CHAPTER 6

PARTITION ALGORITHM APPROACH FOR MINING FREQUENT PATTERNS

6.1 Introduction

In 1997, Partitioning was first introduced in Oracle 8.0 and it is the most important and successful functionality of the Oracle database. Partition is a way to split tables, indexes, and index-organized tables into smaller pieces called partitions which enables the database objects to be managed and accessed at a finer level of granularity. Each partition is known by its specific name and has its own characteristics such as its storage and index.

To illustrate partitioning methodology, suppose an HR manager has one big box that contains employee folders. Each folder specifies the employee hire date. Queries are made often for employees hired in a particular month. One solution is to create an index on employee hire date that specifies the locations of the folders scattered throughout the box. Another solution is partitioning strategy, which uses many smaller boxes, with each box containing folders for employees hired in a given month. Using smaller boxes has several advantages.
To retrieve the folders for employees hired in June, the HR manager can retrieve the June box. If any small box is temporarily damaged, the other small boxes can be used. Shifting offices from one place to another place becomes easier because instead of moving a single heavy box, smaller boxes can be moved very easily.

Partitioning is important, when the table size is greater than 2 gigabytes, tables contain historical data and the contents of a table need to be distributed across different types of storage devices. Partitioning is a tool for building multi-terabyte systems with extreme high availability requirements. It is time efficient because it allows the maintenance and failure operation on a particular partition to be carried out on selected partitions while other partitions are available to users so that it improves the performance of operation.

Partition independence allows for concurrent use of the various partitions for various purposes. The main advantage is that it reduces the total cost of data ownership by maintaining the old relevant information in a compressed format and improves the performance, manageability, reduced contention for shared resources and increases the availability.

To access the partitioned tables, no need to change the SQL queries and DML statements. After partitions have been defined, DDL statements can access and manipulate the particular partitions rather than entire tables or indexes, thus the partitioning can simplify the manageability of large database objects.
Partitioning is entirely transparent to applications. Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints, but each partition can have separate physical attributes such as pctfree, pctused, and tablespaces. Partitioning is useful for many different types of application that manage large volumes of data.

From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed individually or collectively and have flexibility in managing a partitioned object. From the perspective of the application, no modification is necessary when accessing a partitioned table using SQL DML commands; a partitioned table is identical to a non-partitioned table.

The partition key is a set of one or more columns, which determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition. Using Partition Key, Partitioning is done. A partitioned table consists of one or more partitions, which are managed individually and can operate independently of the other partitions.

A table is either partitioned or nonpartitioned. Even if a partitioned table consists of only one partition, this table is different from a nonpartitioned table, which cannot have partitions added to it. Data stored in the table is in the compressed form. Compression increases the speed and saves the space. A partitioned index is an index that, like a partitioned table, has been divided into smaller and more manageable pieces.
Global indexes are partitioned independently of the table on which they are created, whereas local indexes are automatically linked to the partitioning method for a table. Partitioned index has two options: local partitioned index, and global partitioned index. The local partitioned index is divided into local prefixed index and local nonprefixed index. In a local partitioned index, the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as its table.

Each index partition is associated with exactly one partition of the underlying table, so that all keys in an index partition refer only to rows stored in a single table partition. The database automatically synchronizes index partitions with their associated table partitions, making each table-index pair independent. The advantages of local partitioned indexes are, availability is increased and easy partition maintenance.

In Local prefixed indexes, the partition keys are on the leading edge of the index definition. In Local nonprefixe indexes, the partition keys are not on the leading edge of the indexed column list and need not be in the list at all. Both types of indexes can take advantage of partition elimination, which occurs when the optimizer speeds data access by excluding partitions.

Depending on query predicate, a query can eliminate the partitions. A query that uses a local prefixed index, always allows for index partition elimination, whereas a query that uses a local nonprefixed index does not allow index partition elimination.
A global partitioned index is a B-tree index on a partitioned or non-partitioned table that is partitioned independently of the underlying table on which it is created. Global-partitioned indexes are partitioned using range or hash partitioning and are uncoupled from the underlying table. Global indexes are useful for OLTP applications, where rapid access, data integrity, and availability are important. This is partitioned using range or hash partitioning and is uncoupled from the underlying table.

For example, a table could be range-partitioned by month and have twelve partitions, and using a different number of partitions, an index on that table could be hash-partitioned. Decoupling an index from its table automatically means that any partition maintenance operation on the table can potentially cause index maintenance operations and it is most commonly used for OLTP than for data warehousing environments.

A global non-partitioned index is an index on a non-partitioned table. The index structure is not partitioned and uncoupled from the underlying table. In data warehousing environments, the most common usage of global non-partitioned indexes is to enforce primary key constraints. OLTP environments rely on global non-partitioned indexes.

If all the index created on all partitions of a partitioned table, then it is called full indexing. If the entire index created only on a subset of the partition of a partition table, then it is called partial indexing. Partial indexing is an index attribute, only applicable to indexes on partitioned tables, and is complementary to the standard full indexing available for partitioned and nonpartitioned tables.
The main advantages are, it improves the load and query performance. The appropriate indexing strategy is chosen based on the business requirements and access patterns, making partitioning well suited to support any kind of application.

### 6.2 Partitioning Strategies

Oracle Partitioning provides three fundamental data distribution methods as basic partitioning strategies that control how data is placed into individual partitions. They are range, hash and list.

Table can be partitioned according to the data distribution methods as a single-level partitioning and composite level partitioning. The single level partitioning uses only one of method of data distribution like range or hash or list in one or more columns as the partitioning key.

The composite partitioning is a grouping of two data distribution methods which is used to define a composite partition table. Initially, a table is partitioned by one data distribution method, and then each partition is further divided into sub partitions using a second data distribution method. Partitions of a composite partitioned table are metadata, and it will not represent the actual data storage.


6.2.1 **Range Partitioning**

In range partitioning, based on a range of values of the partitioning key the data is distributed. Ranges are always defined as an excluding upper boundary of a partition, and the lower boundary of a partition is automatically defined by the exclusive upper boundary of the preceding partition and partition boundaries are always increasing.

Each partition has values less than clause, which specifies a upper bound for the partitions. Any values of the partitioning key, which are equal to or higher than this literal, are added to the next higher partition.

For example, for a year column as the partitioning key, the '2013' partition contains rows with the partitioning-key values between '01-Jan-2013' and '31-Dec-2013’. Since it is showing the range of values, it comes under range partitioning. When the user is putting the request for 30-Jun-2013, then the control will move on to this partition, to pick the data instead of searching the whole database.

6.2.2 **List Partitioning**

In list partitioning, the database uses a list of discrete values as the partition key for each partition. List partitioning is used to control how individual rows map to specific partitions, and can group the related sets of data when the key used to identify them is not conveniently ordered.
List partitioning, specifies, for a region column as the partitioning key, the 'India' partition may contain values 'Chennai', 'Mumbai' and 'Delhi'. To catch all values for a partition key that are not explicitly defined by any of the lists, a special default partition can be defined.

### 6.2.3 Hash Partitioning

In hash partitioning, an internal hash algorithm is applied to the partitioning key to determine the partition for a given partition key. Hash does not provide any logical mapping between the data and any other partition, but it provides equally balanced sizes of the partitions. This is the main advantage of hash partitioning.

The hashing algorithm is designed to distribute rows evenly across devices so that each partition contains about the same number of rows. Hash partitioning is useful for dividing large tables into several small pieces to increase manageability.

The composite partition techniques are further divided into range-hash, range-list, range-range, list-range, list-list, list-hash, hash-hash, hash-range, and hash-list. Composite Range-Range Partitioning specifies the logical range partitioning along two Dimensions. For example, partition by kids age and range subpartition by rate.
Composite Range-Hash Partitioning specifies the partitions data using the range method, and within each partition, using hash method, sub partitions it. Composite range-hash partitioning provides the improved manageability of range partitioning and hash partitioning. The advantages are, data placement, striping, and parallelism.

Composite Range-List Partitioning specifies the partitions data using the range method, and within each partition, using list method subpartitions it. Range partition by item rate and list subpartition by country_id is the example of composite range list partitioning. Composite range-list partitioning provides the manageability of range partitioning and the explicit control of list partitioning for the subpartitions.

Composite list-range partitioning enables logical range subpartitioning methodology within a given list partitioning strategy. List partition by country_id and range subpartition by item_rate is the example of composite list-range partitioning. Composite list-hash partitioning specifies hash subpartitioning of a list-partitioned object.

Composite list-list partitioning enables logical list partitioning along two dimensions. List partition by country_id and list subpartition by items is the example of composite list-list partitioning. To enhance the usage of the basic partitioning strategies, Oracle provides partitioning extensions.
6.2.4 Partitioning Extensions

Partitioning extensions enhance the manageability of partitioned objects and for defining the partitioning key of a table. The groups of tables that are logically connected through parent-child relationships become more flexible. Partitioning extensions are interval partitioning, reference partitioning, virtual column-based partitioning and partition advisor.

Interval partitioning is an extension of range partitioning method, which defines the equi-partitioned ranges for any future partitions, using an interval definition as part of the table metadata. Even the partitioned table is initially created only with one partition.

An interval partitioned table can automatically extend up to the maximum of 1048575 partitions without any user intervention. Oracle will create any new partition automatically whenever it is needed and data for such a partition is inserted for the very first time. The advantage of Interval partitioning is to improve the manageability of a partitioned table.

An interval partitioned table could be defined so that Oracle creates a new partition for every day in a calendar year; for instance, a partition is automatically created for September, 19th 2013' as soon as the first record for this day is inserted into the database.
The interval partitioned techniques that are currently available are Interval-List, Interval-Hash, and Interval-Range. Interval partition is not supported for index-organized tables and cannot create domain index on an interval partitioned table.

Reference partitioning allows the partitioning of two tables related to one another by leveraging an existing parent-child relationship. Without the necessity to store the parent's partitioning key columns in the child table, the primary key-foreign key relationship is used to inherit the partitioning strategy of the parent table to its child table.

The parent and child table partition strategy becomes identical. For every partition in the parent table, there is exactly one partition in the child table, and through the primary key-foreign key relationship, the child partitioning strategy is defined.

All child records of a given primary key value are stored in the “same” partition of the child table. If a user want to take advantage of the same partitioning strategy as parent and, if there is no reference partitioning then duplicate all partitioning key columns from the parent table to the child table.

The main advantage is, without duplication of the partitioning key columns, reference partitioning allows you to naturally leverage the parent-child relationship, and thus it reduces the manual overhead and saves the memory space. The disadvantage of this method is index-organized tables are not supported by reference partitioning.
In virtual column-based partitioning, using one or more existing columns of a table, the virtual columns allows the partitioning key to be defined by an expression, and also enables the more comprehensive match of the business requirements; business attributes can be used to define the partitioning strategy of an object.

Usually, the columns are overloaded with more information; for example a 8 digit account number of the bank include the account branch information like, which branch, which area, etc. Virtual column partitioning does not support Index-organized partition.

In Partition Advisor, the SQL Access Advisor supports partitioning for indexes, materialized views, materialized view logs, and shows the improvement in the performance. Recommendations generated by the SQLAccess Advisor will show the anticipated performance gains that will result if they are implemented.

The generated script can either be implemented manually or submitted onto a queue within Oracle Enterprise Manager. With the extension of partitioning advice, customers not only can get recommendation specifically for partitioning, but also a more comprehensive holistic recommendation of SQL Access Advisor, improving the collective performance of SQL statements overall.
Oracle supports the combination of the partitioning extensions: Interval Partitioning and Reference Partitioning, combination of Reference Partitioning with both virtual column-based Partitioning and Interval Partitioning, and at last combination of virtual column-based Partitioning with both Reference Partitioning and Interval Partitioning.

Without a specific partition key, system partitioning provides the ability to implement and manage new partitions. Using the extended partitioning syntax for system partitions, each partition is mapped to a tablespace. The database simply provides the ability to break down a table into partitions without knowing what the individual partitions are going to be used for. All aspects of partitioning have to be controlled by the application.

The drawback of system partitions are that the performance benefits available for partitioned tables do not exist, due to lack of partition keys, and the system partitions cannot be used for partition-wise joins or traditional partition pruning operations. The advantage is system partitioning is used for manageability purposes.

6.3 Partitioning Performance, Manageability and Availability

Partitioning improves the performance, manageability and availability. The performance benefits are partition pruning, partition wise joins, and parallel DML. Partition pruning is the skipping of unnecessary index and data partitions or subpartitions in a query. In partition pruning, oracle database server recognizes the partitions and subpartitions, and eliminates the unnecessary partitions or subpartitions from access by SQL statements.
The unwanted partitions or subpartitions can be eliminated depending on the selection criteria given in the query. For example, if a query only involves dresses of the girl baby, then there is no need to retrieve information about men dresses, women dresses etc. Such intelligent knowledgeable pruning can dramatically reduce the data volume and improve the query performance.

The optimizer cannot eliminate the partitions if the SQL statement applies a function to the partitioning column. If the SQL statement applies a function to the indexed column, the optimizer cannot use an index, unless it is a function-based index.

Pruning can eliminate index partitions even when the underlying table’s partitions cannot be pruned. Performance of the large table operations can be improved by creating partitioned indexes that reduces the amount of data the user wants to access or modify.

Consider a sale table, which has been partitioned by day. A query, requesting sales for a single week, would only access seven partitions of the sales table. If the table had 2 years of historical data, this query would access seven partitions instead of 520 partitions. This query could execute 100 times faster because of partition pruning.

Partition-wise joins can be applied when two tables are joined together, and on the join key one of these tables is partitioned. With partition-wise joins, the join operation is broken into smaller joins of identical data sets, that are performed sequentially or in parallel.
Partition-wise joins minimize the amount of data exchanged among parallel slaves during the execution of parallel joins by taking into account data distribution and completes the overall join in less time. So the performance of multi table joins have been improved using partition wise joins.

In decision support systems and data warehouses, parallel execution dramatically reduces response time. In conventional tables, using of parallel DML, and parallel query with range and hash partitioned tables, enhances scalability and performance for batch operations. When the index-organized tables have been used or not, the semantics and restrictions for parallel DML sessions are the same.

Partitioning allows tables and indexes to be partitioned into smaller, more manageable units. It allows database administrators with the ability to pursue a "divide and conquer" approach to data management. SQL commands in oracle are used for managing partitioned tables, and that includes adding new partitions, splitting, moving merging, truncating and exchanging partitions.

With partitioning, maintenance operations can be focused on particular portions of tables. A database administrator could compress a single partition of a table, rather than compress the entire table. These operations have to be performed on a per partition basis, thus dividing the maintenance process into more manageable chunks.
Partitioning for manageability, support a rolling window load process in a data warehouse and also give support when a DBA loads new data into a table on a weekly basis. That table could be partitioned so that each partition contains one week of data.

Using partition exchange load, new partition will be added and is known as load process. Adding a single partition is much more efficient than modifying the entire table, and removing data from the partitioned table by drop or truncate in a efficient manner is another key advantage of partitioning. Partitioned database objects provide partition independence.

For high availability, this characteristic of partition independence is important. For example, if a table contains 10 partitions, five partition of a partitioned table is unavailable, and then, all of the other five partitions of the table remain online and available.

The application can continue to execute queries with the transactions against the available partitions of the table, and it will run successfully, provided if application do not need to access the unavailable partition of the table. The database administrator can specify that each partition is stored in a separate tablespaces; the most common scenario is having these tablespaces stored on different storage tiers.
Different partitions are stored in different tablespaces; enable the database administrator to do backup and recovery operations on each individual partition, independent of the other partitions in the table.

Partitioning can reduce scheduled downtime and performance gains provided by partitioning may enable database administrators to complete maintenance operations on large database objects in relatively small batch windows.

6.4 Mining Frequent Patterns using Partition Algorithm

A partition concept has been proposed to increase the execution speed with minimum cost. For each itemsets, that is one itemset, 2-itemsets, 3-itemsets etc., a separate partition will be created during data insertion into the table. Initially a set of frequent 1-itemsets is found by scanning the databases and get the numbers of occurrences of each item from the partition of those particular items using the pointer, the items satisfying the minimum support count will be included in the frequent 1-itemsets L_1. Like Apriori algorithm L_1 is used to find L_2, the set of L_2 is used to find L_3, and so on, until no more frequent k-itemsets can be found.

To find, L_k, it is not necessary to scan the full database; it is enough to search the count of each data itemsets from its partition. Initially to generate frequent itemsets, an important property called the Apriori property used for reducing the search space. Join and Prune are two-steps to find the frequent itemsets. In join, L_k is find from a set of candidate k-item sets C_k which is generated by joining L_{k-1} with itself.
In pruning, to find the count of each candidate in $C_k$, the partition of each itemsets will be checked and the count which is not less than minimum support count are frequent and belongs to $L_k$. To reduce the size of $C_k$, the apriori property is used. The performance of partition algorithm for finding the frequent data items is efficient when compared to other existing Algorithms.

Table 6.1 Transaction Database for Partition Algorithm

<table>
<thead>
<tr>
<th>TID</th>
<th>List of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>I1, I2, I4</td>
</tr>
<tr>
<td>T2</td>
<td>I2, I4</td>
</tr>
<tr>
<td>T3</td>
<td>I3, I4</td>
</tr>
<tr>
<td>T4</td>
<td>I1, I2, I5</td>
</tr>
<tr>
<td>T5</td>
<td>I1, I4</td>
</tr>
<tr>
<td>T6</td>
<td>I1, I3, I5</td>
</tr>
<tr>
<td>T7</td>
<td>I1, I2, I3, I5</td>
</tr>
<tr>
<td>T8</td>
<td>I1, I2, I3</td>
</tr>
</tbody>
</table>

Consider the Database D, with seven transactions, here partitioning has been used to store and retrieve the data from the database. Different types of partitioning techniques are available, in this thesis; range partitioning has been used to increase the performance when mining the frequent patterns in the database.
Table 6.1 shows a transaction database for partition algorithm and it contains 8 transactions. In this table, transaction T1 contains I1, I2, I3 and transaction T2 contains I2, I4 and so on.

From the candidate 1 itemset $C_1$ generate the frequency 1 itemset $L_1$ by calculating the no of occurrences of the data items directly from the partition instead of scanning the whole database. Here the minimum support count taken is 2. Candidate 1 itemset which is satisfying the minimum support count will be include in $L_1$. The following figure 6.1. shows, the generation of candidate 1 itemsets and frequent 1 itemsets.

![Figure 6.1 Generation of Frequent 1- Itemset using Partition Algorithm](image)

The candidate 2 itemsets are generated by joining $L_1$ by itself and check whether the subset of the frequent itemsets are also frequent, since in $L_1$ all the items have been included, there will be no pruning.
For calculating the support count, instead of scanning the whole database, it is enough to get the count from the appropriate partition. The candidate 2 itemsets which is satisfying the minimum support count will be included in the frequent 2 itemsets $L_2$. The following figure 6.2 shows the generation of frequent 2 itemsets using partition. Finally 8 frequent 2 itemsets have been generated, and this has been used to generate the candidate 2 itemsets.

![Figure 6.2 Generation of Frequent 2-Itemset using Partition Algorithm](image)

The candidate 3 itemsets are generated by joining $L_2$ by itself and find whether the subset of the frequent itemsets are also frequent, only the itemsets $\{I1,I2,I4\},\{I1,I2,I5\},\{I1,I2,I3\}$ and $\{I1,I3,I5\}$ have been included for the next step that is considered as candidate 3 itemsets since whose subset is also a frequent itemset.
The remaining itemsets have been removed because whose subset is not frequent one. For calculating the support count, instead of scanning the whole database, it is enough to get the count from the appropriate partition. The candidate 3 itemsets which is satisfying the minimum support count will be included in the frequent 3 itemsets L3.

The following figure 6.3 shows the generation of frequent 3 itemsets using partition. Finally 3 frequent 3 itemsets have been generated, and this has been used to generate the candidate 3 itemsets.

![Figure 6.3 Generation of Frequent 3 - Itemset using Partition Algorithm](image)

The candidate 4 itemsets are generated by joining L3 by itself and check whether the subset of the frequent itemsets are also frequent, only the itemsets \{I1, I2, I3, I5\} have been included for the next step that is considered as candidate 4 itemsets since whose subset is also a frequent itemset.
For calculating the support count, directly get the count from the partition, instead of scanning the whole database. Since the support count is 1, which is not satisfying the minimum support, it is pruned and $L_4 = \emptyset$ and algorithm terminates. With the frequent 3 itemsets, association rule is applied which shows the association between the data items.

### 6.5 Implementation and Result

The partition algorithm has been implemented using PL/SQL in Oracle Database and then compared with Apriori algorithm. Table 6.2 shows the performance evaluation between Apriori and Partition algorithm. The algorithms are compared with different sets of record like 1000, 2500, 5000 etc.

**Table 6.2. Performance Evaluation of Apriori and Partition**

<table>
<thead>
<tr>
<th>ALGORITHMS</th>
<th>TOTAL NO OF RECORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>APRIORI</td>
<td>8 min</td>
</tr>
<tr>
<td>PARTITION</td>
<td>30 sec</td>
</tr>
</tbody>
</table>
When the no of records is 1000, the Apriori algorithm takes 8 minutes and partition algorithm takes 30 secs to find the frequent data items sets. When the total no of records is 2500, the partition algorithm has taken 1.5 min and Apriori algorithm taken 20 min and when the total no of records is 5000, Apriori algorithm taken 42 minutes and partition algorithm takes 3 minutes.

In this way, the partition algorithm has better performance over Apriori algorithm.

![Figure 6.4 Performance Evaluation Graph of Partition Algorithm](image-url)
Figure 6.4 shows the generation of frequent itemset and association rule generation graph and by using the table 6.2, graph have been plotted for Apriori and partition algorithms. In this graph, total no of records has taken in the x-axis, and the execution time has taken as y-axis. Graphical representation shows, partition algorithm has efficient performance over the Apriori algorithm.