ISSN No. 0976-5697

Volume 6, No. 8, Nov-Dec 2015

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Selection of a Suitable Partitioning Strategy for Multidimensional Database

Dr Anil Rajput
Professor Computer Science & Mathematics
Govt.C.S.A.P.G.College, Sehore, Madhya Pradesh

Kshmasheel Mishra
Former Reader Computer Science
Vikram University,Ujjain,Madhya Pradesh

Vaibhav Khanna*
Assistant Professor Dezyne E'cole College,
Civil lines, Ajmer, Rajasthan

Purushottam sharma
A-35,Vidhya Nagar, Hosangabad Road,
Bhopal(M.P.)

Abstract: **T**aking into consideration the business and practical point of view, a partitioning strategy is normally recognized with a practical goal-seeking perspective. Therefore it needs to be mapped to an Oracle partitioning technical recommendation or specific partitioning strategy matching those business requirements, regulatory compliance, or systems platform. A persisting challenge is the determination of a suitable number of partitions in each dimension.
Assumptions and limitations of current data partitioning and placement in database machines have been discussed. A new formula defining the number of partitions accessed by range queries in a single dimension when the number of partitions is small has been derived. A model relating analytics savings due to partitioning and the resultant costs to transaction processing from the partitioning has been presented which shows how to determine the columns to partition, and to what level. Both single and multi-column partitioning has been considered.

Keywords: Partitioning Strategies, performance, partitioning, interval partitioning, range partitioning, hash partitioning, database performance experiments.

I. INTRODUCTION

Modern enterprises recurrently run mission-critical databases containing upwards of several hundred gigabytes, and often several terabytes of data. These enterprises are challenged by the support and maintenance requirements of very large databases (VLDB), and must devise methods to meet those challenges. Data used to be just data. Now there's "big data," real-time data, multi-structured data, analytic data, and machine data. Likewise, user communities have swollen into thousands of coexisting users, reports, dashboards, scorecards, and analyses. The rising popularity of advanced analytics has driven up the number of power users with their titanic ad hoc queries and analytic workloads. And there are still brave new worlds to explore, such as social media and sensor data.

Partitioning addresses key issues in supporting very large tables and indexes by decomposing them into smaller and more manageable pieces called partitions, which are entirely transparent to an application. SQL queries and Data Manipulation Language (DML) statements do not need to be modified to access partitioned tables. However, after partitions are defined, Data Definition Language (DDL) statements can access and manipulate individual partitions rather than entire tables or indexes. This is how partitioning can simplify the manageability of large database objects.

This research paper focuses on partitioning strategy options and their fitment for use. The experiments were conducted for actual data sets for real life partitioning option as included in commercial databases. The broad areas of this set of experiments included various strategies to Table Partitioning and Sub partitioning.

Partitioning offers several advantages at various stages of data management operations such as data loads, index creation and rebuilding, and backup and recovery at the partition level, rather than on the entire table. As a result of this there is a significant reduction in times for these operations. On the core it is a "Divide and Conquer" technique which can be applied at maintenance level[7]. Through careful selection of partitioning strategy we can have more granular storage allocation options including online, offline, rebuild, reorganize and object level backup/restore options. Partitioning works at the optimizer level and the optimizer eliminates (prunes) partitions that do not need to be scanned (Partition Pruning). Similarly join operations can be optimized to join "by the partition" (Partition-Wise Joins). Also the partitions can be load-balanced across physical devices and this significantly reduces the impact of scheduled downtime for maintenance operations.

Partition independence for partition maintenance operations can help us in performing coexisting maintenance operations on different partitions of the same table or index.

A partition can be divided at a user-defined value and can isolate subsets of rows that must be treated individually. This means that SELECT, UPDATE, INSERT and DELETE operations can be applied on a partition level instead of a table level, which results in huge performance improvements. One can also run coexisting SELECT and DML operations against partitions that are unaffected by maintenance operations. It increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance[5]. Parallel execution provides specific advantages to optimize resource utilization, and minimize execution time. Parallel execution against partitioned objects is key for scalability in a clustered environment. Parallel execution is supported for queries and for DML and DDL.

A table is defined by specifying one of the following data distribution methodologies, using one or more columns as the partitioning key:

- Range Partitioning
- Hash Partitioning
- List Partitioning

An important factor affecting query performance in multidimensional datasets / tables is the partitioning strategy which determines the tables to be partitioned and the partitioning attribute. For evaluation purposes, we categorized query optimizers into three categories—Basic, Intermediate, and Advanced—based on how they exploit partitioning information to perform optimization[2]. The intermediate query optimisers can conduct per-table partition pruning and one-to-one partition-wise joins (like Oracle and SQLServer).

II. PARTITIONING STRATEGY EXPERIMENTS

We conducted several experiments to evaluate the effectiveness of partitioning techniques across a wide range of factors that affect table partitioning. We used the TPC-H benchmark with scale factors ranging from 10 to 40, with 30 being the default scale. We closely followed directions from the TPC-H Standard Specifications for multidimensional database partitioning. To align to the specification guideline we partitioned tables only on primary key, foreign key, and/or date columns. The following section present the experimental results for a representative set of TPC-H queries. The experimental readings were taken several times and the results presented are averaged over five to ten query executions readings.

The Intermediate optimizer is implemented as a variant of the Advanced optimizer that checks for and creates one-to-one partition wise join pairs in place of the regular matching and clustering phases. The advanced optimizers can execute multiple joins and creates one-to-one partition wise join pairs. Given the capabilities of the query optimizer, the DBA has multiple choices regarding the partitioning strategy. In one extreme, the DBA can partition tables based on attributes appearing in filter conditions in order to take maximum advantage of partition pruning. At the other extreme, the DBA can partition tables based on joining attributes in order to take maximum advantage of one-to-one partition-wise joins; assuming the optimizer supports such joins (like the Intermediate optimizer). This enable the creation of multidimensional partitions to take advantage of both partition pruning and partition-wise joins. The experiments utilised the following partitioning scheme for TPC-H.

We will refer to the three schemes as:

- partitioning pruning strategies (PS-P),
- partitioning join strategies for joins (PS-J),
- and for both pruning and join strategies (PS-B).

Partition Scheme	Table	Partitioning Attributes	Number of Partitions
PS-P	orders	o_orderdate	28
	lineitem	l_shipdate	85
PS-J	orders	o_orderkey	48
	lineitem	l_orderkey	48
	partsupp	ps_partkey	12
	part	p_partkey	12
PS-B	orders	o_orderkey, o_orderdate	72
	lineitem	l_orderkey, l_shipdate	120
	partsupp	ps_partkey	12
	part	p_partkey	6

Figure 1 Partitioning scheme for TPC-H

Many companies have built data warehouses and have embraced business intelligence and analytics solutions. Even as companies have accumulated huge amounts of data, however, it remains difficult to provide trusted information at the correct time and in the correct place. The amount of data to mine, cleanse, and integrate throughout the enterprise continues to grow even as the complexity and urgency of receiving meaningful information continues to increase. Before information can become available in a dashboard or a report, many preceding steps must take place: the collection of raw data; integration of data from multiple data stores, business units, or geographies; transformation of data from one format to another; cubing data into data cubes; and finally, loading changes to data in the data warehouse[9].

III. QUERY EXECUTION TIME FOR VARIOUS STRATEGIES

Figure 2 shows the execution times for the plans selected by the three query optimizers for the ten TPC-H queries running on the database with the PS-J scheme. The Intermediate and Advanced optimizers are able to generate a better plan than the Basic optimizer for all queries, providing up to an order of magnitude benefit for some of them. Note that the Intermediate and Advanced optimizers produce the same plan in all cases, since one-to-one partition-wise joins are the only join optimization option for both optimizers for the PS-J scheme.

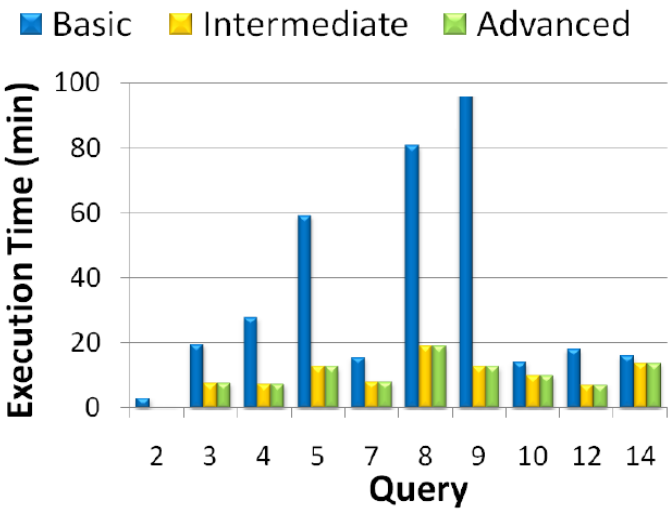

Figure 2 Query Execution time for PS-J Strategy

Figure 3 presents the corresponding execution times for the queries after optimization using partitioning strategies. The Intermediate optimizer introduces some overhead 12% and worst case of 21% due to the creation of child-join paths. The additional overhead introduced by the Advanced optimizer is due to the matching and clustering algorithms. Overall, the optimization overhead introduced by Advanced is low, and is most definitely gained back during execution.

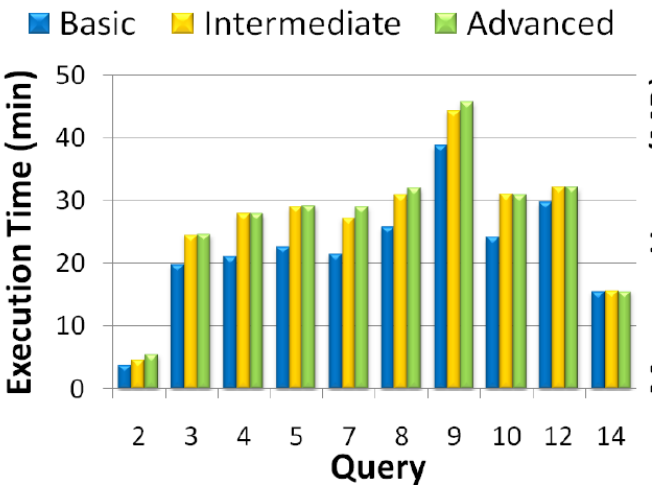

Figure 3 Query Execution time for after partitioning changes

Query performance is related directly to the optimizer capabilities and the partitioning scheme used in the database. Figure 4 shows the performance results for TPC-H queries 5 and 8 for the three optimizers over databases with different partitioning schemes. (Results for other queries are similar.)
Since a database using the PS-P scheme only allows for partition pruning, all three optimizers behave in an identical manner. A PS-J scheme on the other hand, does not allow for any partition pruning since join attributes do not appear in filter conditions in the queries. Hence, the Basic optimizer performs poorly in many cases, whereas the Intermediate and Advanced optimizers take advantage of partition-wise joins to produce better plans with very low overhead.

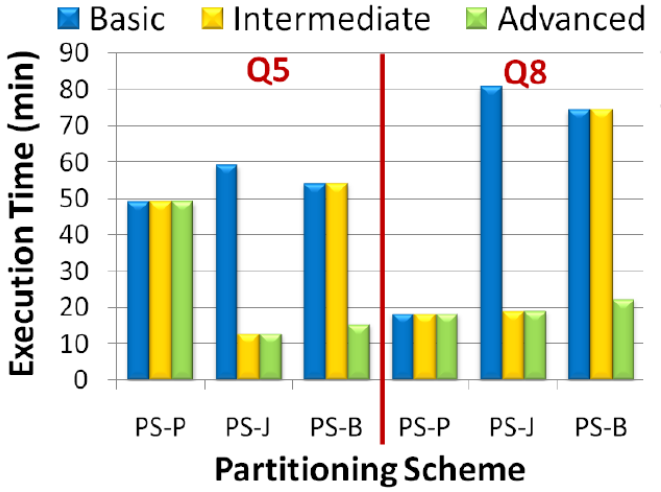

Figure 4 Query Execution time for three optimisers

The presence of multidimensional partitions in a PS-B scheme prevents the Intermediate optimizer from generating any one-to-one partition-wise joins, but it can still perform partition pruning like the Basic optimizer. The Advanced optimizer utilizes both partition pruning and partition-wise joins to find better-performing plans. Consider the problem of picking the best partitioning scheme for a given query workload. The best query performance can be obtained either from
(a) partition pruning (PS-P is best for query 8 in Figure 4), or
(b) from partition-aware join processing (PS-J is best for query 5 in Figure 4), or
(c) from a combination of both due to some workload or data properties. In all cases, the Advanced optimizer enables finding the plan with the best possible performance[1].

IV. SELECTION OF SUITABLE PARTITIONING STRATEGY

The study is aimed at discovering suitable partitioning strategies / new partitioning options for enhancing performance of multidimensional databases. Therefore, fundamental strategies— such as range, hash and list—, composite partitioning strategies including all possible combinations of Basic strategies, and Partition Extensions such as Reference and Interval partitioning strategies are covered.

4.1 Range partitioning

Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition. Partition by Range is used to establish ranges within the domain used as partitioning key. It is the most widespread type of partitioning and is often used with dates.

For a table with a date column as the partitioning key, the January-2010 partition would contain rows with partitioning key values from 01-Jan-2010 to 31-Jan-2010.

For Example the following sample code creates a table with four partitions and enables row movement on a University database[10].

```
CREATE TABLE credit evaluations
( evaluationid VARCHAR2(16) primary key
, graduation_id VARCHAR2(12)
, graduation_date DATE
, degree_granted VARCHAR2(12)
, degree_major VARCHAR2(64)
, college_id VARCHAR2(32)
, final_gpa NUMBER(4,2))
PARTITION BY RANGE (graduation_date)
( PARTITION graduation_date_70s
VALUES LESS THAN (TO_DATE('01-JAN-1980','DD-MON-YYYY')) TABLESPACE T1
, PARTITION graduation_date_80s
VALUES LESS THAN (TO_DATE('01-JAN-1990','DD-MON-YYYY')) TABLESPACE T2
, PARTITION graduation_date_90s
VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')) TABLESPACE T3
, PARTITION graduation_date_2000s
VALUES LESS THAN (TO_DATE('01-JAN-2010','DD-MON-YYYY')) TABLESPACE T4 )
ENABLE ROW MOVEMENT;
```

4.2 List partitioning

List Partitioning provides a list of values matching one partition in the partition key domain and a default partition for those not matched[11]. List partitioning enables you to unambiguously control how rows map to partitions by specifying a list of discrete values for the partitioning key in the description for each partition. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way. A PARTITION BY LIST clause is used in the CREATE TABLE statement to create a table partitioned by list, by specifying lists of literal values, (the discrete values of the partitioning columns qualifying rows matching the partition's single column partitioning key.)

4.3 Hash partitioning

Hash partitioning is used to transform the partitioning key value and maps it to given partition. Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size[11]. Hash partitioning is

the ideal method for distributing data evenly across devices. Hash partitioning is also an easy-to-use alternative to range partitioning, especially when the data to be partitioned is not historical or has no obvious partitioning key[10].

```
CREATE TABLE college_directory
(stid NUMBER PRIMARY KEY,
lname VARCHAR2 (50),
fname VARCHAR2 (50),
phone VARCHAR2(16),
email VARCHAR2(128),
class_year VARCHAR2(4))
PARTITION BY HASH (stid) PARTITIONS 4 STORE IN (t1, t2, t3, t4);
```

The PARTITION BY HASH clause of the CREATE TABLE statement identifies that the table is to be hash-partitioned. The PARTITIONS clause can then be used to specify the number of partitions to create, and optionally, the tablespaces to store them in. Otherwise, PARTITION clauses can be used to name the individual partitions and their tablespaces . The only attribute needed to specify for hash partitions is TABLESPACE. All of the hash partitions of a table must share the same segment attributes (except TABLESPACE), which are inherited from the table level[4].

Figure 5 offers a graphical view of the basic partitioning strategies for a single-level partitioned table[11].

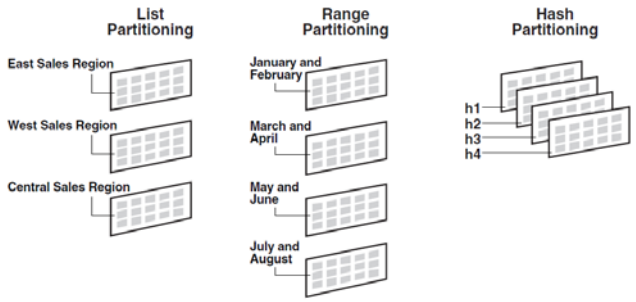

Figure 5 List Range and Hash Partitioning

4.4 Composite Range Partitioning

Composite partitioning is a combination of the basic data distribution methods; a table is partitioned by one data distribution method and then each partition is further subdivided into sub partitions using a second data distribution method. All sub partitions for a given partition represent a logical subset of the data.

The research covers both middle-to-large-size and big data styled databases with significant implications for consolidation, systems integration, high-availability, and virtualization support etc. All the experiments conducted emphasize subsequent performance tuning by partitioning and composite partitioning choices.

Composite partitioning supports historical operations, such as adding new range partitions, but also provides higher degrees of potential partition pruning and finer granularity of data placement through sub partitioning. Figure 6 offers a graphical view of range-hash and range-list composite partitioning.

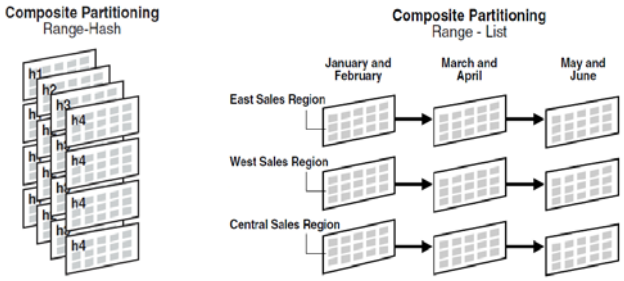

Figure 6 Composite Partitioning

Composite range-range partitioning enables logical range partitioning along two dimensions; for example, partition by order_date and range subpartition by shipping_date.

4.5 Composite Range-Hash Partitioning

Composite range-hash partitioning partitions data using the range method, and within each partition, subpartitions it using the hash method. Range-Hash partitioned tables are probably the most widespread type among the composite partitioning strategies. In general, to create a composite partitioned table, use the PARTITION BY RANGE LIST clause of a CREATE TABLE statement. Next, you specify a SUBPARTITION BY RANGE LIST HASH clause that follows similar syntax and rules as the PARTITION BY RANGE LIST HASH clause.

The partitions of a range-hash partitioned table are logical structures only, as their data is stored in the segments of their subpartitions. As with partitions, these subpartitions share the same logical attributes. Unlike range partitions in a range-partitioned table, the subpartitions cannot have different physical attributes from the owning partition, but they can reside another tablespace. Attributes specified for a range partition apply to all subpartitions of that partition.

4.6 Composite Range-List Partitioning

Composite range-list partitioning partitions data using the range method, and within each partition, subpartitions it using the list method. Composite range-list partitioning provides the manageability of range partitioning and the explicit control of list partitioning for the sub partitions[10].

Range-List partitioned tables are subject to range rules at the first partitioning level and list rules at second, list partitioning level, accordingly.

The Case study for oracle SH schema was also used to study the impact of partitioning on datatbase query performance[6].

```
CREATE TABLE q_territory_sales
( divno VARCHAR2(12), depno NUMBER,
itemno VARCHAR2(16), accrual_date DATE,
sales_amount NUMBER, state VARCHAR2(2),
constraint pk_q_dvdno primary key(divno,depno)
) TABLESPACE t8 PARTITION BY RANGE
(accrual_date) SUBPARTITION BY LIST (state)
(PARTITION q1_2000 VALUES LESS THAN
(TO_DATE('1-APR-2000','DD-MON-YYYY'))
(SUBPARTITION q1_2000_nw VALUES ('OR', 'WY'),
SUBPARTITION q1_2000_sw VALUES ('CA', 'NM'),
SUBPARTITION q1_2000_ne VALUES ('NY', 'CT'),
SUBPARTITION q1_2000_se VALUES ('FL', 'GA'),
SUBPARTITION q1_2000_nc VALUES ('SD', 'WI'),
SUBPARTITION q1_2000_sc VALUES ('TX', 'LA') ),
PARTITION q2_2000 VALUES LESS THAN
(TO_DATE('1-JUL-2000','DD-MON-YYYY'))
(SUBPARTITION q2_2000_nw VALUES ('OR', 'WY'),
SUBPARTITION q2_2000_sw VALUES ('CA', 'NM'),
```

```
SUBPARTITION q2_2000_ne VALUES ('NY', 'CT'),
SUBPARTITION q2_2000_se VALUES ('FL', 'GA'),
SUBPARTITION q2_2000_nc VALUES ('SD', 'WI'),
SUBPARTITION q2_2000_sc VALUES ('TX', 'LA‘)
), PARTITION q3_2000 VALUES LESS THAN
(TO_DATE('1-OCT-2000','DD-MON-YYYY'))
(SUBPARTITION q3_2000_nw VALUES ('OR', 'WY'),
SUBPARTITION q3_2000_sw VALUES ('CA', 'NM'),
SUBPARTITION q3_2000_ne VALUES ('NY', 'CT'),
SUBPARTITION q3_2000_se VALUES ('FL', 'GA'),
SUBPARTITION q3_2000_nc VALUES ('SD', 'WI'),
SUBPARTITION q3_2000_sc VALUES ('TX', 'LA')
), PARTITION q4_2000 VALUES LESS THAN (
TO_DATE('1-JAN-2001','DD-MON-YYYY'))
(SUBPARTITION q4_2000_nw VALUES ('OR', 'WY'),
SUBPARTITION q4_2000_sw VALUES ('CA', 'NM'),
SUBPARTITION q4_2000_ne VALUES ('NY', 'CT'),
SUBPARTITION q4_2000_se VALUES ('FL', 'GA'),
SUBPARTITION q4_2000_nc VALUES ('SD', 'WI'),
SUBPARTITION q4_2000_sc VALUES ('TX', 'LA')) );
```

4.7 Composite List-X Partitioning

Composite list-range partitioning enables logical range subpartitioning within a given list partitioning strategy; for example, list partition by country_id and range subpartition by order_date[10]. Composite list-hash partitioning enables hash subpartitioning of a list-partitioned object; for example, to enable partition-wise joins. Composite list-list partitioning enables logical list partitioning along two dimensions; for example, list partition by country_id and list subpartition by sales_channel.

V. ADVANCED PARTITIONING STRATEGIES

5.1 Reference Partitioning

Reference Partitioning strategy normally uses the referential integrity constraint between to table, and uses the key in the details table to attain partition on the referenced key, which points to a candidate primary key in another partitioned table, the master table. The referential integrity constraint must be enabled and enforced. Reference partitioning enables the partitioning of two tables that are related to one another by referential constraints. The partitioning key is resolved through an existing parent-child relationship, enforced by enabled and active primary key and foreign key constraints[3].

The benefit of this extension is that tables with a parent-child relationship can be logically equipartitioned by inheriting the partitioning key from the parent table without duplicating the key columns. The logical dependency also automatically cascades partition maintenance operations, thus making application development easier and less error-prone[11].

Reference partitioning enables the partitioning of two tables that are related to one another by referential constraints. The partitioning key is resolved through an existing parent-child relationship, enforced by enabled and active primary key and foreign key constraints[8].

The benefit of this extension is that tables with a parent-child relationship can be logically equipartitioned by inheriting the partitioning key from the parent table without duplicating the key columns. The logical dependency also automatically cascades partition maintenance operations, thus making application development easier and less error-prone.

REF Partitioning allows to partition a table by leveraging an existing parent-child relationship. The partitioning strategy of the parent table is inherited to its child table without the necessity to store the parent’s partitioning key column in the child table. Transparently inherits all partition maintenance operations that change the logical shape of a table from the parent table to the child table (for example when we drop/add/split partitions). This automatically enables partition-wise joins for the equal-partitions of the parent and child table. This is also perfect for star schemas in data warehouses as we can partition the fact table according to the dimension tables.

5.2 Interval partitioning

Interval partitioning is an extension of range partitioning which instructs the database to automatically create partitions of a specified interval when data inserted into the table exceeds all of the existing range partitions. We can create single-level interval partitioned tables and also composite partitioned tables: Interval-range, Interval-hash, Interval-list. The INTERVAL clause of the CREATE TABLE statement sets interval partitioning for the table. At least one range partition must be specified using the PARTITION clause[10]. The range partitioning key value determines the high value of the range partitions (transition point) and the database automatically creates interval partitions for data beyond that transition point. For each interval partition, the lower boundary is the non-inclusive upper boundary of the previous range or interval partition. The partitioning key can only be a single column name from the table and it must be of NUMBER or DATE type. The optional STORE IN clause lets you specify one or more tablespaces. At least one range partition must be specified using the PARTITION clause. The range partitioning key value determines the high value of the range partitions (transition point) and the database automatically creates interval partitions for data beyond that transition point. For each interval partition, the lower boundary is the noninclusive upper boundary of the previous range or interval partition. The partitioning key can only be a single column name from the table and it must be of NUMBER or DATE type.

5.3 Virtual Column Partitioning

Virtual Column Partitioning option permits to partition of a column on a virtual column, which is usually the outcome of a mathematical operation on two or more actual columns on the same table. This option extends every basic partitioning strategy. Virtual columns remove that restriction and enable the partitioning key to be defined by an expression, using one or more existing columns of a table. The expression is stored as metadata only. In the context of partitioning, a virtual column can be used as any regular column. All partition methods are supported when using virtual columns, including interval partitioning and all different combinations of composite partitioning. There is no support for calls to a PL/SQL function on the virtual column used as the partitioning column[6, 10].

Virtual Column based partitioning allows the partitioning key to be defined by an expression, using one or more existing columns of a table and storing the expression as metadata only. This enables a more comprehensive match of the business requirements.This is supported with all basic partitioning strategies and can also be used with interval partitioning as well as the partitioning key for REF partitioned tables. Virtual columns are treated as real columns except no DML operations are allowed[10].

```
CREATE TABLE SALES
( PROD_ID NUMBER NOT NULL,
CUST_ID NUMBER NOT NULL,
TIME_ID DATE NOT NULL,
CHANNEL_ID NUMBER NOT NULL,
PROMO_ID NUMBER NOT NULL,
QUANTITY_SOLD NUMBER(10,2) NOT NULL,
AMOUNT_SOLD NUMBER(10,2) NOT NULL,
PROD_TYPE NUMBER(1) AS
(TO_NUMBER(SUBSTR(TO_CHAR(PROD_ID),1,1))))
TABLESPACE USERS
```

```
PARTITION BY RANGE (PROD_TYPE) INTERVAL (1)
(PARTITION p1 VALUES LESS THAN (1));
```

VI. RECOMMENDATIONS FOR PARTITIONING STRATEGIES

Range partitioning is a convenient method for partitioning historical data. The boundaries of range partitions define the ordering of the partitions in the tables or indexes. Interval partitioning is an extension to range partitioning in which, beyond a point in time, partitions are defined by an interval. Interval partitions are automatically created by the database when data is inserted into the partition. Range or interval partitioning is often used to organize data by time intervals on a column of type DATE. Thus, most SQL statements accessing range partitions focus on timeframes.

Range partitioning is also ideal when we periodically load new data and purge old data, because it is easy to add or drop partitions. For example, it is widespread to keep a rolling window of data, keeping the past 36 months' worth of data online. Range partitioning simplifies this process. To add data from a new month, we load it into a separate table, clean it, index it, and then add it to the range-partitioned table using the EXCHANGE PARTITION statement, all while the original table remains online. After we add the new partition, we can drop the trailing month with the DROP PARTITION statement. The alternative to using the DROP PARTITION statement can be to archive the partition and make it read only, but this works only when partitions are in separate tablespaces. We can also implement a rolling window of data using inserts into the partitioned table[1].

With hash partitioning, a row is placed into a partition based on the result of passing the partitioning key into a hashing algorithm. Using this approach, data is randomly distributed across the partitions rather than grouped. As a general rule, hash partitioning is useful to enable partial or full parallel partition-wise joins with likely equisized partitions. It is also useful for distributing data evenly among the nodes of an MPP platform that uses Oracle Real Application Clusters. Consequently, we can minimize interconnect traffic when processing internode parallel statements. Hash partitioning is very useful in partition pruning and partition-wise joins according to a partitioning key that is mostly constrained by a distinct value or value list and to randomly distribute data to avoid I/O bottlenecks if we do not use a storage management technique that stripes and mirrors across all available devices.

We should use list partitioning when we want to specifically map rows to partitions based on discrete values. For instance all the customers for one states are stored in one partition and customers in other states are stored in other partitions. Account managers who analyze their accounts by region can take advantage of partition pruning.

Composite partitioning offers the benefits of partitioning on two dimensions. From a performance perspective we can take advantage of partition pruning on one or two dimensions depending on the SQL statement, and we can take advantage of the use of full or partial partition-wise joins on either dimension[9].

We can take advantage of parallel backup and recovery of a single table. Composite partitioning also increases the number of partitions significantly, which may be beneficial for efficient parallel execution. From a manageability perspective, we can implement a rolling window to support historical data and still partition on another dimension if many statements can benefit from partition pruning or partition-wise joins.

Composite range-hash partitioning is particularly widespread for tables that store history, are very large consequently, and are recurrently joined with other large tables. For these types of tables (typical of data warehouse systems), composite range-hash partitioning provides the benefit of partition pruning at the range level with the opportunity to perform parallel full or partial partition-wise joins at the hash level. Specific cases can benefit from partition pruning on both dimensions for specific SQL statements.

Interval partitioning can be used for every table that is range partitioned and uses fixed intervals for new partitions. The database automatically creates interval partitions as data for that partition is inserted. Until this happens, the interval partition exists but no segment is created for the partition. We should consider using interval partitioning unless we create range partitions with different intervals, or if we always set specific partition attributes when we create range partitions.

Reference partitioning is useful if we have denormalized, or would denormalize, a column from a master table into a child table to get partition pruning benefits on both tables. If two large tables are joined recurrently, then the tables are not partitioned on the join key, but we want to take advantage of partition-wise joins. Reference partitioning implicitly enables full partition-wise joins. If data in multiple tables has a related life cycle, then reference partitioning can provide significant manageability benefits.

Virtual column partitioning enables we to partition on an expression, which may use data from other columns, and perform calculations with these columns. PL/SQL function calls are not supported in virtual column definitions that are to be used as a partitioning key.

Virtual column partitioning supports all partitioning methods, plus performance and manageability features. To get partition pruning benefits, consider using virtual columns if tables are recurrently accessed using a predicate that is not directly captured in a column, but can be derived. Traditionally, to get partition pruning benefits, we would have to add a separate column to capture and calculate the correct value and ensure the column is always populated correctly to ensure correct query retrieval.

VII. CONCLUSION

Multi-column partitioning to reduce the scan time of large multidimensional database queries is a viable proposition. However, there are many constraints and practical considerations and large number of partitions does not aid the resolution of large queries, but does increase the cost. The semantics of the database must be considered when determining the number of partitions in each dimension. The work indicated how partitioning is beneficial to reduce wasted work.

VIII. REFERENCES

[1] A. Papadopoulos, P. Rigaux and M. Scholl, A performance evaluation of spatial join processing strategies, Proceedings of the 6th International Symposium on Advances in Spatial Databases, p.286-307 (1999).

[2] Bellatreche, L., Woameno, K.Y.: Dimension Table Driven Approach to Referential Partition Relational Data Warehouses. In: ACM 12th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 9–16 (2009)

[3] Eadon, G., Chong, E.I., Shankar, S., Raghavan, A., Srinivasan, J., Das, S.: Supporting Table Partitioning By Reference in Oracle. In: Proceedings of SIGMOD'08, pp. 1111–1122 (2008).

[4] Legler, T., Lehner, W., Ross, A.: Query Optimization For Data Warehouse System With Different Data Distribution Strategies, In BTW, pp. 502–513 (2007).

[5] Maria Halkidi, Michalis Vazirgiannis. Clustering Validity Assessment: Finding the Optimal Partitioning of a Data Set. In ICDM, 2001.

[6] Sanjay, A., Narasayya, V.R., Yang, B.: Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In: Proceedings of SIGMOD'04, pp. 359–370 (2004)

[7] Simon, E.: Reality check: a case study of an EII research prototype encountering customer needs. In Proceedings of EDBT'08, pp. 1 (2008)

[8] Swami, A.N., Schiefer, K.B.: On the Estimation of Join Result Sizes. In: Proceedings of EDBT'04, pp. 287–300 (1994).

[9] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy. Advances in Knowledge Discovery and Data Mining. AAAI Press, 1996.

[10] http://noriegaaoracleexpert.blogspot.in/2009/06/comprehensive-guide-to-oracle_16.html

[11] http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm