 Results and Discussions 33

5.1 Results and Discussions on Range Partitioning 33
5.2 Results and Discussions on Hash Partitioning 34
5.3 Results and Discussions on List Partitioning 35

6 Conclusion 36

References 37

Appendices

Appendix 1 Complete Source Codes for Range Partitioning Tables
Appendix 2 Complete Source Codes for Hash Partitioning Tables
Appendix 3 Complete Source Codes for List Partitioning Tables

Abbreviations

DBA Database Administrator.
DML Data Manipulation Language.

I/O Input/Output.
ILM Information life cycle management.

OLTP Online Transaction Processing.

PL/SQL Procedural Language/Structured Query Language.

RAM Random Access Memory.

SQL Structured Query Language.

Glossary

ad-hock query a query that cannot be determined prior to the moment the query is issued. It is created in order to get information when need arises and it consists of dynamically constructed SQL which is usually constructed by desktop-resident query tools. This is in contrast to any query which is predefined and performed routinely like a procedure, function or trigger.

contention a scenario in Oracle databases where multiple processes try to access the same resource simultaneously. Some processes must then wait for access to various database structures.

full table scan (also known as sequential scan) a scan made on a database where each row of the table under scan is read in a sequential (serial) order and the columns encountered are checked for the validity of a condition.

partition key a column or set of columns from the same table whose consolidated value decide the partition for a given data.

tablespace a container for segments (tables, indexes, etc).

1 Introduction

When a database table grows in size tremendously like in hundreds of gigabytes or even more, it can become very difficult to load new data, remove old data, and maintain indexes. Just the big size of the table causes such operations to take much longer. Even the data that must be loaded or removed can be very sizable, making Data Manipulation Language (DML) operations like INSERT and DELETE on the table difficult. Maintenance of large tables and indexes can become very time-and resource-consuming. At the same time, data access performance can reduce for these objects.

But are there any techniques that will solve this problem and improve the performance of the database? If there are any, is it possible to apply these techniques in both large and small data? If these partitioning techniques are applied to both a small table and a large table, which one of these will benefit from the techniques performance-wise?

A partition is a division of tables, indexes, and index-organized tables into distinct independent parts. Database partitioning is normally done for manageability, performance or availability reasons. [1]

Partitioning a large table divides the table and its indexes into smaller partitions, so that maintenance operations can be applied on a partition-by-partition basis, rather than on the entire table. In addition, the optimizer can direct and properly filter queries to appropriate partitions rather than the entire table.

The current version of Oracle database[1] can be accessed through the Internet from a cloud database service provider to be delivered on demand. In other words, a cloud database is designed for a virtualized computer environment. A cloud database is implemented by using cloud computing that means utilizing the software and hardware resources of the cloud computing service provider. Cloud computing is getting attention and is growing in the IT industry around the world. Nowadays, many companies have started moving towards cloud computing and accessing their data from a cloud database. [2]

[1] At the time of writing, Oracle database version 12c

This thesis documents the different techniques of Oracle partitioning for performance, manageability, and availability of data in an Oracle database [2]. The different partitioning techniques are applied starting from a very small table to a relatively large table with millions of records on it. In this thesis the advantages and disadvantages of Oracle partitioning are also discussed.

2 Partitioning Fundamentals

2.1 Advantages of Partitioning

Partitioning may help operations like data loads, index creation and rebuilding, backup and recovery which are considered as data management operations. These operations may be possible to implement at the granularity of the partition level rather than at the entire table level.

Partitioning may help boost query performance. Often it is queried a subset of partitions rather than the entire table and this may provide a reasonable amount of performance gain which is also known as partition pruning. Because it is accessed a portion of the whole data, which is the piece of the table that contains the data needed by the query, partition pruning can also reduce the amount of data retrieved from a disk and reduce processing time, especially for Input/Output (I/O)-bound databases with small Random Access Memory (RAM) data buffers.

Partitioning can reduce the impact of the scheduled downtime for maintenance operations. Individual partitions are independent of other partitions in the same table when it comes to maintenance. Due to this reason, maintenance of individual partitions or groups of partitions on the same table or index can be done without affecting one another. It is also possible to run concurrent SELECT and DML operations against partitions that are unaffected by maintenance operations.

Partitioning can be implemented without requiring any modifications to applications. For example, a non-partitioned table can be converted to a partitioned table without needing to modify any of the SELECT statements or DML statements which access that table. It

[2] see footnote 1

is not necessary to rewrite the application code to take advantage of partitioning. [3]

2.2 Disadvantages of Partitioning

Experience has indicated that Oracle partitioning should be applied to databases having 500 gigabytes of data or more.[4] Most articles and papers on Oracle say that Oracle partitioning has many advantages. However, there are some drawbacks that do not seem to get much press. Certain conditions highlight specific circumstances in which it is not always the best option especially on small databases. Partitioning is mainly targeted toward large database environments. Indexes can pose a serious problem if they fail. If an index goes down or is damaged, the underlying tables can be damaged beyond repair, making it unusable and unrecoverable. The only solution is to revert to backup if there is one or to rebuild the index. In a partition context, this can take a very long time.[5]

Partitions are much harder to manage than standard tables devoid of them. This is because the partition aspects have to be identified and managed as part of an operation, such as the use of "`truncate`". It is not enough to use the simple truncate command; the correct command would be "`alter table truncate partition`". Therefore, partitions are harder to manage.

2.3 Partitioning Concepts

When performing partitioning, tables and indexes are subdivided into smaller individual parts. Each part of the database objects which are divided is known as a partition. As an option, a single partitioned database object may have its own storage characteristics. For someone like Database Administrator (DBA) who accesses the partitioned objects, the partitions have multiple parts that can be managed either individually or as a group. This will give the DBA the flexibility and control over the partitioned objects. From the application perspective, the partitioned and non partitioned objects are considered identical. When manipulating data of the partitioned table using Structured Query Language (SQL) DML commands, no changes are necessary.

When partitioning database objects, a set of columns with a partition key determines on

which partitions a given row should be placed. If there are composite partitions in the table, the partition is further divided into sub-partitions using a second set of columns in which the data placement of a given row is determined by both partitioning key criteria and placed on the appropriate sub-partition.

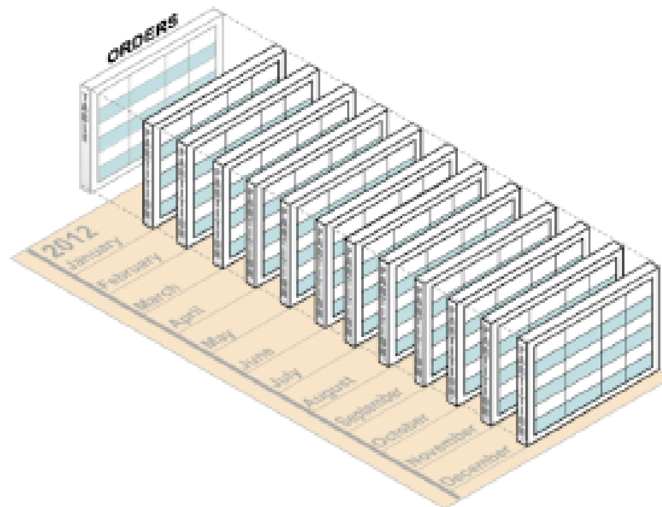


Figure 1: Application and DBA view of a partitioned table (Copied from Oracle corporation (2014) [6])

In figure 1, the table ORDERS is partitioned in one of the partitioning techniques called range-partition. The table is partitioned using a monthly partitioning strategy in which any application will treat these partitions as single object. On the other hand, DBAs can manage and store monthly partitions separately using for example different storage tiers, applying table compression to the older data, or store complete ranges of older data in read-only tablespaces.

From the application developers' point of view, there should not be any difference whether a table is partitioned or not. But developers may benefit from it in a way that if a resource-intensive DML operation has to be purged from a table, a partition maintenance operations may be used to improve runtime while reducing the resource consumption. [6, p.2]

2.4 Partitioning Indexes

An index is a database object which is a duplicate of a very small section of a table, like a single field. An index in Oracle databases behaves like an index at the back of a book or

a table of contents at the front of a book. When looking for specific information in a book, it is easier to find the term in the index or table of contents first and then use the page reference number to find the information within the pages of the text. [7, p.65]

When the partitioned tables are used as part of a database design system strategy, most of the time partitioned indexes are used because both partitioned tables and partitioned indexes go together, though it is not mandatory to have both at the same time. Deciding on whether to have a partitioned index or not may depend mainly on different factors such as data volume, client query requirements, data purging requirements and so on. But of all the factors, the data volume plays an important role to make the decision. The reason is that if the volume of a data is increased, the speed of loading, querying and purging the data will be affected.

Basically there are two ways of partitioning an index. The first method is called local index and the second is called global index and each method has different ways of usage and implementation.

2.4.1 Local Indexes

In the local indexes method, for every partition of a table there will be a corresponding index partition which is associated with it. A single table partition will have an index partition pointing to it and all the rows in a single table partition are represented in a single index partition. For example if there is a table partitioned by a range based on dates and there is a partition for every month of the year, then for all the data for the month of March 2014 partition, there will be an index created for all the partitions.

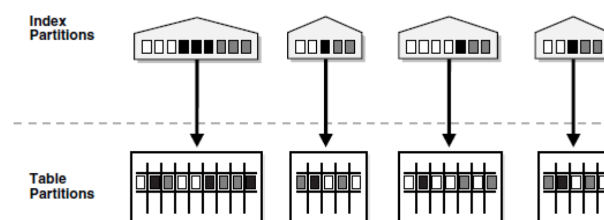


Figure 2: Locally partitioned index (Copied from Oracle® Database (2015) [8])

When creating a locally partitioned index as shown in figure 2, it may be either prefixed or non-prefixed. Prefixed locally partitioned indexes are indexes that contain keys from the

partitioning key as the leading edge of the index. For example, let us take a table called ORDERS. If the table was created and range-partitioned using its columns ORDERID and ORDERDATE, then the table will have a locally prefixed index based on ORDERID column. The partitions of the index are equipartitioned, meaning the partitions of the index are created with the same range boundaries as those of the table.

On the other hand, non-prefixed locally partitioned indexes are indexes that do not have the leading column of the partitioning key as the leading column of the index. Using the same ORDER table with the same partitioning key (ORDERID and ORDERDATE), an index on the ORDERDATE column would be a local non-prefixed index. A local non-prefixed index can be created on any column in the table, but each partition of the index only contains the keys for the corresponding partition of the table.

If a primary key constraint is needed on a table, it is a good practice to create first a unique index using the columns to be used for the primary key constraint and then add the constraint after the index has been created, for example as shown in listing 1. In this case managing the primary keys and unique indexes will be easy. Additionally, this will help to re-enable and disable the constraint without dropping the underlying index.

```
1 CREATE UNIQUE INDEX departments_pk
2 ON departments (departments_id, hire_date)
3 LOCAL;
4 alter table departments add constraint departments_pk
5 primary key (departments_id, hire_date);
```

Listing 1: Creating Unique Index

In this case managing the primary keys and unique indexes will be easy. Disabling constraints before loading bulk data into table and enabling afterwards is a common data warehouse environment practice in order to save time when processing data.

2.4.2 Global Indexes

The second method, called global index partitioning as shown in figure 3, is a method by which a single index partition may point to any or all table partitions. In other words the index created has a different partitioning schema and is partitioned based on a different

column or set of columns than the data. A globally partitioned index may be created on a partitioned or non-partitioned table. Usually, the globally partitioned indexes are not used for performance boost purposes on non-partitioned tables due to the maintenance considerations on the table. [8]

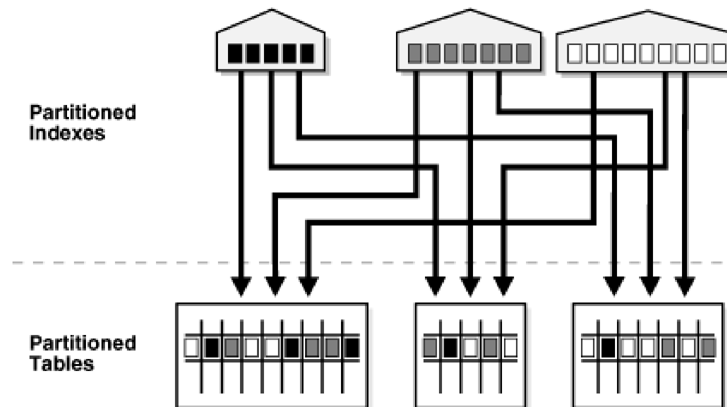


Figure 3: Globally partitioned index architecture (Copied from Oracle® Database (2015) [8])

Except for the already taken table partition key, a globally partitioned index is used for all other indexes. When a table partition is, say, dropped as part of a reorganization, the entire global index will be affected. But, when defining a globally partitioned index, the DBA has the flexibility of specifying as many partitions for the index as needed.

2.5 Partitioning for Performance

The value of a partitioning key determines where a given row is stored. The way the data of the table is placed across the partitions is registered as partitioning metadata of a table or index. For every SQL queries and partition maintenance operations performed on the table, the metadata will be used as a determining factor. In order to see what benefits are obtained from partitioning, let us consider two different categories of SQL statements. There are SQL statements such as ALTER, CREATE and DROP which define the structure of the database and modify objects like table and index. The second type of SQL statements are those which are used for read-only purposes and change data which is already created, for example INSERT, UPDATTE and SELECT.

In association with SELECT, there are two types of operations to be considered: partition elimination and parallel operation. Partition elimination is a way of disregarding some

part of data when processing a query and parallel operation is a way of performing multiple tasks concurrently for example a full table scan and index range scan concurrently. However, the benefits obtained from these operations depend on the type of system used.

2.5.1 Online Transaction Processing (OLTP) Systems

In an Online Transaction Processing (OLTP) system, partitioning should be considered with caution as this may negatively affect runtime performance. This is because the query results are expected to return virtually and instantaneously, and most of the query results from the database are expected to be through small index range scans. Due to this reason, the above mentioned benefits of partitioning in section 2.1 will have little effect on this type of system. However, partition elimination is important in a scenario where a full table scan is needed as it will, in fact, only run through some partitions and thus avoid a real full table scan.

There are also some other advantages which can be obtained from partition elimination in OLTP systems. For example, partition elimination may be used to increase concurrency by decreasing contention. If there is one table segment with one index segment and 30 table segments with 30 index segments are created for it, it will be as if there are 30 tables instead of one, hence contention will be decreased for the shared resource at the time of modification.

Parallel operations in OLTP are not recommended as these operations are used by DBAs to perform rebuilds, index creation, statistics gathering, etc. When it comes to OLTP, the queries should already be characterized by fast index accesses and be at a point where partitioning should not make a huge difference. Even though most OLTP applications do not benefit from the fact that partitioning is able to enhance query performance, administrative ease and higher availability of data are some of the advantages that this operation offers to OLTP systems.

2.5.2 Data Warehouse Systems

Partitioning is an important practice when there are millions of records to be manipulated in the data warehouse . For example if there is a large table on which an ad hoc query is being performed, the ad hoc query is implemented on a portion of the table assuming that the table is partitioned. So, logically it will be a good idea if it is possible to query the partition which has the query results. Using one of the partitioning techniques, it is possible to avoid scanning the whole table. Rather, only a portion of the big table will be scanned and as a result the performance of the system might be kept at its best.

Unlike OLTP systems, data warehouse often uses parallel query operations. In these kinds of environments, operations such as parallel index range scans or parallel fast full index scans are used frequently and will have a good chance of speeding up the processing.

2.6 Partitioning for Manageability

Data management is a key issue when dealing with big data. By partitioning tables and indexes into smaller and manageable units, DBAs can have a control over their data in terms of manageability. Oracle has a set of SQL commands for managing partitioning tables. Examples of these types of commands include adding new partition, dropping, moving, merging, truncating, and exchanging partitions. Maintenance operations with partitions are easy as they can be focused on a single partition rather than on the entire table. For example, a DBA could compress data in a single partition for the year 2014 of a table, rather than compressing the whole table.

With the new version of Oracle database[3], maintenance operations can be done in a totally online fashion. Managing databases in an online fashion is a type of databases management by which the database resources are controlled by a third party in a different location. This type of cloud database feature allows other operations like queries and DML operations to carry on while the data maintenance operation is in process [6]. A cloud database is offered by different vendors. Some of the vendors that offer cloud databases

[3] starting with version 12c

include Oracle, Microsoft, IBM and Amazon.

In Oracle database[4] partition operations, it is possible to perform maintenance operations on multiple partitions combining them as single partition making it a single atomic operation. For instance, three different partitions 'March 2014', 'April 2014', and 'May 2014' can be merged into a single partition named, 'L1 2014' with a single merge partition operation. Another usage of partitioning for data manageability is that it supports rolling window load process in data warehouse environments. Rolling window load process is a situation in a data warehouse by which data may be loaded on a daily basis and in order to manage this, it is easier to create a new range partition and load the data there than modifying the whole table as the DBA does not need to modify any other partitions [9, p.28]. Other database vendors such as Microsoft, also provides rolling window load process in their data warehouse management system.

Partitioning may also help to remove data easily without affecting other partitions. To purge data in a partitioned table, the target partition can just be dropped or truncated instead of using the operation DELETE which may use many resources and touching all the rows being deleted. When removing data with a partition maintenance operation like drop or truncate in the Oracle database[5], it is not necessary to have an immediate index maintenance to keep all indexes valid as it will be optimized by the database itself [8]. Other database vendors also provide index maintenance.

Information life cycle management (ILM) is a way of managing the flow of a company's data and related metadata from the origin and creation of the data to a point where it becomes obsolete and is deleted. Information life cycle management with partitioning is today's challenge for many companies because of the reason that they want to store their data for a low cost for the amount of time their data has to be stored before deletion. This kind of problem can be minimized using Oracle partitioning with automatic data optimization and a heat map, which is a new functionality in the advanced compression option of the Oracle database[6].

Partitioning can help individual partitions or groups of partitions to be stored on different

[4] at the time of writing, version 12C
[5] starting with version 8.0 and beyond
[6] see footnote 3

storage tiers giving different physical attributes like compression and price points. For example if there is a table containing 2 years of worth data, it is possible to store the most recent data, which is a quarter of the whole data in an expensive high-end storage tier and keep the rest of the table on an inexpensive low cost storage tier [9, p.154]. ILM is not unique to the Oracle database as other database vendors also support this feature.

2.7 Partitioning for Availability

When database objects are partitioned, they are independent of each other in terms of data maintenance, manageability and other database operations. For example, if one partition of a partitioned table is not accessible, all of the other partitions of the table will still be available. This behavior of partition independence can be an important part of a high availability strategy. The applications using the database will run and execute queries against the available partitions as long as these queries don't need to access the unavailable partitions.

By the help of partition-to-tablespace mapping, it is possible to store each partition in a different tablespace independent of the other partitions allowing DBAs to perform backup and recovery operations on an individual partition or collection of partitions. This is helpful in scenarios where a disaster or loss of data occurred in the database. It may be possible to recover the database with just the partitions available in other tablespaces which are active at the moment. The lost data from the other partitions found in different tablespaces may be recovered at any convenient time to avoid the downtime of the system.

The size of a database will not have an impact, in terms of availability, on objects which are partitioned, which means that the most relevant data may be available in a reasonably short period of time.

3 Oracle Database Partitioning Strategies

In Oracle database[7] partitioning, there are three basic data distribution methods that manage how data is placed in each partitions, namely:

[7] from version 8.0 and beyond

- Range partitioning
- Hash partitioning
- List partitioning

A table can be partitioned as either single-level partitioning or composite-partitioned using the above data distribution methods. Each of the partitioning strategies has different design methods and advantages which are more appropriate for a particular partitioning [8].

3.1 Single-Level Partitioning

Using one or more columns as a partitioning key, a table can be put into one of the data distribution methodologies, namely range, hash and list partitioning.

3.1.1 Range Partitioning

Range partitioning is the most commonly used type of partitioning and it instructs the Oracle database[8] to place rows in partitions according to ranges of values such as dates, numbers, or characters. Each partition has a lower and upper bound on which, when inserting rows into partitions, Oracle determines in which of these partitions to place data based on the specified bound values.

In order to determine to which partition the row belongs, the partition by range clause in the create table statement should have a range-based partition key. Each statement of range partition must have VALUES LESS THAN clause that will identify the noninclusive value of the upper bound of the range. When partitioning by range, the first partition defined for a range has no lower bound. The values less than those set in the first partition's VALUE LESS THAN clause are put into the first partition. The lower bound of the rest of the partitions, other than the first partition, is determined by the upper bound of the previous partitions.

As an option, it is possible to create range partitioned tables with the highest partition value by specifying the MAXVALUE clause. The purpose of this key is that any row that

[8] see footnote 7

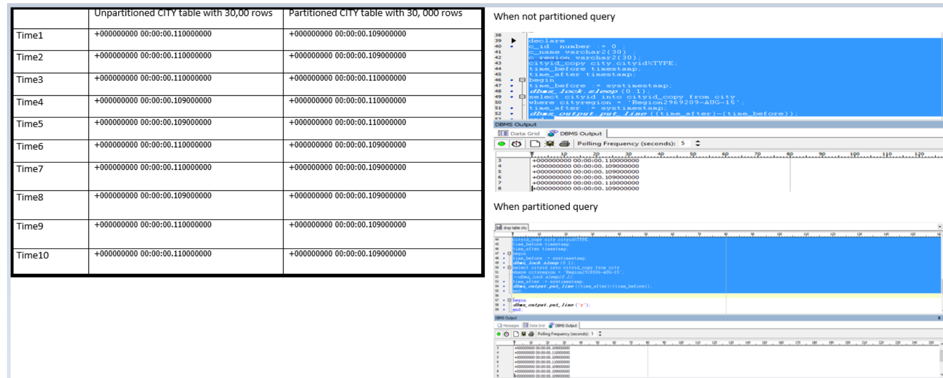
does not have a partition key and that falls in the lower ranges is inserted into this top most MAXVALUE partition.

As an example, in listing 2, a table is created first with no partitions and data is populated into the table. After data is inserted into the table, a SELECT statement is applied against the table to see how long it takes to fetch those data.

```
1  create table city
2  (cityid numeric(20),
3  cityname varchar2(30),
4  cityregion varchar2(30),
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  declare
8  c_id  number := 0 ;
9  c_name varchar2(30) ;
10 c_region varchar2(30);
11 c_pop varchar2(30);
12 cityid_copy city.cityid%TYPE;
13 time_before timestamp;
14 time_after timestamp;
15 begin
16 while (c_id <= 300000)
17 loop
18 c_name     := 'City' || c_id || sysdate;
19 c_region   := 'Region' || c_id || sysdate;
20 c_pop      := c_id + 500 || sysdate;
21 insert into city
22 (cityid,cityname,cityregion,citypopulation)
23 values(c_id,c_name,c_region,c_pop);
24 c_id := c_id +1;
25 end loop;
26 end;
```

Listing 2: Populating a newly created table with data

Using the Procedural Language/Structured Query Language (PL/SQL) code of listing 2, 30,000 rows were first inserted into CITY table to see how a small-sized table is affected by partitioning. After that, the same process was repeated for CITY table with 300,000 and 3,000,000 rows.



	Unpartitioned CITY table with 30,00 rows	Partitioned CITY table with 30, 000 rows
Time1	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time2	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time3	+000000000 00:00:00.110000000	+000000000 00:00:00.110000000
Time4	+000000000 00:00:00.109000000	+000000000 00:00:00.110000000
Time5	+000000000 00:00:00.109000000	+000000000 00:00:00.110000000
Time6	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time7	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time8	+000000000 00:00:00.109000000	+000000000 00:00:00.109000000
Time9	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time10	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000

Figure 4: Time to fetch a row for a partitioned and non-partitioned CITY table with 30,000 rows

Using the PL/SQL code in listing 3, data is fetched from CITY table and the time it took to fetch the data was also registered in a table as shown in figure 4. This table contains 10 different execution times for a row to be fetched from CITY table when the table is partitioned and not partitioned. In PL/SQL code of listing 3, an Oracle procedure called `dbms_lock.sleep` is added. This procedure, which is also sometimes known by the name Oracle sleep procedure, stops the execution of the current thread for n seconds. In this case, the thread at the execution time of the below PL/SQL code in listing 3, was forced to delay for 100 milliseconds to get a more accurate time reading. For example in the table of figure 4, the first row of the non-partitioned CITY table shows 110 milliseconds. Subtracting the sleep time of 100 milliseconds will give 10 milliseconds. So the average time of the 10 execution times listed in figure 4 for the non-partitioned CITY table with 30, 000 rows of record is 9.7 milliseconds.

```
1 declare
2 c_id  number := 0 ;
3 c_name varchar2(30) ;
4 c_region varchar2(30);
5 cityid_copy city.cityid%TYPE;
6 time_before timestamp;
7 time_after timestamp;
8 begin
```

```
9  time_before := systimestamp;
10 dbms_lock.sleep(0.1);
11 select cityid into cityid_copy from city
12 where cityregion = 'Region29946203-AUG-15';
13 time_after := systimestamp;
14 dbms_output.put_line((time_after)-(time_before));
15 end;
```

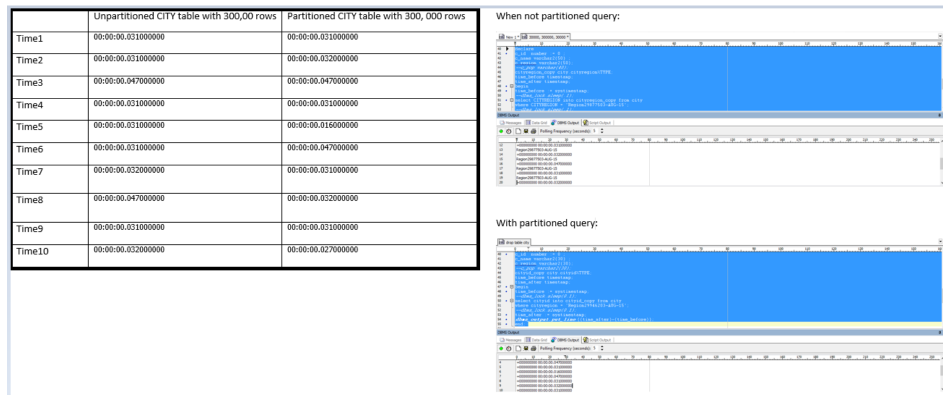
Listing 3: PL/SQL code for time output

The second column in figure 4 shows a 10 execution time list for CITY table when partitioned. Here, CITY table is partitioned into four different partitions, namely partition A1, A2, A3 and A4. As shown in listing 4, CITY table is partitioned into four different partitions and each partition is stored in four different tablespaces, namely `system`, `sysaux`, `users` `and` `LARGE_TBL`. (The complete code is available in listing 16 of appendix 1).

```
1  create table city
2  (cityid numeric(20),
3  cityname varchar2(30),
4  cityregion varchar2(30),
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  partition by range (cityid)(
8  partition A1 values less than (7500) tablespace system,
9  partition A2 values less than (15000) tablespace sysaux,
10 partition A3 values less than (22500) tablespace users,
11 partition A4 values less than (maxvalue) tablespace
       LARGE_TBL);
```

Listing 4: Populating and range partitioning a newly created table with data

After partitioning CITY table into four partitions as shown in listing 4, a PL/SQL code is executed against CITY table as shown in listing 3. The 10 execution times are listed in the second column of the table of figure 4. The average time of the 10 execution times listed in figure 4 for partitioned CITY table with 30, 000 rows of record is 9.3 milliseconds. For CITY table with 30,000 rows, the average time of execution for that particular SELECT statement inside the PL/SQL code of listing 3 is almost the same whether the table is partitioned or not.



	Unpartitioned CITY table with 300,00 rows	Partitioned CITY table with 300, 000 rows
Time1	00:00:00.031000000	00:00:00.031000000
Time2	00:00:00.031000000	00:00:00.032000000
Time3	00:00:00.047000000	00:00:00.047000000
Time4	00:00:00.031000000	00:00:00.031000000
Time5	00:00:00.031000000	00:00:00.016000000
Time6	00:00:00.031000000	00:00:00.047000000
Time7	00:00:00.032000000	00:00:00.031000000
Time8	00:00:00.047000000	00:00:00.032000000
Time9	00:00:00.031000000	00:00:00.031000000
Time10	00:00:00.032000000	00:00:00.027000000

Figure 5: Time to fetch a row for a partitioned and non-partitioned CITY table with 300,000 rows

In a similar way as in listing 2, CITY table is populated with data. But in this step, CITY table is dropped first and recreated. After that, CITY table is populated with 300,000 rows. The complete code to perform this operation is available in listing 17 of appendix 1.

As shown in figure 5, the table contains 10 different execution times for CITY table with 300,000 rows when partitioned and non-partitioned. The average time of the 10 execution times listed in figure 5 is 34.4 milliseconds. The second column of this table shows the execution times when CITY table is partitioned into four different partitions as shown in listing 5. The complete code is available in listing 18 of appendix 1.

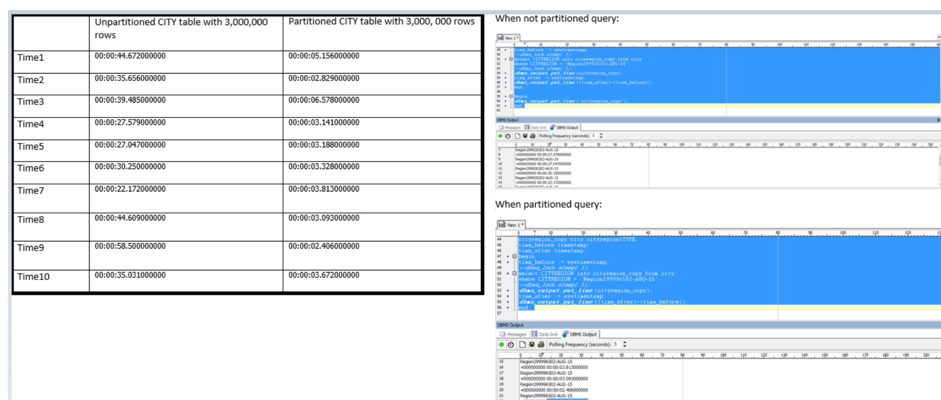
```
1  create table city
2  (cityid numeric(20),
3  cityname varchar2(30),
4  cityregion varchar2(30),
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  partition by range (cityid)(
8  partition A1 values less than (75000) tablespace system,
9  partition A2 values less than (150000) tablespace sysaux,
10 partition A3 values less than (225000) tablespace users,
11 partition A4 values less than (maxvalue) tablespace
       LARGE_TBL);
```

Listing 5: Populating and partitioning a newly created table with data

The average time is 32.5 milliseconds. Comparing the two average execution times, there is no huge difference whether CITY table with 300,000 rows is partitioned or not. So far,

the only difference between the two analyses, CITY table with 30,000 and 300,000 rows, is that it takes more time to execute the PL/SQL code in listing 3 for CITY table with 300,000 rows than 30,000 rows.

The next analysis is for CITY table with 3 millions of records. The same analysis as in this subsection 3.1.1 for 30,000 and 300,000 rows is applied against CITY table except here the number of rows is increased to 3 million. As in the table of figure 6, the average time for a non-partitioned CITY table with 3 millions rows is 36 seconds and 500 milliseconds. The second column of the table in figure 6 shows the list of 10 execution times when CITY table is partitioned as in listing 19 of appendix 1. So the average time is 3 seconds and 720 milliseconds. Therefore, when CITY table with 3 million records is partitioned as in listing 19 of appendix 1 , retrieving a single row takes 32 seconds and 780 milliseconds less time, which is 9.8 times faster than when not partitioned.

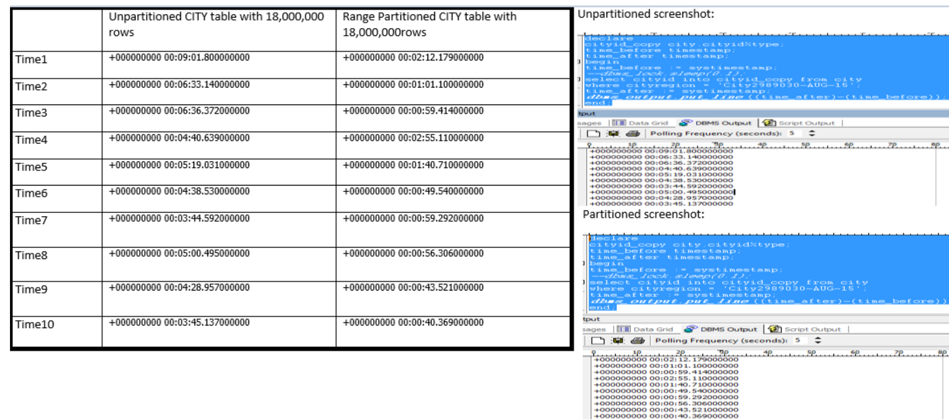


	Unpartitioned CITY table with 3,000,000 rows	Partitioned CITY table with 3,000, 000 rows
Time1	00:00:44.672000000	00:00:05.156000000
Time2	00:00:35.656000000	00:00:02.829000000
Time3	00:00:39.485000000	00:00:06.578000000
Time4	00:00:27.579000000	00:00:03.141000000
Time5	00:00:27.047000000	00:00:03.188000000
Time6	00:00:30.250000000	00:00:03.328000000
Time7	00:00:22.172000000	00:00:03.813000000
Time8	00:00:44.609000000	00:00:03.093000000
Time9	00:00:58.500000000	00:00:02.406000000
Time10	00:00:35.031000000	00:00:03.672000000

Figure 6: Time to fetch a row for partitioned and non-partitioned CITY table with 3,000,000 rows

Figure 7 shows 10 execution times for a select statement against CITY table with 18 million records when partitioned and non-partitioned. According to this table, the average time for the non-partitioned part is 283 seconds and 243 milliseconds (more than 4 minutes). The average time for the second column in figure 7 is 77 seconds and 354 milliseconds. Therefore, if CITY table with 18 million rows is not partitioned, it will take about 206 seconds more time, which is 3.6 times slower to fetch a single row than when CITY table is range-partitioned into four different partitions.

Creating a range-partitioned global index is also a way of increasing performance. When creating a range-partitioned global index, the rules used are similar to creating a range-partitioned table.



	Unpartitioned CITY table with 18,000,000 rows	Range Partitioned CITY table with 18,000,000rows
Time1	+000000000 00:09:01.800000000	+000000000 00:02:12.179000000
Time2	+000000000 00:06:33.140000000	+000000000 00:01:01.100000000
Time3	+000000000 00:06:36.372000000	+000000000 00:00:59.414000000
Time4	+000000000 00:04:40.639000000	+000000000 00:02:55.110000000
Time5	+000000000 00:05:19.031000000	+000000000 00:01:40.710000000
Time6	+000000000 00:04:38.530000000	+000000000 00:00:49.540000000
Time7	+000000000 00:03:44.592000000	+000000000 00:00:59.292000000
Time8	+000000000 00:05:00.495000000	+000000000 00:00:56.306000000
Time9	+000000000 00:04:28.957000000	+000000000 00:00:43.521000000
Time10	+000000000 00:03:45.137000000	+000000000 00:00:40.369000000

Figure 7: Time to fetch a row for a partitioned and non-partitioned CITY table with 18,000,000 rows

```
1 CREATE INDEX cityid_ix ON city(cityid)
2 GLOBAL PARTITION BY RANGE(cityid)
3 ( PARTITION p1 VALUES LESS THAN (1000000)
4 , PARTITION p2 VALUES LESS THAN (2000000)
5 , PARTITION p3 VALUES LESS THAN (maxvalue)
6 );
```

Listing 6: Creating a range partitioned global Index on CITY Table

As an example, listing 6 creates a range partitioned global index on CITYID for the table CITY. Each index partition in the above list is named but is stored in the default tablespace for the index.

3.1.2 Hash Partitioning

Hash partitioning is a way of distributing data evenly across the devices giving partitions almost the same size. This is done by the help of an algorithm called a hashing algorithm which Oracle uses for its table partitioning techniques. This algorithm maps data into the available partitions under the condition of the partitioning primary key that is identified. Hash partitioning is mostly used as an alternative to range partitioning as it is an easy to use method especially when it comes to non-historic data or when the data has no known key [10, p.154].

A table with no partitions is created and populated with random data that has no primary and unique keys. After data is inserted using the below PL/SQL code as in listing 7, a

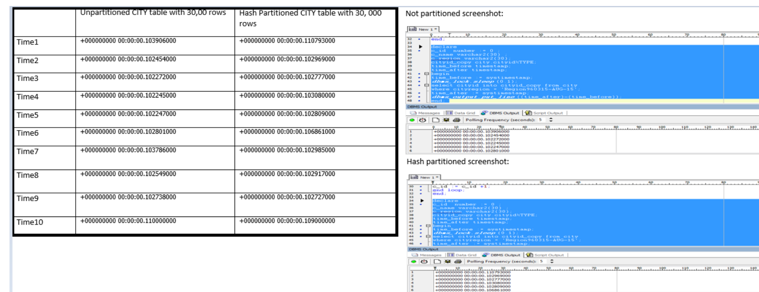
basic SELECT statement is performed to see how long it takes to fetch the data.

The same pattern of the populating and partitioning of CITY table as in subsection 3.1.1 is used to test hash partitioning. In this subsection, the only difference is that the partitioning method is hash partitioning as shown in line 6 of listing 7.

```
1  create table city
2  (cityid numeric(20),
3  cityname varchar2(30),
4  cityregion varchar2(30),
5  citypopulation varchar2(30)),
6  partition by hash (cityid)
7  partitions 4
8  STORE IN (system, sysaux, users, LARGE_TBL);
9  declare
10 c_id  number := 0 ;
11 c_name varchar2(30);
12 c_region varchar2(30);
13 c_pop varchar2(30);
14 cityid_copy city.cityid%TYPE;
15 time_before timestamp;
16 time_after timestamp;
17 begin
18 while (c_id <= 30000)
19 loop
20 c_name     := 'City' || c_id || sysdate;
21 c_region   := 'Region' || c_id || sysdate;
22 c_pop      := c_id + 500 || sysdate;
23 insert into city
24 (cityid,cityname,cityregion,citypopulation)
25 values(c_id,c_name,c_region,c_pop);
26 c_id := c_id +1;
27 end loop;
28 end;
```

Listing 7: Populating a table with data and hash partitioning

In hash partitioning, the only attribute that can be specified is tablespace. As a mandatory rule for hash partitioning, all of the hash partitions of a table share the same segment attributes except the tablespace which is inherited from the table level.



	Unpartitioned CITY table with 30,00 rows	Hash Partitioned CITY table with 30, 000 rows
Time1	+000000000 00:00:00.103906000	+000000000 00:00:00.110793000
Time2	+000000000 00:00:00.102454000	+000000000 00:00:00.102969000
Time3	+000000000 00:00:00.102272000	+000000000 00:00:00.102777000
Time4	+000000000 00:00:00.102245000	+000000000 00:00:00.103080000
Time5	+000000000 00:00:00.102247000	+000000000 00:00:00.102809000
Time6	+000000000 00:00:00.102801000	+000000000 00:00:00.106861000
Time7	+000000000 00:00:00.103786000	+000000000 00:00:00.102985000
Time8	+000000000 00:00:00.102549000	+000000000 00:00:00.102917000
Time9	+000000000 00:00:00.102738000	+000000000 00:00:00.102727000
Time10	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000

Figure 8: Time to fetch a row for a hash partitioned and non-partitioned CITY table with 30,000 rows

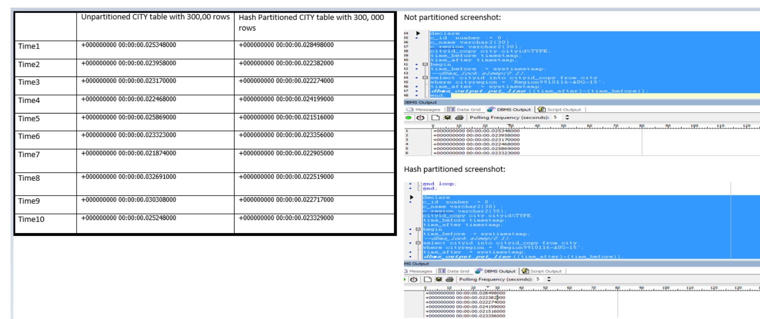
In this scenario, the table is hash-partitioned into four partitions and each partition is stored in a different tablespace. As in figure 8, 10 execution times are registered for a SELECT statement. So the average time for CITY table when not partitioned is 10.35 milliseconds.The second column of the table in figure 8 shows ten execution times when CITY table with 30,000 is hash partitioned into four different partitions. So the average time is 10.47 milliseconds. Therefore, there is no difference in performance whether CITY table with 30,000 rows is hash partitioned or not.

In the following analysis, CITY table is created and populated in the same way as in listing 7. But first, it is dropped and recreated. Then CITY table is hash partitioned and populated with 300,000 rows as in 8. The complete code can be found in listing 20 of appendix 2.

```
1  begin
2  drop table CITY;
3  create table city
4  (cityid numeric(20),
5  cityname varchar2(30),
6  cityregion varchar2(30),
7  citypopulation varchar2(30)),
8  partition by hash (cityid)
9  partitions 4
10 STORE IN (system, sysaux, users, LARGE_TBL);
```

Listing 8: Populating and partitioning a newly created table with data

As shown in figure 9, the average time of the first column which is when CITY table with 300,000 rows is not partitioned is 25.4 milliseconds. The second column shows the list of execution times when CITY table with 300,000 rows is hash partitioned into four partitions and each partition is stored in four different tablespaces. So the average time is 23.4 milliseconds. Therefore, there is no difference in performance whether CITY table with 300,000 rows is hash partitioned or not.



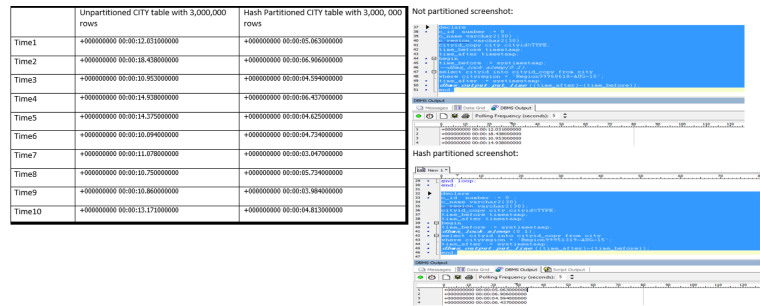
	Unpartitioned CITY table with 300,00 rows	Hash Partitioned CITY table with 300, 000 rows
Time1	+000000000 00:00:00.025348000	+000000000 00:00:00.028498000
Time2	+000000000 00:00:00.023958000	+000000000 00:00:00.022382000
Time3	+000000000 00:00:00.023170000	+000000000 00:00:00.022274000
Time4	+000000000 00:00:00.022468000	+000000000 00:00:00.024199000
Time5	+000000000 00:00:00.025869000	+000000000 00:00:00.021516000
Time6	+000000000 00:00:00.023323000	+000000000 00:00:00.023356000
Time7	+000000000 00:00:00.021874000	+000000000 00:00:00.022905000
Time8	+000000000 00:00:00.032691000	+000000000 00:00:00.022519000
Time9	+000000000 00:00:00.030308000	+000000000 00:00:00.022717000
Time10	+000000000 00:00:00.025248000	+000000000 00:00:00.023329000

Figure 9: Time to fetch a row for a hash partitioned and non-partitioned CITY table with 300,000 rows

What if the the number of rows in CITY table is increased to 3 million, will the time to SELECT a single row from the table be the same when partitioned and not partitioned? Figure 10 shows the list of times for a single row SELECT statement for a hash partitioned and non-partitioned CITY table. According to this table in figure 10, when CITY table with 3 million rows is not partitioned, the average execution time of a SELECT statement for a single row is 12 seconds and 669 milliseconds. On the other hand, the second column of the table in figure 10 shows the time lists when CITY table with 3 million records is hash partitioned into four different partitions. Each partition is stored in four different tablespaces, namely `system`, `sysaux`, `users` **and** `LARGE_TBL`. The average time is 4 seconds and 494 milliseconds.

Therefore, retrieving a single row from a hash partitioned CITY table with 3 million records will take 8 seconds and 175 milliseconds less time, which is 2.81 times faster than retrieving from a non-partitioned CITY table with the same number of rows.

Figure 11 below shows time lists for a single row fetch against CITY table with 18 million records when hash partitioned and non-partitioned. So the average time for the first column in figure 11 which is when CITY table is not partitioned is around 300 seconds and 963 milliseconds (more than 5 minutes). The average time for the second column in figure 11 which is when CITY table is hash partitioned into four partitions is 56 seconds

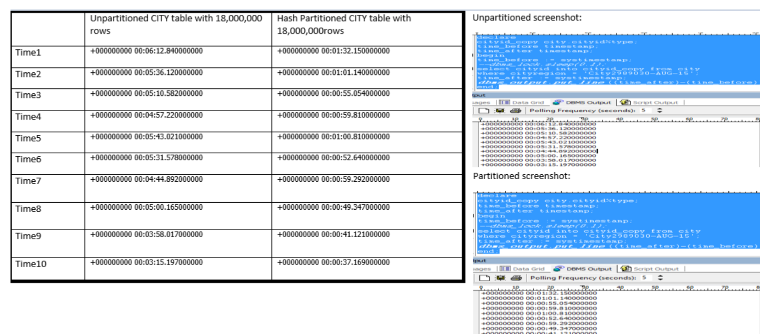


	Unpartitioned CITY table with 3,000,000 rows	Hash Partitioned CITY table with 3,000, 000 rows
Time1	+000000000 00:00:12.031000000	+000000000 00:00:05.063000000
Time2	+000000000 00:00:18.438000000	+000000000 00:00:06.906000000
Time3	+000000000 00:00:10.953000000	+000000000 00:00:04.594000000
Time4	+000000000 00:00:14.938000000	+000000000 00:00:06.437000000
Time5	+000000000 00:00:14.375000000	+000000000 00:00:04.625000000
Time6	+000000000 00:00:10.094000000	+000000000 00:00:04.734000000
Time7	+000000000 00:00:11.078000000	+000000000 00:00:03.047000000
Time8	+000000000 00:00:10.750000000	+000000000 00:00:05.734000000
Time9	+000000000 00:00:10.860000000	+000000000 00:00:03.984000000
Time10	+000000000 00:00:13.171000000	+000000000 00:00:04.813000000

Figure 10: Time to fetch a row for a hash partitioned and non-partitioned CITY table with 3,000,000 rows

and 853 milliseconds (less than a minute).

Therefore, fetching a row from a hash partitioned CITY table with 18 million records will take 244 seconds and 111 milliseconds less time, which is 5.3 times faster than fetching from a non-partitioned CITY table with the same number of records.



	Unpartitioned CITY table with 18,000,000 rows	Hash Partitioned CITY table with 18,000,000rows
Time1	+000000000 00:06:12.840000000	+000000000 00:01:32.150000000
Time2	+000000000 00:05:36.120000000	+000000000 00:01:01.140000000
Time3	+000000000 00:05:10.582000000	+000000000 00:00:55.054000000
Time4	+000000000 00:04:57.220000000	+000000000 00:00:59.810000000
Time5	+000000000 00:05:43.021000000	+000000000 00:01:00.810000000
Time6	+000000000 00:05:31.578000000	+000000000 00:00:52.640000000
Time7	+000000000 00:04:44.892000000	+000000000 00:00:59.292000000
Time8	+000000000 00:05:00.165000000	+000000000 00:00:49.347000000
Time9	+000000000 00:03:58.017000000	+000000000 00:00:41.121000000
Time10	+000000000 00:03:15.197000000	+000000000 00:00:37.169000000

Figure 11: Time to fetch a row for a hash partitioned and non-partitioned CITY table with 18,000,000 rows

If a local index is created for the table, the database builds the index so that it is equally partitioned with the underlying table. When maintenance operations are performed on the table, the index will also be maintained automatically. A local index on a table can be created as in listing 9:

```
1 CREATE INDEX loc_city_ix ON CITY(CITYID) LOCAL;
```

Listing 9: Creating Local Index on CITY Table

When creating a local index, it is possible to optionally name the hash partitions and table partitions into which the local index partitions are to be stored. But if the hash partitions and table partitions are not specified, then the database uses the name of the corresponding base partition as the index partition name and stores the index partition in the same tablespace as the table partition.

The second type index that can possibly increase the performance of a hash partitioned table is a hash partitioned global index. The hash partitioned global indexes can also limit the impact of index skew on homogeneously increasing column values. The hash partitioned global index is effective with queries that has the equal or IN comparisons. The syntax used to create a hash partitioned global index is similar to that used for a hash partitioned table as shown in listing 10.

```
1 CREATE INDEX glob_city_ix ON CITY (CITYID) GLOBAL
2 PARTITION BY HASH (CITYID1, CITYID2)
3 (PARTITION p1 TABLESPACE tbs_1,
4 PARTITION p2 TABLESPACE tbs_2,
5 PARTITION p3 TABLESPACE tbs_3,
6 PARTITION p4 TABLESPACE tbs_4)
7 PARTITION p5 TABLESPACE tbs_5);
```

Listing 10: Creating Global Index

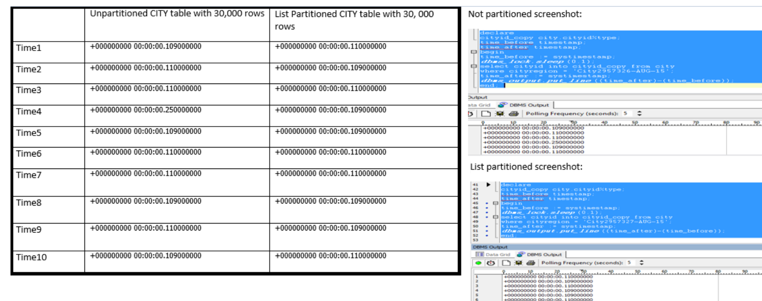
For range-based globally partitioned indexes, there must always be a maximum specified for the index, with a high value of MAXVALUE. This assures that any new insertions into the corresponding table will have a place in the globally partitioned index, so that there will not be an out of bound issue on the index. If MAXVALUE is not mentioned in creating a globally partitioned index, an error will be generated. [11, p.123]

3.1.3 List Partitioning

The methods for creating list partitions are almost similar to those for creating range partitions. However, to create list partitions, the `PARTITION BY LIST` clause has to be specified in the `CREATE TABLE` statement. Lists of literal values which are the distinct values of the partitioning columns that qualify the rows to be part of the partition will specify the `PARTITION`.

By specifying a list of unique values for the partitioning key in the description for each partition, list partitioning is used to explicitly handle rows mapping. List partitioning is mostly used to group and organize data which are in non-relational behaviour.

The default partition in this method is helpful to avoid specifying all possible values for a



	Unpartitioned CITY table with 30,000 rows	List Partitioned CITY table with 30, 000 rows
Time1	+000000000 00:00:00.109000000	+000000000 00:00:00.110000000
Time2	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time3	+000000000 00:00:00.110000000	+000000000 00:00:00.110000000
Time4	+000000000 00:00:00.250000000	+000000000 00:00:00.109000000
Time5	+000000000 00:00:00.109000000	+000000000 00:00:00.109000000
Time6	+000000000 00:00:00.110000000	+000000000 00:00:00.110000000
Time7	+000000000 00:00:00.110000000	+000000000 00:00:00.110000000
Time8	+000000000 00:00:00.109000000	+000000000 00:00:00.109000000
Time9	+000000000 00:00:00.110000000	+000000000 00:00:00.109000000
Time10	+000000000 00:00:00.109000000	+000000000 00:00:00.110000000

Figure 12: Execution times for a non-partitioned and List partition CITY table with 30,000 records

list partitioned table by using a default partition. If there are any rows that do not belong to existing partitions, an error will not be generated.

In list partitions as in range partitions, optional subclauses of a `PARTITION` clause can include physical and other attributes particular to a partition segment. Partitions get the attributes of their parent tables if they are not overridden at the partition level.

To demonstrate and compare a list partitioned table with a table which is not list partitioned, first in linsting 12 a table which is not partitioned is created. CITY table in listing 13 contains 30,000 records.

```
1  declare
2  cityid_copy city.cityid%type;
3  time_before timestamp;
4  time_after timestamp;
5  begin
6  time_before := systimestamp;
7  dbms_lock.sleep(0.1);
8  select cityid into cityid_copy from city
9  where cityregion = 'City2989030-AUG-15';
10 time_after := systimestamp;
11 dbms_output.put_line((time_after)-(time_before));
12 end;
```

Listing 11: PL/SQL code for time output of a SELECT statement

After the table is created, a SELECT statement is performed as shown in listing 11. Ten execution times are listed as in shown in figure 12. Listing 13 shows CITY table with 30,000 List partitioned into four partitions. Each partition is stored in different tablespaces.

```
1  create table city (
2  cityid number not null,
3  cityregion varchar2(100) not null,
4  state varchar2(2) not null,
5  population varchar2(255)) nologging;
6  alter session force parallel dml;
7  insert /*+ append */ into city
8  with states as (
9  select 'NY' state, 1 id_offset from dual union all
10 select 'NJ' state, 2 id_offset from dual union all
11 select 'CA' state, 3 id_offset from dual union all
12 select 'TX' state, 4 id_offset from dual union all
13 select 'WA' state, 5 id_offset from dual union all
14 select 'NC' state, 6 id_offset from dual union all
15 select 'IL' state, 7 id_offset from dual union all
16 select 'KS' state, 8 id_offset from dual union all
17 select 'SC' state, 9 id_offset from dual union all
18 select 'NV' state, 10 id_offset from dual),
19 generator as (
20 select /*+ materialize cardinality(5000000) */ (level - 1) *
       10 id from dual connect by level <= 3000)
21 select id + id_offset, 'City' || (id + id_offset) || sysdate
      , state, id + 500 || sysdate
22 from generator cross join states;
```

Listing 12: Creating a table with no list partition (Copied from Stackoverflow (2015) [12])

The first partition, Partition1 which is stored in the tablespace system, contains records that belong to states NC and NJ. The second partition, Partition2, contains records that belong to three states CA, TX, and WA and these records are stored in the tablespace sysaux. The third partition, Partition3, which is stored in tablespace users contains records for the states of NC and IL. The last partition called Partition4, which is stored in the tablespace `large_tbl` stores records for states KS, SC, and NV. According to the table in figure 12 , there is no difference in terms of performance whether CITY table with 30,000

records is list partitioned or not.

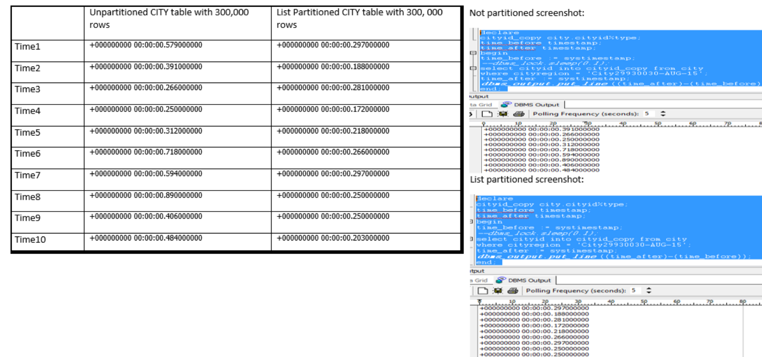
The complete code for listing 13 can be found in listing 21 of appendix 3.

```
1  create table city (
2  cityid number not null,
3  cityregion varchar2(100) not null,
4  state varchar2(2) not null,
5  population varchar2(255))
6  partition by list(state)
7  (
8  partition partion1 values ('NY','NJ') tablespace system,
9  partition partion2 values ('CA', 'TX','WA') tablespace
       sysaux,
10 partition partion3 values ('NC', 'IL') tablespace users,
11 partition partion4 values ('KS','SC','NV') tablespace
       large_tbl) nologging;
```

Listing 13: Creating a table with List partition (Copied from Stackoverflow (2015) [12])

Using the same steps as for CITY table with 30,000 records, the records in CITY table are increased to 300,000 rows. This is to test if the time to SELECT a single row from CITY table is improved when the table is list partitioned and non-partitioned. Figure 13 shows 10 different execution times for both non-partitioned and list partitioned CITY table with 300,000 records. The average time to fetch a single row from the table when not list partitioned is 489 milliseconds. The average time when CITY table is list partitioned is 242.2 milliseconds. Therefore, selecting a single row from CITY table with 300,000 rows and list partitioned takes 246.8 milliseconds less time, which is 2 times faster, than selecting from not list partitioned CITY table.

To see if this performance gain will be maintained, improved or will be worst, the number of records in CITY table will be increased to 3 million. Figure 14 below shows the execution times of a SELECT statement for a single row for both non-partitioned and List partitioned CITY table with 3 million records. So the average time for the first column which is for CITY table when not partitioned is 12 seconds and 992.4 milliseconds. For the list partitioned

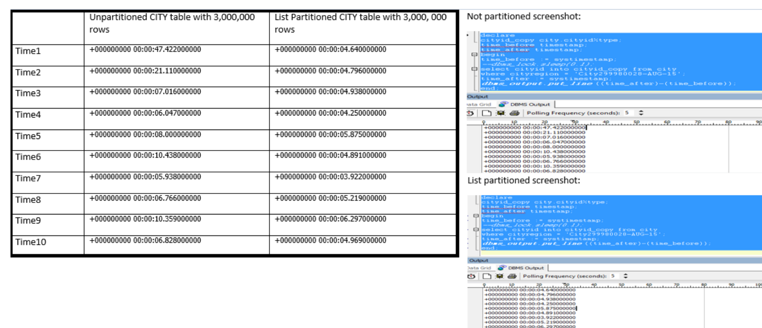


	Unpartitioned CITY table with 300,000 rows	List Partitioned CITY table with 300, 000 rows
Time1	+000000000 00:00:00.579000000	+000000000 00:00:00.297000000
Time2	+000000000 00:00:00.391000000	+000000000 00:00:00.188000000
Time3	+000000000 00:00:00.266000000	+000000000 00:00:00.281000000
Time4	+000000000 00:00:00.250000000	+000000000 00:00:00.172000000
Time5	+000000000 00:00:00.312000000	+000000000 00:00:00.218000000
Time6	+000000000 00:00:00.718000000	+000000000 00:00:00.266000000
Time7	+000000000 00:00:00.594000000	+000000000 00:00:00.297000000
Time8	+000000000 00:00:00.890000000	+000000000 00:00:00.250000000
Time9	+000000000 00:00:00.406000000	+000000000 00:00:00.250000000
Time10	+000000000 00:00:00.484000000	+000000000 00:00:00.203000000

Figure 13: Execution times for a non-partitioned and list partitioned CITY table with 300,000 records

part, the average time is 4 seconds and 979.7 milliseconds.

In this scenario, fetching a single row from list partitioned CITY table with 3 million records takes 8 seconds and 12.7 milliseconds less time, which is 2.6 faster than pulling a row from a non-partitioned CITY table with the same amount of data.

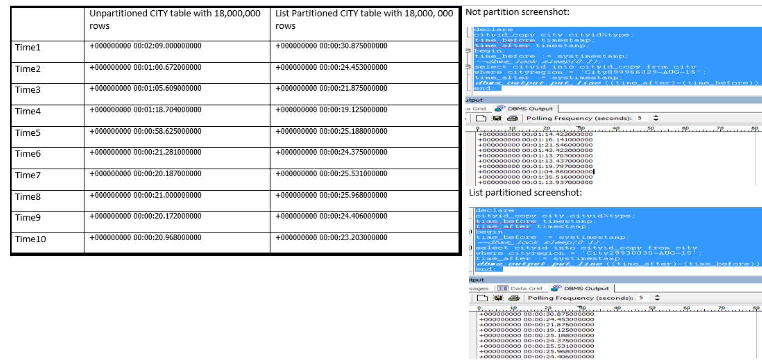


	Unpartitioned CITY table with 3,000,000 rows	List Partitioned CITY table with 3,000, 000 rows
Time1	+000000000 00:00:47.422000000	+000000000 00:00:04.640000000
Time2	+000000000 00:00:21.110000000	+000000000 00:00:04.796000000
Time3	+000000000 00:00:07.016000000	+000000000 00:00:04.938000000
Time4	+000000000 00:00:06.047000000	+000000000 00:00:04.250000000
Time5	+000000000 00:00:08.000000000	+000000000 00:00:05.875000000
Time6	+000000000 00:00:10.438000000	+000000000 00:00:04.891000000
Time7	+000000000 00:00:05.938000000	+000000000 00:00:03.922000000
Time8	+000000000 00:00:06.766000000	+000000000 00:00:05.219000000
Time9	+000000000 00:00:10.359000000	+000000000 00:00:06.297000000
Time10	+000000000 00:00:06.828000000	+000000000 00:00:04.969000000

Figure 14: Execution times for a non-partitioned and list partitioned CITY table with 3,000,000 records

Figure 15 shows the execution times of a SELECT statement for a single row for both non-partitioned and list partitioned CITY table when the record is increased up to 18 million rows. According to these time lists, the average time of SELECT statement against CITY table with 18 million records and not partitioned is 49 seconds and 621.8 milliseconds. The second column in figure 15 shows the time lists to fetch a single row when CITY table is List partitioned into four different partitions and each partition is stored in different tablespaces. So the average time for this is 24 seconds and 499.9 milliseconds.

Therefore, when CITY table with 18 million records is list partitioned, it takes 25 seconds and 122 milliseconds less time to pull up a single row, which is 2.02 faster than when not list partitioned.



	Unpartitioned CITY table with 18,000,000 rows	List Partitioned CITY table with 18,000, 000 rows
Time1	+000000000 00:02:09.000000000	+000000000 00:00:30.875000000
Time2	+000000000 00:01:00.672000000	+000000000 00:00:24.453000000
Time3	+000000000 00:01:05.609000000	+000000000 00:00:21.875000000
Time4	+000000000 00:01:18.704000000	+000000000 00:00:19.125000000
Time5	+000000000 00:00:58.625000000	+000000000 00:00:25.188000000
Time6	+000000000 00:00:21.281000000	+000000000 00:00:24.375000000
Time7	+000000000 00:00:20.187000000	+000000000 00:00:25.531000000
Time8	+000000000 00:00:21.000000000	+000000000 00:00:25.968000000
Time9	+000000000 00:00:20.172000000	+000000000 00:00:24.406000000
Time10	+000000000 00:00:20.968000000	+000000000 00:00:23.203000000

Figure 15: Execution times for a non-partitioned and list partitioned CITY table with 18,000,000 records

3.2 Composite Partitioning

Composite partitioning is a collection of the basic partitioning techniques. Composite partitioning works in a way that if a table is partitioned by one of the partitioning techniques, the partitioned will then be further sub-partitioned using another partitioning technique and all the sub-partitioned represent a logical subset of the data for their respective partitions. [10, p.314]

Composite partitioning is mostly used to support historical operations like adding a new range partition to the existing one. The types of composite partitioning available are:

- Composite Range-Range Partitioning
- Composite Range-Hash Partitioning
- Composite Range-List Partitioning
- Composite List-Range Partitioning
- Composite List-Hash Partitioning
- Composite List-List Partitioning
- Composite Hash-Hash Partitioning
- Composite Hash-List Partitioning
- Composite Hash-Range Partitioning

3.2.1 Composite Range-Range Partitioning

The methods used to create a range-range composite partitioned table is similar to a non-composite range partitioned table. The method enables some optional subclauses. The subclauses of the partition which may be included are tablespaces which will specify

physical and other attributes. The example in listing 14 will create a composite range-range partitioning.

```
1  CREATE TABLE order_line(
2  trip_id NUMBER NOT NULL,
3  received_date DATE NOT NULL,
4  delivery_date DATE NOT NULL,
5  customer_code NUMBER NOT NULL,
6  amount_paid NUMBER NOT NULL
7  )
8  PARTITION BY RANGE (received_date)
9  SUBPARTITION BY RANGE (delivery_date)
10 ( PARTITION p_2014_Jan VALUES LESS THAN (TO_DATE('01-FEB
      -2014','dd-MON-yyyy'))
11 ( SUBPARTITION p14_Jan1 VALUES LESS THAN (TO_DATE('15-FEB
      -2014','dd-MON-yyyy'))
12 , SUBPARTITION p14_Jan2 VALUES LESS THAN (TO_DATE('01-MAR
      -2014','dd-MON-yyyy'))
13 , SUBPARTITION p14_Jan3 VALUES LESS THAN (MAXVALUE)
14 )
15 , PARTITION p_2014_Feb VALUES LESS THAN (TO_DATE('01-MAR
      -2014','dd-MON-yyyy'))
16 ( SUBPARTITION p14_Feb1 VALUES LESS THAN (TO_DATE('15-MAR
      -2014','dd-MON-yyyy'))
17 , SUBPARTITION p14_Feb2 VALUES LESS THAN (TO_DATE('01-APR
      -2014','dd-MON-yyyy'))
18 , SUBPARTITION p14_Feb3 VALUES LESS THAN (MAXVALUE)
19 )
20 , PARTITION p_2014_Mar VALUES LESS THAN (TO_DATE('01-APR
      -2014','dd-MON-yyyy'))
21 ( SUBPARTITION p14_Mar1 VALUES LESS THAN (TO_DATE('15-APR
      -2014','dd-MON-yyyy'))
22 , SUBPARTITION p14_Mar2 VALUES LESS THAN (TO_DATE('01-MAY
      -2014','dd-MON-yyyy'))
```

```
23  , SUBPARTITION p06_Mar3 VALUES LESS THAN (MAXVALUE)
24  )
25  , PARTITION p_2014_Apr VALUES LESS THAN (TO_DATE('01-MAY
      -2014','dd-MON-yyyy'))
26  ( SUBPARTITION p14_Apr1 VALUES LESS THAN (TO_DATE('15-MAY
      -2014','dd-MON-yyyy'))
27  , SUBPARTITION p14_Apr2 VALUES LESS THAN (TO_DATE('01-JUN
      -2014','dd-MON-yyyy'))
28  , SUBPARTITION p14_Apr3 VALUES LESS THAN (MAXVALUE)
29  )
30  , PARTITION p_2014_May VALUES LESS THAN (TO_DATE('01-JUN
      -2014','dd-MON-yyyy'))
31  ( SUBPARTITION p14_May1 VALUES LESS THAN (TO_DATE('15-JUN
      -2014','dd-MON-yyyy'))
32  , SUBPARTITION p14_May2 VALUES LESS THAN (TO_DATE('01-JUL
      -2014','dd-MON-yyyy'))
33  , SUBPARTITION p14_May3 VALUES LESS THAN (MAXVALUE)
34  )
35  , PARTITION p_2014_Jun VALUES LESS THAN (TO_DATE('01-JUL
      -2014','dd-MON-yyyy'))
36  ( SUBPARTITION p14_Jun1 VALUES LESS THAN (TO_DATE('15-JUL
      -2014','dd-MON-yyyy'))
37  , SUBPARTITION p14_Jun2 VALUES LESS THAN (TO_DATE('01-AUG
      -2014','dd-MON-yyyy'))
38  , SUBPARTITION p14_Jun3 VALUES LESS THAN (MAXVALUE)
39  )
40  );
```

Listing 14: Creating a composite range range partition

When inserting data into a composite range-range partitioned table, a row is mapped to a partition by checking whether the value of the partitioning column for a row falls within a specific partition range. The row is then further mapped to a sub-partition within that partition by specifying whether the value of the sub-partitioning column belongs to that specific range.

3.2.2 Composite Range-Hash Partitioning

The way composite range-hash partitioning works is that it partitions data using the range method, and within each partition, subpartitions it using the hash method. The benefits obtained from using composite range-hash partitioning are improved manageability of range partitioning and the data placement, striping, and parallelism of hash partitioning.

The following example in listing 15 creates a Composite Range-Hash Partition.

```
1  CREATE TABLE transaction
2  ( transaction_id NUMBER(6)
3  , customer_id NUMBER
4  , period_id DATE
5  , promotion_code NUMBER(6)
6  , quantity_sold NUMBER(4)
7  , amount_paid NUMBER(10,2)
8  )
9  PARTITION BY RANGE (period_id) SUBPARTITION BY HASH (
      customer_id)
10 SUBPARTITIONS 10 STORE IN (tbs1, tbs2, tbs3, tbs4, tbs5)
11 ( PARTITION transaction_p1_2014 VALUES LESS THAN (TO_DATE('
      01-FEB-2014','dd-MON-yyyy'))
12 , PARTITION transaction_p2_2014 VALUES LESS THAN (TO_DATE('
      01-APR-2014','dd-MON-yyyy'))
13 , PARTITION transaction_p3_2014 VALUES LESS THAN (TO_DATE('
      01-JUN-2014','dd-MON-yyyy'))
14 , PARTITION transaction_p4_2014 VALUES LESS THAN (TO_DATE('
      01-AUG-2014','dd-MON-yyyy'))
15 , PARTITION transaction_p4_2014 VALUES LESS THAN (TO_DATE('
      01-JAN-2015','dd-MON-yyyy'))
16 );
```

Listing 15: Creating a composite range hash partition

In listing 15, there are five range partitions each containing ten sub-partitions. The `STORE`

IN clause distributes the sub-partitions across the five tablespaces mentioned.

4 Methods and Materials

The working environment for this thesis and the materials used were as follows:

1. A personal computer (Laptop)
2. Oracle Database 12c installed on the personal computer
3. Toad for Oracle used to access the Oracle database

Figure 16 shows the description of the personal laptop used in this thesis and this laptop has basic features.

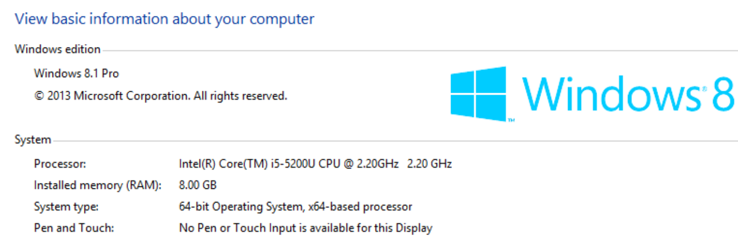


Figure 16: Personal Laptop description used in this thesis

The database installed and used on the personal computer was Oracle database version 12c. Oracle Database 12c is a relational database management system (RDBMS) designed for both on-premises and cloud uses. This database is deployable on a choice of clustered or single servers. It provides features for managing data in transaction processing, business intelligence and content management applications.

In order to access the database and manipulate the data, the application used was Toad for Oracle. Toad, which stands for Tools for Oracle Application Development, is used for application development, database development and other purposes. The languages used in this application are both SQL and PL/SQL.

The measurements were copied from Toad after execution of PL/SQL code as shown in

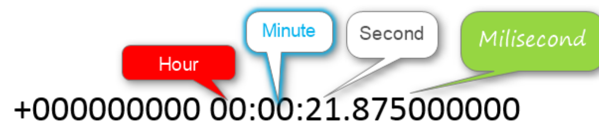


Figure 17: Sample measurement time explanation

listing 3. Figure 17 shows what each section in one time measurement represents.

5 Results and Discussions

The target of this thesis was to test the different Oracle partitioning techniques for 30,000, 300,000, 3,000,000, 30,000,000 and 300,000,000 rows in CITY table. But testing was limited only for 30,000, 300,000, 3,000,000, 18,000,000 for two reasons:

- Disk space limitation
- Inserting rows beyond 18 million took a long time and ended up shutting down the database and locking the CITY table repeatedly

5.1 Results and Discussions on Range Partitioning

Figure 18 shows the performance difference of a range partitioned and non-partitioned CITY table with the same amount of data inserted.

According to this graph, when more data is added to CITY table, more time will be needed to pull data from the table whether the table is partitioned or not. But there is a difference in the amount of time spent to pull data from CITY table when range partitioned and not partitioned. Figure 18 shows that as the number of data inserted into CITY table increases, it takes more time to select a single row from the table when the table is not partitioned than when range partitioned. For example, when CITY table with 18 million rows is not partitioned, it will take about 206 seconds more time to fetch a single row than when CITY table is range partitioned into four different partitions.

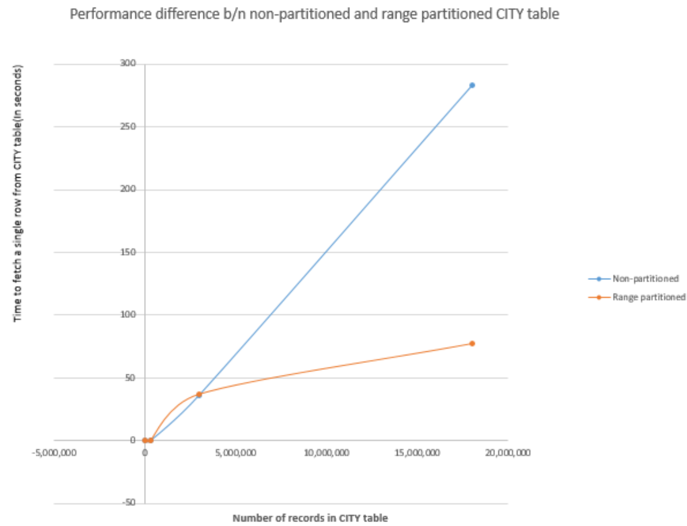


Figure 18: Performance difference between non-partitioned and range partitioned CITY table

5.2 Results and Discussions on Hash Partitioning

The graph in figure 19 shows the performance difference for CITY table when hash partitioned and not partitioned. The blue line, which is when CITY table is not partitioned, shows a sharp increase in time as the size of data in CITY table rises. The second line is for CITY table when hash partitioned into different partitions. This line also shows when the number of data in the table increases, the amount of time spent to fetch a single row also increases. But, when compared with the one when CITY table is not hash partitioned that progression is smaller.

For instance, when tried to pull a single row from CITY table with 3 million data and not partitioned, it took 12 seconds and 669 milliseconds. For the same amount of data but hash partitioned into different partitions, it took 4 seconds and 494 milliseconds. For CITY table with 18 million records inserted and not partitioned, retrieving a single row lasted for more than 5 minutes. But when hash partitioned into different partitions, the time dramatically decreased to less than a minute for a single row fetch.

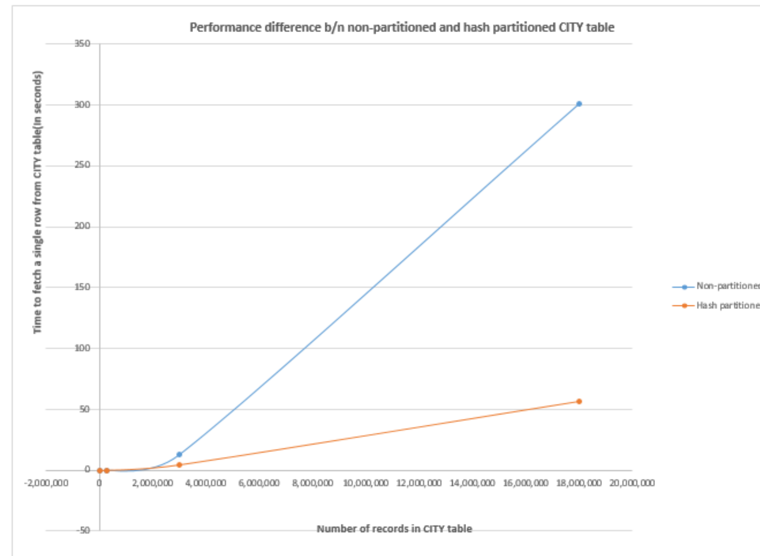


Figure 19: Performance difference between non-partitioned and Hash partitioned CITY table

5.3 Results and Discussions on List Partitioning

The graph in figure 20 shows the different performance levels when selecting a single row from CITY table for the number of records ranging from 30,000 to 18 million.

Figure 20: Performance difference between non-partitioned and List partitioned CITY table

According to this graph in figure 20, there is no much difference in performance whether CITY table with 30,000 and 300,00 records is list partitioned or not. But the effect of list partitioning on CITY table was observed when the number of records in the table was increased to 3 million and beyond. For example, for CITY table with 18 million records

and list partitioned, it took 25 seconds and 122 milliseconds less time to retrieve a row from the table than when CITY table was not partitioned.

6 Conclusion

In this thesis, it was planned to test and examine the different techniques of Oracle partitioning ranging from 30,000 records in a table up to 300 million worth of data inserted to the table. The testing and examining of the different Oracle partitioning techniques was limited to the size of the table having 18 millions rows. The main problem was that there was no enough space available to accumulate all that much data.

Regardless of the space issue, different performance behaviors were observed when fetching data from the table which was the whole purpose of this thesis. The three Oracle partitioning techniques tested in this thesis were range, hash and list partitioning. The four groups of data tested in this thesis were a table with 30,000, 300,000, 3 million and 18 million rows. In most of the cases, these three Oracle partitioning techniques showed no difference in performance whether a table with 300,000 records or less was partitioned or not.

But, when the three Oracle partitioning techniques were applied to a table with 3 million records and more, a performance difference between the partitioned and non-partitioned table was observed. The difference in performance when data was fetched from both partitioned table and non-partitioned table was a gain in performance. For example, when CITY table with 18 million rows was hash partitioned into four different partitions fetching a single row took around 4 minutes less time, which was 5.3 faster, than when not partitioned.

Therefore, Oracle partitioning makes a big difference in terms of performance, availability and manageability when the number of records in a table is big. According to the tests done in this thesis, it is not necessary to partition a table if its size is below 2.19 MB. But if the size of the table has reached up to 323 MB and beyond, it is recommended to partition the table for manageability, availability and performance purposes.

In this thesis, the number of records in a table that are evaluated for Oracle partitioning

techniques was limited to 18 million rows. So, it would be interesting to continue with the testing for records of 30, 300 million and more. Creating and partitioning global index for performance was not tested either. It would be a good idea to continue the test on how global index would change the performance of an Oracle database. The number of partitions created for partitioning test were only four. It would also be a good idea to investigate on how the database's performance and manageability would be affected if the number of partitions are increased to more than four partitions.

References

1 Oracle Help Center. Partitioning Concepts. oracle.com; 2015. Available from: http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm [cited June 16, 2015].

2 Al Shehri W. CLOUD DATABASE Database as a Service. International Journal of Database Management Systems (IJDMS). 2013;5(2):1–2. Available from: http://airccse.org/journal/ijdms/papers/5213ijdms01.pdf [cited June 16, 2015].

3 Oracle Help Center. Oracle® Database VLDB and Partitioning Guide 11g Release 2 (11.2). oracle.com; 2015. Available from: http://docs.oracle.com/cd/E18283_01/server.112/e16541/intro.htm [cited May 12, 2015].

4 Burleson D. Partitioning an Oracle table Tips. Burleson Consulting; 2015. Available from: http://www.dba-oracle.com/t_partitioning_tables.htm [cited June 22, 2015].

5 Carr A. The Advantages and Disadvantages of Oracle Partition. ehow; 2015. Available from: http://www.ehow.com/info_8723134_advantages-disadvantages-oracle-partition.html [cited June 22, 2015].

6 Oracle Help Center. Oracle Partitioning with Oracle Database 12c. oracle.com; 2014. Available from: http://www.oracle.com/technetwork/database/options/partitioning/partitioning-wp-12c-1896137.pdf [cited June 24, 2015].

7 Powell G. Beginning Database Design. Indianapolis, Indiana USA: Wiley Publishing, Inc; 2006.

8 Oracle Help Center. Database VLDB and Partitioning Guide. oracle.com; 2015. Available from: https://docs.oracle.com/database/121/VLDBG/title.htm [cited June 30, 2015].

9 Greenwald R, Stackowiak R, Stern Y. Oracle Essentials. Sebastopol, California USA: O'Reilly; 2013.

10 Kuhn D. Pro Oracle Database 12c Administration. New York, USA: Apress; 2013.

11 Kuhn D, Alapati SR, Padfield B. Expert Indexing in Oracle Database 11g. New York, USA: Apress; 2012.

12 Husqvik. Is there a better query/code that will take less time to insert these data? Stackoverflow; 2015. Available from: http://stackoverflow.com/questions/32169642/is-there-a-better-query-code-that-will-take-less-time-to-insert-these-data/32185501#32185501 [cited August 30, 2015].

1 Complete Source Codes for Range Partitioning Tables

```
1  create table city
2  (cityid numeric(20),
3  cityname varchar2(30),
4  cityregion varchar2(30),
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  partition by range (cityid)(
8  partition A1 values less than (7500) tablespace system,
9  partition A2 values less than (15000) tablespace sysaux,
10 partition A3 values less than (22500) tablespace users,
11 partition A4 values less than (maxvalue) tablespace
       LARGE_TBL);
12 declare
13 c_id  number := 0 ;
14 c_name varchar2(30) ;
15 c_region varchar2(30);
16 c_pop varchar2(30);
17 cityid_copy city.cityid%TYPE;
18 time_before timestamp;
19 time_after timestamp;
20 begin
21 while (c_id <= 30000)
22 loop
23 c_name     := 'City' || c_id || sysdate;
24 c_region   := 'Region' || c_id || sysdate;
25 c_pop      := c_id + 500 || sysdate;
26 insert into city
27 (cityid,cityname,cityregion,citypopulation)
28 values(c_id,c_name,c_region,c_pop);
29 c_id := c_id +1;
```

```
30 end loop;
31 end;
```

Listing 16: Populating and range partitioning a newly created table with data

```
1 drop table CITY;
2 create table city
3 (cityid numeric(20),
4 cityname varchar2(30),
5 cityregion varchar2(30),
6 citypopulation varchar2(30),
7 constraint city_pk primary key (cityid))
8 declare
9 c_id  number := 0 ;
10 c_name varchar2(30) ;
11 c_region varchar2(30);
12 c_pop varchar2(30);
13 cityid_copy city.cityid%TYPE;
14 time_before timestamp;
15 time_after timestamp;
16 begin
17 while (c_id <= 3000000)
18 loop
19 c_name     := 'City' || c_id || sysdate;
20 c_region   := 'Region' || c_id || sysdate;
21 c_pop      := c_id + 500 || sysdate;
22 insert into city
23 (cityid,cityname,cityregion,citypopulation)
24 values(c_id,c_name,c_region,c_pop);
25 c_id := c_id +1;
26 end loop;
27 end;
```

Listing 17: Populating a newly created table with data

```
1 create table city
2 (cityid numeric(20),
3 cityname varchar2(30),
```

```
4  cityregion varchar2(30),
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  partition by range (cityid)(
8  partition A1 values less than (75000) tablespace system,
9  partition A2 values less than (150000) tablespace sysaux,
10 partition A3 values less than (225000) tablespace users,
11 partition A4 values less than (maxvalue) tablespace
      LARGE_TBL);
12 declare
13 c_id  number := 0 ;
14 c_name varchar2(30) ;
15 c_region varchar2(30);
16 c_pop varchar2(30);
17 cityid_copy city.cityid%TYPE;
18 time_before timestamp;
19 time_after timestamp;
20 begin
21 while (c_id <= 300000)
22 loop
23 c_name    := 'City' || c_id || sysdate;
24 c_region  := 'Region' || c_id || sysdate;
25 c_pop     := c_id + 500 || sysdate;
26 insert into city
27 (cityid,cityname,cityregion,citypopulation)
28 values(c_id,c_name,c_region,c_pop);
29 c_id := c_id +1;
30 end loop;
31 end;
```

Listing 18: Populating and partitioning a newly created table with data

```
1 create table city
2 (cityid numeric(20),
3 cityname varchar2(30),
4 cityregion varchar2(30),
```

```
5  citypopulation varchar2(30),
6  constraint city_pk primary key (cityid))
7  partition by range (cityid)(
8  partition A1 values less than (750000) tablespace system,
9  partition A2 values less than (1500000) tablespace sysaux,
10 partition A3 values less than (2250000) tablespace users,
11 partition A4 values less than (maxvalue) tablespace
      LARGE_TBL);
12 declare
13 c_id  number := 0 ;
14 c_name varchar2(30) ;
15 c_region varchar2(30);
16 c_pop varchar2(30);
17 cityid_copy city.cityid%TYPE;
18 time_before timestamp;
19 time_after timestamp;
20 begin
21 while (c_id <= 3000000)
22 loop
23 c_name     := 'City' || c_id || sysdate;
24 c_region   := 'Region' || c_id || sysdate;
25 c_pop      := c_id + 500 || sysdate;
26 insert into city
27 (cityid,cityname,cityregion,citypopulation)
28 values(c_id,c_name,c_region,c_pop);
29 c_id := c_id +1;
30 end loop;
31 end;
```

Listing 19: Populating and partitioning a newly created table with data

2 Complete Source Codes for Hash Partitioning Tables

```
1  begin
2  drop table CITY;
3  create table city
4  (cityid numeric(20),
5  cityname varchar2(30),
6  cityregion varchar2(30),
7  citypopulation varchar2(30)),
8  partition by hash (cityid)
9  partitions 4
10 STORE IN (system, sysaux, users, LARGE_TBL);
11 declare
12 c_id  number := 0 ;
13 c_name varchar2(30);
14 c_region varchar2(30);
15 c_pop varchar2(30);
16 cityid_copy city.cityid%TYPE;
17 time_before timestamp;
18 time_after timestamp;
19 while (c_id <= 300000)
20 loop
21 c_name    := 'City' || c_id || sysdate;
22 c_region  := 'Region' || c_id || sysdate;
23 c_pop     := c_id + 500 || sysdate;
24 insert into city
25 (cityid,cityname,cityregion,citypopulation)
26 values(c_id,c_name,c_region,c_pop);
27 c_id := c_id +1;
28 end loop;
29 end;
```

Listing 20: Populating and partitioning a newly created table with data

3 Complete Source Codes for List Partitioning Tables

```
1  create table city (
2  cityid number not null,
3  cityregion varchar2(100) not null,
4  state varchar2(2) not null,
5  population varchar2(255))
6  partition by list(state)
7  (
8  partition partion1 values ('NY','NJ') tablespace system,
9  partition partion2 values ('CA', 'TX','WA') tablespace
       sysaux,
10 partition partion3 values ('NC', 'IL') tablespace users,
11 partition partion4 values ('KS','SC','NV') tablespace
       large_tbl) nologging;
12 alter session force parallel dml;
13 insert /*+ append */ into city
14 with states as (
15 select 'NY' state, 1 id_offset from dual union all
16 select 'NJ' state, 2 id_offset from dual union all
17 select 'CA' state, 3 id_offset from dual union all
18 select 'TX' state, 4 id_offset from dual union all
19 select 'WA' state, 5 id_offset from dual union all
20 select 'NC' state, 6 id_offset from dual union all
21 select 'IL' state, 7 id_offset from dual union all
22 select 'KS' state, 8 id_offset from dual union all
23 select 'SC' state, 9 id_offset from dual union all
24 select 'NV' state, 10 id_offset from dual),
25 generator as (
26 select /*+ materialize cardinality(5000000) */ (level - 1) *
        10 id from dual connect by level <= 3000)
27 select id + id_offset, 'City' || (id + id_offset) || sysdate
```

```
      , state, id + 500 || sysdate
28 from generator cross join states;
```

Listing 21: Creating a table with List partition (Copied from Stackoverflow (2015) [12])