Database Partitioning for Oracle E-Business Suite

An Oracle White Paper
February 2008

ORACLE

Database Partitioning for Oracle E-Business Suite

An Introduction to Database Partitioning

OVERVIEW

Over the last few decades, the nature of enterprise computing has changed. Many Enterprise Resource Planning (ERP) systems have merged with Customer Relationship Management (CRM) systems and Business Intelligence (BI). The result is a single consolidated source of corporate information. Using Oracle E-Business Suite, customers can obtain a complete picture of their business from a single source. At the centre of the Oracle E-Business Suite is the Oracle database. Oracle's database architecture includes the ability to partition tables and indexes. Partitioning allows a single table and its associated indexes to be broken into smaller components depending on the table and choice of index partitioning methods.

INTRODUCTION

This white paper introduces the concept of partitioning using examples from the Oracle E-Business Suite and describes the best practices that can be employed to partition tables in the Oracle E-Business Suite and briefly describes the SQL commands that are available for maintaining partitioned tables and indexes

The Oracle E-Business Suite remains the information backbone for many enterprises and provides a platform that integrates data from a variety of sources. Inevitably, data correlation sometimes results in high data volumes concentrated in a few key tables. This can introduce performance problems when working with the data, and therefore you need a method of reducing the cost of retrieving and manipulating the data. A number of methods can be employed to enhance performance such as adding additional indexes, changing the access path of the optimizer, in order to avoid more expensive solutions such as adding faster storage, additional caching, or CPU. Both hardware solutions have constraints. Disks and CPU have a finite performance and the fastest technologies command a significant price premium. Inevitably, you will need to consider data management in order to control the system resources and ensure maintain system performance levels.

Archiving and purging can be used to control the growth of data. Instead, you may want to consider the partitioning database feature, where tables and indexes are broken down into smaller components. This feature has the advantage of ensuring good performance while ensuring immediate accessibility without needing to restore archives. Database partitioning is transparent and fully supported for the Oracle E-Business Suite. All decisions relating to how to access data is handled within the database, specifically by the cost based optimizer (CBO). The use of *custom partitioning* within the Oracle E-Business Suite is fully supported and it is a feature we encourage all customers to explore the usage of. This paper introduced the concept of database partitioning by describing several partitioning methods and

describes the differences between each approach using examples from the Oracle E-Business Suite. Additionally, this paper discusses how database partitioning can be used to provide high performance and improve scalability by reducing SQL transactions to only access data subsets. It also describes how partitioning can be used with table compression to further reduce storage requirements.

The second part of this paper describes the best practices for partitioning a table or index within the Oracle E-Business Suite. It includes guidelines on the selection of a suitable partition key and how to migrate data to a partitioned table. Finally a brief description of the commands available for maintaining partitioned tables and indexes is also included.

It should be noted that the examples being used in this paper are not absolute. In most cases these examples are demonstrating a concept. It is recommended that customers perform a detailed analysis before deciding on what and how to partition.

WHAT IS PARTITIONING AND HOW DOES IT WORK?

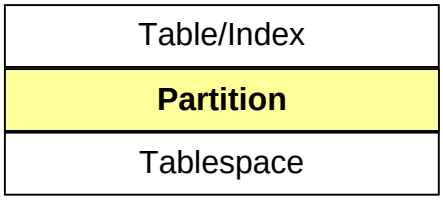
Partitioning splits tables and indexes into smaller, more manageable, components and is a key requirement for any large database with high performance requirements.

Partitioning divides a table, index, or index-organized table into smaller pieces. Each piece is called a partition (or subpartition for composite partitioned objects). Each partition has its own name, and may optionally have individual storage characteristics, such as compression or being stored in different tablespaces. Partitioning enables fast query access and updates by limiting the DML operation to only those partitions that need be accessed. Partitioning provides several performance-enhancing features:

Partitioning for Performance - Partition Pruning

This feature causes the optimizer (CBO) to skip unnecessary partitions that are not required by a particular SQL statement. Depending upon the SQL statement, the optimizer can explicitly recognize partitions and subpartitions that need to be accessed and the ones that can be eliminated. This can result in substantial improvements in query performance, because the optimizer is focusing on a specific subset of data which can be refined further if additional predicates exist. Internally the optimizer will eliminate the partitions at query execution time using the partition information in the data dictionary. The advantage of this is that we can determine the content of a given partition without having to query that partition; this eliminates the need to query the actual data in the partition.

The optimizer cannot prune partitions if the SQL statement applies a function to the partitioning column. Pruning is specified for a range of partitions, and the relevant partitions for the query are all the partitions between the first and the last partition of that range. Partitioning is an extra layer of the data dictionary between Tables/Indexes and Tablespaces.



Partition-wise Joins

Partitioning can improve the performance of multi-table joins, if the tables are partitioned on the join key. Partition-wise joins break larger joins into smaller joins that occur between each of the partitions, completing the overall join in less time and providing performance benefits for both serial and parallel execution.

WHY PARTITION?

Partitioning improves the manageability, performance, and availability of a partitioned object.

There are a number of benefits that partitioning can bring in the area of performance and in terms of manageability, availability, administration, and physical organization.

Table Availability/Improved Manageability

Partitioning can significantly reduce recovery times of key transaction tables by recovering the current partitions first. Table manageability, backup, restore, and rebuild can be performed at the partition level, as can index rebuilds. Partition aware operations such as `MOVE`, `EXCHANGE`, `REBUILD` can be used without affecting other active partitions.

Performance – The Need for Speed

When deciding its access path, the CBO automatically prunes unnecessary partitions and restricts access to only those necessary to fulfill the operation. Partitioning improves the access path of queries, as the majority of transactions will normally only reference current data. As the optimizer is partition aware, joins in SQL statements also benefit from pruning. Parallel operations also benefit from partitioning as they can occur at the partition level.

Within the Oracle E-Business Suite, partitioning can improve the performance of heavy batch jobs and upgrade performance, thus leading to reduced upgrade times and reduced system downtime. As the CBO is partition aware given certain conditions, the CBO can eliminate partitions that are not needed by the SQL statement and thus improves the performance for customers who typically run large batch jobs such as the Order Management (OM) Import Process and Payables (AP) Invoice Import.

Physical Organization & Reduction in Total Cost of Ownership (TCO)

Partitioning allows you to organize your data across different devices and therefore you can categorize your data in terms of need and hold data that is less frequently accessed on cheaper storage devices.

Understanding the growth and appropriate partitioning means that you have improved control over your data, resulting in a reduction of the total cost of ownership. Partitioning allows you to segregate your data and move your historic data to lower-cost storage. This is discussed in more detail in a separate paper entitled *Archiving, Purging and ILM for Oracle E-Business Suite*.

PARTITIONING CONCEPTS

Partition Key

At the heart of partitioning lies the concept of a partition key; this key determines in which partition a particular row will reside. The location of a row is maintained by the Oracle database.

The partition key is a set of one or more columns that determines in which partition a row will reside. It follows that each row in a partitioned table must reside in a single partition. The Oracle database automatically ensures, the correct location of a row based on its partition key. The location is automatically maintained as part of insert, update and delete operations. The following list specifies attributes for a partition:

- Consists of an ordered list of up to 16 columns
- Can contain columns that are NULLABLE
- Cannot contain a LEVEL, ROWID, or MLSLABEL pseudo column or a column of type ROWID

Table Partitions

In Oracle 10*g,* a table can have a maximum of 1 million partitions. With the exception of tables containing columns of `LONG` or `LONG RAW` datatypes, all tables can be partitioned (including columns of type `CLOB` or `BLOB`).

THE EVOLUTION OF PARTITIONING

Partitioning is now in it's 8th generation with Oracle 11g and in every major release of the database the partitioning functionality is being enhanced, by either adding new partitioning techniques, enhancing the scalability or extending manageability and maintenance capabilities.

Partitioning has been included with the Oracle database, since Oracle 8.0 and has been enhanced with each release of the Oracle database. New partitioning methods and features have been added. The following table shows a synopsis of the main changes.

Oracle 8.0	Partitioned Tables and Indexes Range partitioned tables Local partitioned indexes Global Range partitioned indexes Partition Pruning
Oracle 8*i*	Hash and Composite range-hash partitioned tables Range partitioned index-organized tables
Oracle 9*i*	List partitioned tables Hash partitioned index-organized tables
Oracle 9*i*R2	Composite Range-list partitioned tables List partitioned index-organized tables
Oracle 10g	Global hash partitioned indexes
Oracle 11g	Composite Partitioning Range-Range, Range-List, Range-Hash List-List, List-Range, List-Hash Interval-Range, Interval-List, Interval-Hash REF Partitioning, Virtual Column based partitioning

PARTITIONING & THE ORACLE E-BUSINESS SUITE

Several products in the Oracle E-Business Suite utilize partitioning in the base product. Partitioning doesn't require any change to your Oracle E-Business Suite application code.

Currently, several products in the Oracle E-Business Suite utilize partitioning in the base product, including; Advanced Planning and Scheduling, Oracle Payables (Trial Balance), Projects Resources, Workflow, Directory Services (Runtime tables), Daily Business Intelligence, HR (Employee Directory) and Engineering.

As stated earlier, partitioning is implemented at the database level and therefore no modification should be required to any product code to take advantage of its benefits. All that is normally required is to convert an existing non-partitioned table to a partitioned table; the same idea also applies to indexes.

Frequently Asked Questions: Using Database Partitioning with the Oracle E-Business Suite

A common question asked is, "Can I use the database partitioning feature in my Oracle E-Business Suite environment?" The answer is yes: the use of *custom partitioning* with the Oracle E-Business Suite is fully supported. However, if an incorrect partitioning strategy is chosen, partitioning can degrade performance, this is discussed in more detail in the second part of this paper. In addition, several Oracle E-Business Suite modules take advantage of partitioning straight out of the box.

What Does "Fully Supported" Mean?

If custom partitioning causes a particular Oracle E-Business Suite flow or transaction to fail with standard Oracle E-Business Suite product code, it is considered a product defect. Since the Oracle E-Business Suite is committed to being transparent to custom partitioning. Oracle Development will create patches or workarounds for all reported issues with standard Oracle E-Business Suite product code.

What Is Custom Partitioning?

Custom partitioning applies when an existing Oracle E-Business Suite product table is not partitioned and the table is redefined as a partitioned table by one of the following approaches:

1. Using a range, list, hash or composite partitioning method.
2. The partition scheme and/or partitioning method of an existing standard product table, which is already partitioned (as part of the standard product), is altered from that included in the base product.

Further information can be found at the following URL:
http://blogs.oracle.com/schan/2006/11/17

Examples of Custom Partitioning

An example of custom partitioning would be choosing to partition the table `OE_ORDER_LINES_ALL` which is currently not partitioned in the standard product.

Another example of custom partitioning would be changing the partition key or partition method used by an existing partitioned table as shipped in the base product. For example, changing the partition key of the table `WF_ITEM_ACTIVITY_STATUSES,` which is already partitioned by the column `ITEM_TYPE` and sub-partitioned by `ITEM_KEY`.

Example of Creating a Partitioned Table

All that is required is the `partition clause` needs to be included, as part of the `create table` command. This is shown in the following example:

```
CREATE TABLE GL_PERIODS
(PERIOD_SET_NAME NOT NULL,
PERIOD_NAME  NOT NULL,
LAST_UPDATE_DATE  NOT NULL,
LAST_UPDATED_BY NOT NULL,
START_DATE  NOT NULL,
END_DATE  NOT NULL,
PERIOD_TYPE NOT NULL,
PERIOD_YEAR NOT NULL,
PERIOD_NUM  NOT NULL,
QUARTER_NUM NOT NULL,
ENTERED_PERIOD_NAME  NOT NULL,
ADJUSTMENT_PERIOD_FLAG NOT NULL,
CREATION_DATE
CREATED_BY
LAST_UPDATE_LOGIN
DESCRIPTION
ATTRIBUTE1
ATTRIBUTE2
ATTRIBUTE3
ATTRIBUTE4
ATTRIBUTE5
ATTRIBUTE6
ATTRIBUTE7
ATTRIBUTE8
CONTEXT
YEAR_START_DATE
QUARTER_START_DATE
)
PARTITION BY RANGE (PERIOD_NAME)
(
PARTITION jan04_per VALUES LESS THAN ('JAN-2005'),
PARTITION feb04_per VALUES LESS THAN ('FEB-2005')
. . .
. . .);
```

TABLE PARTITIONING STRATEGY

There are several partitioning strategies available. The strategy chosen to partition a table depends on the data distribution of the table and access methods.

The four partitioning strategies available for partitioning tables, within the Oracle database, are as follows:

- Range
- List
- Hash

- Composite

General Syntax - A partitioned table declaration contains following elements:

- The physical structure of the table
- The partition structure, which defines the partition key

There are four different types of partitioning methods, these are declared in the `PARTITION BY` clause part of the `create table` command. Composite partitioning is limited to being a RANGE partition on the top level and HASH partitioning on a sublevel.

Multicolumn Partition Key

It is possible for a table to have a partition key that consists of several columns, which is analogous to composite column indexes. The exception is list partitions where a static list of literals is used; this will be covered later in the section titled: Multicolumn Partitioning.

Table Type

Partitioning can be applied to normal Heap Organized tables and to Index Organized Tables (IOT). An IOT cannot be list-partitioned. Clustered tables cannot be partitioned. Materialized Views (snapshots) can be partitioned.

Heap Organized Tables

Heap organized tables are the most common type of tables that are used, where data is stored as an unordered collection (heap). Typically, as data is added, it will be placed in the first free space found in the segment that can fit the data. As data is removed from the table, it allows space to become available and reused by `INSERT` and `UPDATE` statements.

Index Organized Tables

Internally within index organized tables, the table is stored as an index structure, where data is stored according to the primary key. There are several performance benefits to using index organized tables, such as reduced storage and faster access to table rows, by the primary key. As rows are stored in primary key order, range access by the primary key involves minimum block accesses. Performance can further be enhanced, by moving infrequently accessed non-key columns, from the B-Tree leaf block to an optional overflow storage area. This has the benefit of faster access to frequently accessed columns. Internally, this causes a reduction in size, which then results in a smaller B-Tree and therefore faster access.

As you would expect, there are a complete set of SQL commands for managing partitioning including adding, dropping, splitting, and merging partitions.

Range Partitioning

Range partitioning should be considered where data is based on consecutive ranges of values.

Range partitioning was introduced in Oracle 8. Rows are identified by a "partition key", which then determines the partition to which the row belongs. Each partition's end point is specified using: `VALUES LESS THAN` (value-list). The value-list must correspond in type and position to the partition key. The values in the value list are a static list of non-inclusive literals.

Range partitioning is useful for tables in the Oracle E-Business Suite, but the partition key needs to match the access predicate (access path key). This makes it possible to map rows to partitions, based upon ranges of column values. For example, we could range partition the table `AP_INVOICES_ALL` by using the `INVOICE_ID` column, as the partitioning key.

However, it is not always possible to predict how much data will map into a given range. In some cases, sizes of partitions may differ quite substantially, resulting in sub-optimal performance for certain operations, like parallel DML.

The partition key consists of any columns from the table, within the data type restrictions. When creating a table using range partitioning you will need to specify:

- Partitioning Method: Range
- Partition Columns
- Partitions: Identifying the partition boundaries

Another example of range partitioning would be to partition `GL_BALANCES` by the column PERIOD`_NAME`.

Example creation script

```
CREATE TABLE GL_BALANCES
(SET_OF_BOOKS_ID NUMBER(15) NOT NULL,
CODE_COMBINATION_ID NUMBER(15) NOT NULL,
CURRENCY_CODE   VARCHAR2(15) NOT NULL,
PERIOD_NAME     VARCHAR2(15) NOT NULL,
ACTUAL_FLAG     VARCHAR2(1) NOT NULL,
BUDGET_VERSION_ID NUMBER(15),
LAST_UPDATE_DATE  DATE NOT NULL,
. . . . . .
)
PARTITION BY RANGE (PERIOD_NAME)
(
PARTITION jan04_per VALUES LESS THAN ('JAN-2005'),
PARTITION feb04_per VALUES LESS THAN ('FEB-2005')
. . .
. . .
);
```

In this example, the majority of the SQL that accesses this table will use the condition `PERIOD_NAME` in the query, which will effectively map to one of the partitions.

Summary:

Range partitioning is suitable when one or more of the following conditions apply:

- There is a rolling window of data
- There are tables are frequently accessed using a range predicate on a suitable partitioning column. If using this approach the optimizer prunes unnecessary partitions.

List Partitioning

List partitioning gives you complete control as it allows you to map your rows to specific partitions. Typically this partitioning strategy is used when the partitioning criteria is an unordered list of values.

Unlike range partitioning, list partitioning, which was introduced in Oracle 9*i,* gives you complete control over how rows map to specific partitions by specifying a list of static values, as the description of the partitioning key. List partitioning does not support multi-column partition keys and therefore can only consist of a single column. The advantage of this kind of partitioning is that it allows you to group unorganized and unrelated sets of data together. The partition key values are specified with `VALUES` (value-list) clause.

The partition key can be any single column from the table, within the data type restrictions. All values of the partition key for a partition must be listed as literals. There is no "other" values clause.

NULL can be specified as a value. Any literal or NULL values must only appear once.

When creating list partitions you specify the following:

- Partitioning Method: List
- Partitioning column
- Partition description: A set of literal values, which determines if a row is in the partition or not

Index Only Tables (IOTs) cannot be list partitioned.

An example this type of approach is partitioning the `OE_ORDER_LINES_ALL` table by `OPEN_FLAG`, where open orders and closed orders are the partition names.

Example creation script

```
CREATE TABLE OE_ORDER_LINES_ALL
(LINE_ID       NUMBER NOT NULL,
ORG_ID         NUMBER,
HEADER_ID      NUMBER NOT NULL,
LINE_TYPE_ID   NUMBER NOT NULL,
LINE_NUMBER    NOT NULL NUMBER
ORDERED_ITEM   VARCHAR2(2000),
OPEN_FLAG      VARCHAR2(1) NOT NULL,
. . . . . .
)
PARTITION BY LIST (open_flag)
(
PARTITION open_orders VALUES ('Y'),
PARTITION closed_orders VALUES ('N')
);
```

Summary

List partitioning specifically maps rows to partitions based on a static list of literal values. The partition key for list partitioning can only be based on a single column.

Hash Partitioning

Hash partitioning used a hashing algorithm to decide the physical placement of data. Hash partitioning will distribute data evenly across a fixed number of partitions. Typically, this partitioning strategy will be used when there is no clear partitioning criteria.

Hash partitioning, which was introduced in Oracle 8*i,* uses a hashing algorithm that is applied to the partitioning key to stripe data into different partitions. Hash partitioning controls the physical placement of data across a fixed number of partitions. The hashing algorithm evenly distributes rows amongst partitions, making each approximately the same size. Hash partitioning is the ideal method for distributing data evenly across devices.

Hash partitioning is an easy-to-use alternative to range partitioning when data does not follow the classic historical pattern and there is no obvious partitioning key. There are two drawbacks to hash partitioning which are as follows:

- Hash partitioning only utilizes partition pruning on equality predicates, i.e. equality and IN lists.
- Range queries will not be suitable for use with hash-partitioned tables.

Generally, this method of partitioning is useful for Oracle E-Business Suite batch programs using parallel workers where block contention is significant a typical example would be interface import program.

When creating hash partitions you specify the following:

- Partitioning Method: Hash
- Partitioning column(s)
- Number of partitions or individual partition descriptions

An example this type of approach is partitioning the table `OE_ORDER_LINES_ALL` by the primary key, `LINE_ID`.

```
CREATE TABLE OE_ORDER_LINES_ALL
(LINE_ID       NUMBER NOT NULL,
ORG_ID         NUMBER,
HEADER_ID      NUMBER NOT NULL,
LINE_TYPE_ID   NUMBER NOT NULL,
LINE_NUMBER    NOT NULL NUMBER
ORDERED_ITEM   VARCHAR2(2000),
OPEN_FLAG      VARCHAR2(1) NOT NULL,
. . . . . .
)
PARTITION BY HASH (LINE_ID)
PARTITIONS 8
. . .
. . .;
);
```

Summary:

Hash partitioning is suitable when there is no natural partition key for your table, or the data doesn't follow any business view or logical view. Hash partitioning offers

some of the performance benefits of range partitioning. Additionally, it makes use of partition pruning and partition-wise joins based upon the partition key.

If you want to avoid data skew amongst partitions, hash partitioning allows data to be mapped evenly to a number of partitions and this will maximize I/O throughput. However, this method is not suitable for historic data.

Hash partitioning, minimizes block contention for batch processing there by increasing scalability.

Composite Partitioning

Composite partitioning is based upon a combination of the previously described partitioning strategies of Range, List, and Hash Partitioning.

Composite partitioning, as the name suggests, partitions tables using a combination of range and hash or list partitioning. However, there is a limit to the combination of partitioning methods that you can use together. Only range-partitioned tables can be sub-partitioned, therefore the only possible choices for composite partitioning methods are range-hash or range-list.

The composite partitioning method of range-hash combines the best of both worlds by allowing logical groupings at the partition level and handling data skew within the sub-partitions.

Using composite partitioning, the database would first distribute data into partitions according to the limits, established by the first method of partitioning, i.e. by range. Depending on the secondary method of partitioning used:

- Range-hash partitioning: Database uses a hashing algorithm to further divide the data into sub-partitions within each partition range.
- Range-list partitioning: Database divides the data into sub-partitions. Each range is partitioned, based on an explicit list specified. It follows that the secondary method of partitioning will be less specific and not be range orientated.

The subpartition partition key, can be the same as, or different to the range partition key.

When creating range-hash partitions you specify the following:

- Partitioning Method: Range
- Partitioning column(s)
- Boundary of partition
- Subpartitioning method: Hash
- Subpartitioning column(s)
- Number of subpartitions for each partition

The following example shows how to partition the `WF_ITEM_ACTIVITY_STATUSES` table using composite partitioning (range and hash).

```
create table wf_item_activity_statuses
…
partition by range (item_type)
subpartition by hash (item_key)
subpartitions 8
 (partition wf_item1 values less than ('A1'),
  partition wf_item2 values less than ('AM'),
  partition wf_item3 values less than ('AP'),
  partition wf_item4 values less than ('AR'),
  partition wf_item5 values less than ('AZ'),
  . . .
  partition wf_item48 values less than ('OE'),
  partition wf_item49 values less than ('OF'),
  partition wf_item50 values less than ('OK'),
  partition wf_item51 values less than ('OL'),
  . . .
  partition wf_item56 values less than ('PO'),
  partition wf_item57 values less than ('PQ'),
  partition wf_item58 values less than ('PR'),
  partition wf_item59 values less than ('QA'),
  . . .);
```

Summary:

Composite Range-Hash partitioning is suitable for the following::

- Historical data and cases using composite partitioning method, where the leading key has a small number of distinct values and the second key has a high number of distinct values.
- Consider an example where the table `WF_ITEM_ACTIVITY_STATUSES` has a composite key based upon `ITEM_TYPE` and `ITEM_KEY`. This table is generally accessed by `ITEM_TYPE`, which has a low number of distinct values (NDV). Typically customers will only have 5 or fewer `ITEM_TYPES`. However, when combined with the `ITEM_KEY` column, which has a high number of distinct values and selectivity, the partitioning key efficiency improves and results in the best of both partitioning methods.

Multicolumn Partitioning

Multicolumn partitioning should be used when the partitioning key is composed of several columns, where later columns give a higher degree of granularity then previous columns. An applicable scenario would be for example if you wanted to decompose a country by region or state.

In certain cases, you may need a higher degree of granularity in your partition key, which is typically provided by the trailing columns in the partition key. For range and hash partitioned tables you can specify up to 16 partition key columns. When deciding in which partition to place a particular row, the database will only use the next column that makes up the partition key if it can't uniquely identify a single destination partition from the first key column. The database doesn't evaluate all the columns in the partition key when deciding in which partition to place a row.

INDEX PARTITIONING METHODS

The rules for partitioning indexes are similar to those for tables and you can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables.

The partitioning of indexes offers the same benefits in terms of manageability and performance as that of partitioned tables. The rules for partitioning indexes are similar to those for tables. Several different methods are available for partitioning indexes, these are as follows:

- Local (inherits same partitioning method as table)
- Global (Hash, Range)

- Prefixed
- Nonprefixed

Indexes can be of different types: B*Tree, Bitmap, Bitmap Join, and Function-Based indexes. The index types are independent of the index partition method. All index types can be partitioned, but some restrictions apply. For example, a bitmap index cannot be global partitioned, it must be local to the partitioned table.

Indexes can be partitioned or non-partitioned and can be used with a partitioned or non-partitioned table. It should be noted that when we do partition an index, each partition is a completely separate index. The information is stored in that partition's index and as such, there is no master index or larger super index to refer to.

Local Indexes: A local index on a partitioned table is made where the index is partitioned in the exact same manner as the underlying partitioned table; i.e. the local index inherits the partitioning method of the table. This is known as *equipartitioning*. Each partition of a local index corresponds to one and only one partition of the underlying table. For example, if we partitioned the table `AP_INVOICES_ALL` using range partitioning on the `INVOICE_ID` column, it follows that any local indexes would also be partitioned automatically by the database using the same partition key.

Global Partitioned Indexes: A global partitioned index is an index on a partitioned or non-partitioned table which is partitioned *independently;* i.e., using a different partitioning-key from the table. Global-partitioned indexes can be range or hash partitioned. For example, we could range-partition the table `GL_BALANCES` using the `SET_OF_BOOKS_ID` column. We could then create a global index on this table, that could be range-partitioned using a different partitioning key, e.g. `CODE_COMBINATION_ID`.

Global Non-Partitioned Indexes: A global non-partitioned index is essentially identical to an index on a non-partitioned table. The index structure is not partitioned.

These types of indexes can be sub-divided into two further categories, *prefixed* and *non-prefixed*. Here also, there are restrictions on the combination of index type and partitioned type allowed; these are discussed later in this paper.

Automatically Maintaining Indexes

As a result of running a table maintenance operation on a partitioned table (for example, merging two adjacent partitions), the corresponding indexes/index partitions get marked as `UNUSABLE` (for global indexes).

In Oracle 10g, using the `ALTER TABLE` command and specifying the `UPDATE INDEXES` clause forces the database to override this behavior and updates the index at the same time as it executes the maintenance operation. There are several advantages of using this clause, including having the index remaining highly available. The index can be used to access partitions that are not impacted by the

table maintenance operation. From a maintenance point of view, each index does not have to be rebuilt individually and the index is updated at the same time as the base table.

Global Partitioned Indexes

A global partitioned index is an index on a partitioned or non-partitioned table, which is partitioned *independently* i.e. using a different partitioning-key from the table. Global-partitioned indexes can be partitioned using range or hash partitioning.

Global partitioned indexes are flexible, in that they allow you to pick a partitioning method that is different from that of the table. There is no required relation between the table and index partitioning method. Global partitioned indexes can also be used on nonpartitioned tables. Currently global partitioned indexes can be either range or hash partitioned. It therefore follows that the index key in a global partitioned index will refer to rows stored in more than one table partition.

Prefixed and Non-prefixed Indexes

Global partitioned indexes can be either prefixed or non-prefixed. This is not an attribute that you specify, but rather a consequence, of the index key column and partition key column matching.

A global partitioned index is prefixed, if the partition key is the leading index key. Global prefixed partitioned indexes can be unique or non-unique.

A global partitioned index is non-prefixed, if the leading index key is not the same as the table partition key.

Only B*Tree indexes can be created on global partitioned tables.

Maintaining Global Partitioned Indexes

Almost every attribute of a partitioned table or index is alterable after the table or index has been created and populated. This is different from altering a nonpartitioned table or index. Changes to a table's logical properties, such as the number and types of data columns, can be made to partitioned as well as nonpartitioned tables, with the same syntax. The partition's logical property, for example, the name, can also be altered.

Individual partitions can be exchanged, moved, truncated, or dropped, affecting the data within the partition.

Physical properties of a partition, such as the storage attribute can be altered. For some attributes the partition must be moved for the changes to take effect. The table or index's partition key definition can be altered by adding, dropping, merging, or splitting partitions (of the table) or index. However, this may affect the data within existing partitions.

Global partitioned indexes are harder to maintain than local indexes, however they do offer an efficient access method to any individual record. During table or index interaction during partition maintenance, all partitions in a global index will be affected. This is because, when the underlying table partition has any of the following maintenance operations applied to it: `SPLIT, MOVE`, `DROP,` or

TRUNCATE, both global indexes and global partitioned indexes will be marked as unusable. It therefore follows that it is not possible for partition independence to occur for global indexes

Depending on the type of operation performed on a table partition, the indexes on the table will be affected. When altering a table partition you can use the UPDATE INDEXES clause; this automatically maintains the affected global indexes and partitions. The advantage of using this command is that the index will remain available and online throughout the operation and therefore prevent any unnecessary interruption to the users. Furthermore, indexes do not have to be rebuilt once the operation has completed.

If a table partition is recovered to a point in time, the index must be recovered to the **same** point. You also need to recreate the entire global index otherwise the index entries will be scattered across partitions.

Maintaining Global Hash Partitioned Indexes

Global hash partitioned indexes are a new feature in the Oracle 10g database. They offer better performance by reducing contention when the index is right growing (most of the index insertions occur only on the right edge of an index.). This is due to the index entries having a hashing algorithm applied to them. This algorithm will evenly spread index entries over multiple partitions, which in turn spreads the contention over multiple partitions. An example of a right growing index is an index based on a sequence value. For example in the table OE_ORDER_LINES_ALL, the OE_ORDER_LINES_U1 index is based on the column LINE_ID. The value for LINE_ID comes from a sequence. Global hash partitioned indexes are useful in scenarios where there are is a low number of index leaf blocks accessed.

Local Partitioned Indexes

A local index on a partitioned table is partitioned using the same method as used by the underlying partitioned table; i.e., the local index inherits the partitioning method of the table, which is know as *equipartitioning*.

Global partitioned indexes can be defined on a non-partitioned table; however, local indexes must be defined on partitioned tables. For local indexes, the index keys within the index will refer only to the rows stored in the single underlying table partition. A local index is created by specifying the LOCAL attribute and can be created UNIQUE or NONUNIQUE. The table and the local index are partitioned in exactly the same manner or have the same partition key. Local indexes can only be unique if the partition key is part of the index key. By enforcing this restriction, the database ensures that the rows with the exact same index key will always map to the same partition.

The term *equipartitioning* refers to the concept of having each local partitioned index associated with exactly one partition of the table.

Local indexes are much easier to maintain than other types of partitioned indexes and offer greater availability as the local indexes are automatically maintained. The Oracle database ensures that the index partitions are synchronized with their

corresponding table partitions. It follows that the Oracle database automatically maintains the index partition whenever any kind of maintenance operation occurs on the underlying tables, such as when partitions are added, dropped, or merged . This enables each table-index pair to be independent; and therefore any actions that make one partition's data invalid or unavailable only affects a single partition.

Local Prefixed & Non-Prefix (Partitioned Indexes)

Local prefixed indexes can be unique or nonunique and can be specified against all four table partition types.

A local index is *prefixed* if the partition key of the table and the index key are the same; otherwise it is a local *non-prefixed* index. For example, if the `AP_INVOICES_ALL` table is partitioned on the column `INVOICE_ID`. If we create a local index `AP_INVOICES_L1` on this partitioned table, having it's index key as the (`INVOICE_ID, SUPPLIER_ID`) columns. The index `AP_INVOICES_L1` is *local prefixed* as the leading column and the partition key match. On the other hand, if the `AP_INVOICES_L1` index is defined on column `INVOICE_DATE,` then it is a *non-prefixed* local index. Generally, the need for non-prefixed local indexes will not arise in the Oracle E-Business Suite.

Global Nonpartitioned Indexes

Global nonpartitioned indexes, offer the same efficient access to any individual record in any partition and behave just like a non-partitioned index. Since the index structure is not partitioned, the index is available to all partitions. A scenario where this type of index would be useful is with a query that does not include the partition key of the table as a filter, but you still want the optimizer to use an index.

Consider the case where the referenced column was as part of the join predicate and had a global index created on it. For example, partition the `GL_BALANCES` table using the `PERIOD_NAME` column and create a global nonpartitioned index on the column `CODE_COMBINATION_ID`. You would want a SQL statement that uses this column as a join predicate to use the CBO to make use of this index.

Performance Considerations regarding Prefixed and Nonprefixed Indexes

Before deciding between a prefixed or nonprefixed index, it is worth considering its usage, by defining exactly how the table is accessed. You also need to consider how you expect the optimizer to behave if you use one these types of indexes. For example, is the index likely to be used by end-users transactions, or as part of a large batch program? In either case, choosing the wrong index prefix method could result in poor performance due to the additional overhead of using or maintaining an additional index.

The optimizer can make use of partition pruning, if we have a prefixed (local or global) index and we are using a predicate that is part of the index column. By doing this we are restricting the application of the predicate to a subset of index partitions. With a nonprefixed index, the database will have to apply a predicate,

involving the index column to all index partitions. This is required to look up a single key, or to do an index range scan. Note that if there is also a predicate on the partitioning column(s), multiple index probes may not be required due to the local index being *equipartitioned* with the underlying table. In this case, the database will simply prune the partitions based on the partition key.

Summary

The Oracle database has a rich set of techniques for partitioning indexes, so that they can be optimally applied in any business environment.

The following index partition methods are available:-

- Global Non-partitioned (prefixed)

  Use Global Non-partitioned indexes for all indexes, which are not prefixed by the table partition key.

- Global Range Partitioned (prefixed)
- Global Hash Partitioned (prefixed) – introduced in 10g

  Extremely useful for right-growing indexes experiencing contention due to high levels of concurrency.

  Allows the index to be partitioned without affecting the table.

- Local Partitioned Prefixed

  Use in place of indexes which already contain the partition key as a prefix.

- Local Partitioned Non-Prefix

  Should only be used when all the queries, using the local (non-prefixed) index, always include the partition key filter.

Maintenance of Partitioned Tables & Indexes

Oracle provides a comprehensive set of SQL commands for managing partitioned tables and indexes, which includes commands for adding new partitions, dropping, truncating, splitting, merging, moving and compressing partitions.

There are number of different kind of maintenance operations for partitions and subpartitions that can be performed on tables and indexes. For example:

- Merging two partitions together or adding new partitions to a table
- Dropping unused partitions or partitions containing historical data that is no longer required; or splitting or adding partitions if new partition key values are added

Within the `alter table` command, there are a number of clauses that let you achieve this:

```
ADD (HASH), COALESCE (HASH), DROP, EXCHANGE, MERGE,
MOVE, SPLIT, TRUNCATE
```

It is important to understand which partition maintenance operations are allowed with the different table and index partitioning methods. These are summarized in the next section of this paper.

Table: ALTER TABLE Clause for Maintenance Operations of Table Partitions

Operation	Description	Range	Hash	List	Composite: Range/Hash	Composite: Range/List
Adding Partitions	To add a new partition after the highest partition.	ADD PARTITION	ADD PARTITION	ADD PARTITION	ADD PARTITION MODIFY PARTITION ... ADD SUBPARTITION	ADD PARTITION MODIFY PARTITION ... ADD SUBPARTITION
Coalescing partitions	Reduces the number of partitions in a table or index by redistributing contents into one or more remaining partitions as determined by the hash function.	n/a	COALESCE PARTITION	n/a	MODIFY PARTITION ... COALESCE SUBPARTITION	n/a
Dropping partitions	Drops a partition for a given table.	DROP PARTITION	n/a	DROP PARTITION	DROP PARTITION	DROP PARTITION DROP SUBPARTITION
Exchanging Partitions	To change a partition into a non-partitioned table and vice versa. Useful when want to convert non-partitioned tables to partitioned table.	EXCHANGE PARTITION	EXCHANGE PARTITION	EXCHANGE PARTITION	EXCHANGE PARTITION EXCHANGE SUBPARTITION	EXCHANGE PARTITION EXCHANGE SUBPARTITION
Merging Partitions	Merges the contents of two partitions into one partition. The two original partitions are dropped, as are any corresponding local indexes. Not used for hash-partitioned table.	MERGE PARTITIONS	n/a	MERGE PARTITIONS	MERGE PARTITIONS	MERGE PARTITIONS MERGE SUBPARTITIONS
Modifying Default Attributes	This command allows you to modify the storage parameters of all partitions. Note: The new attributes affect only future partitions.	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES MODIFY DEFAULT ATTRIBUTES FOR PARTITION	MODIFY DEFAULT ATTRIBUTES MODIFY DEFAULT ATTRIBUTES FOR PARTITION
Modifying Real Attributes of Partitions	Allows the modification of existing attributes of a table or index; e.g., moving an existing partition to a new tablespace.	MODIFY PARTITION	MODIFY PARTITION	MODIFY PARTITION	MODIFY PARTITION MODIFY SUBPARTITION	MODIFY PARTITION MODIFY SUBPARTITION
Modifying List Partitions: Adding Values	More literal values can be added to the defining value list, that establishes what the partition key is.	n/a	n/a	MODIFY PARTITION... ADD VALUES	n/a	MODIFY SUBPARTITION ... ADD VALUES
Modifying List Partitions: Dropping Values	Removal of literal values can be added to the defining value list, that establishes what the partition key is.	n/a	n/a	MODIFY PARTITION... DROP VALUES	n/a	MODIFY SUBPARTITION ... DROP VALUES
Moving Partitions	Allows you to move a partition to another tablespace, store data in a compressed format using table compression.	MOVE PARTITION	MOVE PARTITION	MOVE PARTITION	MOVE SUBPARTITION	MOVE SUBPARTITION
Renaming Partitions	Renaming of partitions and sub-partitions of tables and indexes. Allows you to give a more meaningful name rather than using the system generated ones.	RENAME PARTITION	RENAME PARTITION	RENAME PARTITION	RENAME PARTITION RENAME SUBPARTITION	RENAME PARTITION RENAME SUBPARTITION
Splitting Partitions	Redistributes the contents of a partition into two new partitions for example when a partition gets too large.	SPLIT PARTITION	n/a	SPLIT PARTITION	SPLIT PARTITION	SPLIT PARTITION SPLIT SUBPARTITION
Truncating Partitions	Removes all rows from a table partition, but the physical partition is still kept. The same logic only applies to local indexes due to equipartitioning.	TRUNCATE PARTITION	TRUNCATE PARTITION	TRUNCATE PARTITION	TRUNCATE PARTITION TRUNCATE SUBPARTITION	TRUNCATE PARTITION TRUNCATE SUBPARTITION

Table: ALTER INDEX Clause for Maintenance Operations of Table Partitions

Operation	Index Type	Range	Composite: Hash and List	Composite: Range/List
Adding Index Partitions	Global	-	ADD PARTITION (hash only)	-
	Local	n/a	n/a	n/a
Dropping Index Partitions	Global	DROP PARTITION	-	-
	Local	n/a	n/a	n/a
Modifying Default Attributes of Index Partitions	Global	MODIFY DEFAULT ATTRIBUTES	-	-
	Local	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES	MODIFY DEFAULT ATTRIBUTES MODIFY DEFAULT ATTRIBUTES FOR PARTITION
Modifying Real Attributes of Index Partitions	Global	MODIFY PARTITION	-	-
	Local	MODIFY PARTITION	MODIFY PARTITION	MODIFY PARTITION MODIFY SUBPARTITION
Modifying Real Attributes of Index Partitions	Global	MODIFY PARTITION	-	-
	Local	MODIFY PARTITION	MODIFY PARTITION	MODIFY PARTITION MODIFY SUBPARTITION
Rebuilding Index Partitions	Global	REBUILD PARTITION	-	-
	Local	REBUILD PARTITION	REBUILD PARTITION	REBUILD SUBPARTITION
Renaming Index Partitions	Global	RENAME PARTITION	-	-
	Local	RENAME PARTITION	RENAME PARTITION	RENAME PARTITION RENAME SUBPARTITION
Splitting Index Partitions	Global	SPLIT PARTITION	-	-
	Local	n/a	n/a	n/a

Additional Information:

Please refer to **Oracle® Database Administrator's Guide 10g Release 2 (10.2)** *Chapter: 17– Managing Partitioned Tables & Indexes* for additional information.

Partitioning can only change the way data is organized and accessed, but it cannot reduce the physical size of tables in a database. It can only change the logical view of the tables so that only those database records are accessed that are applicable to a transaction.

TABLE COMPRESSION

Using the compression feature of the Oracle database, an entire table can be compressed or specific partitions belonging to a table can be compressed. B-tree indexes can also be compressed using a technique known as key compression.

Table compression is a database feature that was introduced in Oracle 9iR2. When using compression, it is possible to compress some or all of the partitions that belong to a particular table; i.e., table compression can also be specified at the partition level.

Some or all of the partitions of a B-Tree index can also be compressed using key compression. Key compression is applicable only to B-Tree indexes. Bitmap indexes are stored in a compressed manner by default. The benefit of using key

compression is that it eliminates the repeated occurrences of key column prefix values and this has the added benefit of saving storage space and reducing I/O.

The Oracle database compresses data by removing duplicate values in a data block. The compression algorithm maintains a symbol table of commonly used values per column at the beginning of the data block and all duplicate values are replaced with a short reference to the symbol table. The compression ratio increases proportionally to the number of duplicate column values in each block. Each data block can have different levels of compression and given the nature of the data, not all tables will have good compression ratios. Therefore, some degree of analysis needs to be done before deciding whether to compress a table or not in a production instance. The compression algorithm guarantees that compression will never increase the size of the existing table.

There are some restrictions when it comes to compressing tables, such as data is only compressed during bulk operations/direct path inserts. Compression is suitable more for read-only data or where rows are rarely updated. Internally, data has to be uncompressed to be updated and then recompressed, which results in a small CPU overhead. Table compression may improve query performance by reducing disk I/O and memory use in the buffer cache. For example, you can compress partitions containing data to keep it on-line for longer.

Not all DDL/DML commands are supported on compressed tables. In Oracle 9iR2 adding a column to a compressed table results in an error, but this has been allowed in Oracle 10g. The following session extract shows an example of the problem.

```
SQL*Plus: Release 9.2.0.6.0 - Production on Fri Apr 6 10:50:32 2007

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.

Connected to:
Oracle9i Enterprise Edition Release 9.2.0.6.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.6.0 - Production

SQL>  select table_name, compression
  2   from dba_tables
  3   where owner='AR'
  4   and table_name = 'AP_INVOICES_ALL';

TABLE_NAME                     COMPRESS
------------------------------ --------
AP_INVOICES_ALL                ENABLED

SQL> ALTER TABLE AP_INVOICES_ALL ADD TRANSMISSION_FLAG VARCHAR2(1);
ALTER TABLE AP_INVOICES_ALL ADD TRANSMISSION_FLAG VARCHAR2(1)
                                *
ERROR at line 1:
ORA-22856: cannot add columns to object tables
```

Given that range and composite partitioning can separate data into distinct partitions, consider using the table compression feature to compress those partitions that are read only. This allows more data to be kept online, as well as reducing the physical storage costs.

In summary, restrictions on using table compression are as follows:

- Can be used for RANGE or LIST partitions
- Can not be used with HASH partitions or HASH or LIST sub-partitions

Implementing Compression

Table compression can be implemented in a number of different ways such as:

- Converting an existing compressed table to a compressed one
- Creating a new compressed table
- Adding compressed partitions to an uncompressed table

Creating a Compressed Table

This can be achieved by adding the `compressed` clause to the `create table` statement.

Example: Creating a compressed table:

```
CREATE TABLE OE_ORDER_LINES_ALL (
 LINE_ID        NUMBER NOT NULL,
 ORG_ID         NUMBER,
 HEADER_ID      NUMBER NOT NULL,
 LINE_TYPE_ID   NUMBER NOT NULL,
 LINE_NUMBER    NUMBER NOT NULL,
 ORDERED_ITEM   VARCHAR2(2000),
 OPEN_FLAG      VARCHAR2(1) NOT NULL
 )
 COMPRESS;
```

Converting Tables to Compressed Tables:

Using the `ALTER TABLE` command with the `COMPRESS` clause you can compress a database table.

```
ALTER TABLE AP_INVOICES_ALL MOVE COMPRESS;
```

To speed up this process you can also use the parallel option and specify the number of parallel workers. Keep in mind that this may affect your overall system performance.

```
ALTER TABLE RA_CUST_TRX_LINE_GL_DIST_ALL MOVE PARALLEL 20 COMPRESS;
```

Example: Compressing an Existing Table

(Performed under Oracle E-Business Suite 11.5.10 CU2 (Vision Database)/RDBMS ver: 9.2.0.6.0)

Before compression:

```
TABLE_NAME                       NUM_ROWS      BYTES      GB
------------------------------ ---------- ---------- -------
AP_INVOICES_ALL                     19817    8388608    .008
```

After Compression:

```
TABLE_NAME                     COMPRESS
------------------------------ --------
AP_INVOICES_ALL                ENABLED

TABLE_NAME                       NUM_ROWS      BYTES      GB
------------------------------ ---------- ---------- -------
AP_INVOICES_ALL                     19817    2359296  .00225
```

Example: Compressing Large Tables (Large Volume Oracle E-Business Suite Customer 10.5TB Database).

(Performed under Oracle E-Business Suite 11.5.10 CU2 (10.5TB Production Database)/RDBMS ver: 10.2.0.2.0)

Before compression:

```
TABLE_NAME                      NUM_ROWS      BYTES         GB     BLOCKS EMPTY_BLOCKS
------------------------------ ---------- ---------- ---------- ---------- ------------
GL_JE_LINES                    540851270 1.4076E+11 134.243438   17058752            0
GL_IMPORT_REFERENCES           243433330 9.5104E+10 90.6981875   11602728            0
RA_CUST_TRX_LINE_GL_DIST_ALL   481210970 8.9921E+10   85.75575   10922809            0
```

After Compression:

```
TABLE_NAME                      NUM_ROWS      BYTES         GB     BLOCKS EMPTY_BLOCKS
------------------------------ ---------- ---------- ---------- ---------- ------------
GL_JE_LINES                    540851270 3.6342E+10   34.65825   17058752            0
GL_IMPORT_REFERENCES           243433330 2.1499E+10    20.5035   11602728            0
RA_CUST_TRX_LINE_GL_DIST_ALL   481210970 1.8341E+10  17.490875   10922809            0
```

Creating compressed tablespaces:

```
CREATE TABLESPACE USER_DATA
       DATAFILE 'diska:tabspace_file2.dat' SIZE 20M
       DEFAULT COMPRESS STORAGE ( … );
```

WHEN TO USE PARTITIONING?

Decision Process: Steps for Creating Partitioned Tables/Indexes

There are a number of reasons, for using custom partitioning tables and indexes it is therefore important to employ the optimal partitioning scheme and partition key, to help achieve your goal.

It is well known that the basic reasons for partitioning tables and indexes include performance and manageability. Other important reasons include archiving, purging, and data movement resulting from changes in the data lifecycle. It is important to employ an optimal partitioning scheme and partition key, to help ensure the best performance.

Step 1 – Is Partitioning Necessary?

Partitioning a table may not necessarily be the best solution to a performance problem, that you are encountering. In some cases your performance issue can be solved by means other than partitioning.

There are often cases where customers have a performance issue with a particular flow or process; e.g. a concurrent program where the evidence suggests that partitioning a particular table/set of tables will resolve the problem. As with all performance issues, start by analyzing the root cause of the problem. In the vast majority of cases, performance issues can be solved by means other than partitioning. Customers are encouraged to work with Oracle Support Services for all standard code performance issues.

Some of the reasons why partitioning should be considered are as follows:

1. Performance – Users complain that a particular page or concurrent request is running too slowly, in particular against tables with high volumes. If the data being accessed is within a specified range, partitioning may be an ideal solution. The goal should be to optimize access to specific data sets (usually current), while still maintaining historical data online. It is very simple to determine which are the largest tables within your Oracle E-Business Suite implementation.

   If archiving and purging are done before an upgrade is performed, they can help contribute to the reduction of time taken to upgrade.

2. Archiving/Purging - Oracle E-Business Suite includes several archive and purge routines that will remove data from your system that is no longer required. This may reduce the overall data volumes and therefore improve performance by reducing the amount of I/O and associated CPU usage.

3. Integration with Oracle Information Lifecycle Management
   Information Lifecycle Management (ILM) is concerned with everything that happens to data during its lifetime. ILM is an approach to move less frequently accessed data to cheaper storage and therefore reduces the total cost of ownership.

4. Simplified Administration – Reduced backup and recovery times are possible due to the ability to limit administrative activities to specific partitions. For example, only certain partitions will need to be backed up or recovered (not applicable to hash partitioning).

Once a table has been identified a table or an index that appears to be a suitable candidate for partitioning, detailed analysis is necessary to establish the best partitioning approach.

Step 2 - Should I Partition This Object?

Once you have identified a table or index as a candidate for partitioning, you need to answer the following questions:

- What is the functional use of the table, i.e., what exactly does the table store? This will help determine how the table is being used within the Oracle E-Business Suite, e.g. is this table representing a key entity, such as an invoice.
- How is the data accessed? There are several methods to determine access paths, such as Statspack, AWR. or Partition Advisor (an Enterprise Manager Plug-in). These tools should be run over at least a complete monthly business cycle to ensure that you understand how a particular table is accessed. Traces for a particular set of OAF/HTML screens, Oracle Forms, and/or concurrent requests can also help in the diagnosis.
- Which other modules or business flows use this table? Tables are generally used by multiple products e.g. `HZ_PARTIES`, `GL_PERIODS.`
- What is the growth rate and patterns of this table? By analyzing the statistics over a period of time, it is possible to understand the growth rate and patterns of the data. For example, a posting program may set the POSTED_FLAG to "N" for new records and then set then to null once they have been posted. This means that over time, 90% of the table will have rows with null values in this column.

Step 3 - Which Partitioning Method Should I Use?

After it has been determined that a table or index will benefit from partitioning, the next step is to identify an appropriate partitioning strategy.

In order to determine the optimal table partitioning method for a particular implementation, refer to the summary table below. For indexes, also evaluate different indexing methods: global and local partitioned, and prefixed and nonprefixed indexes.

Partitioning Strategy	When to Use?
Range	Convenient for partitioning historical and transaction data, as the boundaries of range partition define the order of partition in tables and indexes. Useful for tables which have a natural partition key such as OPEN_FLAG, PERIOD_NAME, PLAN_ID and the majority of the access paths are based on this key. Good for SQL that primarily scans data based on time periods or key values.
List	Used when rows are to be mapped to a particular partition based upon a specific value. Note: Partition key is based on a single column.
Hash	This distributes data in a manner that does not correspond to either a logical or business view of data and is not an effective way to manage historical data. It shares some characteristics with range partitioning e.g. partition pruning. Use it to avoid data skew across partitions. The Oracle database hashes the data resulting in approximately equal stripes across devices, thus enabling I/O to be maximized while avoiding hot spindles. It uses partition pruning & partition joins according to the partition key. It is useful for scalability (minimizes block contention) and for tables that do not have natural partition keys.
Composite Partitioning Range/Hash	This offers the benefits of range and hash partitioning. Typically, the Oracle database partitions by range, then creates sub-partitions within each range to distribute the data. Data placed within these partitions is logically ordered by the boundaries that define the range. Useful for tables, which have a natural partition key, however, range alone would lead to wide skews. Note: Partitioning of data within the sub-partition has no logical order.
Composite Partitioning Range/List	Offers the benefits of range and list partitioning, Oracle database partitions by range and then creates sub-partitions for each specific value. Generally not useful for the Oracle E-Business Suite.

Step 4 - Identifying the Partition Key

At the heart of any good partitioning strategy lies the choice of a good partitioning key. Typically, this is the column that is most frequently used to access the data.

There will be a natural and logical partition key for many large transaction tables/entities. This will typically be the column, which is most frequently used as the predicate to access the data. As an additional validation step, re-examine which SQL statements are using the table by reviewing the AWR or Statspack SQL repository.

When selecting a partition key, consider what happens when an update takes place on the column that the partition key is based on. An update may result in row migration, which will cause the loss of partition independence. This actually causes partition dependence, as an `update` statement for example will translate into a `delete` statement, followed by an `insert` statement.

Consider partitioning the `GL_BALANCES table.` Queries often include the `PERIOD_NAME` column and therefore this would be a good candidate partition key. Whenever any particular query includes the `PERIOD_NAME`, the optimizer uses partition pruning and only accesses those partitions that match the required `PERIOD_NAME.`

Step 5 – Performance Check & Access Path Analysis

It is important to ensure that your partitioning strategy does not cause performance regressions and therefore it is necessary to test your chosen partitioning strategy. For example, ensure that the transactions access the correct number of partitions. Always thoroughly test your newly partitioned tables and indexes before introducing them into a production instance. Remember that an inefficient partitioning scheme for a given table will result in poor performance.

You can confirm your partitioning strategy by performing some basic explain plan analysis. For example, test to check the optimizer is accessing the correct number of partitions using the DBMS_XPLAN tool (which has been available since Oracle 8i) as this will verify that partition pruning is occurring. The output produced by this tool will include the columns PSTART and PSTOP. In the example below, the optimizer had to scan all 32 partitions.

```
UPDATE /*+ ROWID (ALB) */ ap_liability_balance ALB
       SET    (ALB.vendor_id,
               ALB.vendor_site_id) = (SELECT AI.vendor_id,
                                             AI.vendor_site_id
                                      FROM   ap_invoices_all AI
                                      WHERE  AI.invoice_id =
                                             ALB.invoice_id),
              ALB.creation_date = SYSDATE,
              ALB.created_by = 1,
              ALB.last_update_date = SYSDATE,
              ALB.last_updated_by = 1,
              ALB.last_update_login = -1
       WHERE  ALB.rowid
       BETWEEN chartorowid(:l_start_rowid) AND
               chartorowid(:l_end_rowid)
```

Execution Plan:

```
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT              |                     |   176K|     9M|   199K  (3)| 00:39:57 |       |       |
|   1 |  UPDATE                       | AP_LIABILITY_BALANCE |       |       |            |          |       |       |
|*  2 |   FILTER                      |                     |       |       |            |          |       |       |
|   3 |    PARTITION HASH ALL         |                     |   176K|     9M|   199K  (3)| 00:39:57 |     1 |    32 |
|*  4 |     TABLE ACCESS BY ROWID RANGE| AP_LIABILITY_BALANCE |   176K|     9M|   199K  (3)| 00:39:57 |     1 |    32 |
|   5 |   TABLE ACCESS BY INDEX ROWID | AP_INVOICES_ALL     |     1 |    19 |     3   (0)| 00:00:01 |       |       |
|*  6 |    INDEX UNIQUE SCAN          | AP_INVOICES_U1      |     1 |       |     2   (0)| 00:00:01 |       |       |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(CHARTOROWID(:L_START_ROWID)<=CHARTOROWID(:L_END_ROWID))
   4 - access("ALB".ROWID>=CHARTOROWID(:L_START_ROWID) AND "ALB".ROWID<=CHARTOROWID(:L_END_ROWID))
   6 - access("AI"."INVOICE_ID"=:B1)
```

Step 6 – Partitioned Table Creation and Data Migration

Two methods can be employed to migrate data to a partitioned table. Each method has its own advantages and disadvantages in terms of rollback segment, redo log, and buffer cache usage. Always ensure that there is a method in place to revert to a non-partitioned table if problems ensue. Some methods for migration are more simple and straightforward than others and therefore you will need to analyze the data and estimate how many partitions you will have to create.

It is always the best practice to create the indexes after the data has been successfully migrated to the newly created partitioned table.

You can use SQL*LOADER to import/export and datapump utilities to load or unload data stored in partitioned tables. These utilities are all partition and subpartition aware.

Method: 1 – Straight Insert

```
INSERT /*+ APPEND PARALLEL*/ INTO RA_CUSTOMER_TRX_ALL_PART
SELECT * FROM RA_CUSTOMER_TRX_ALL
```

This method is the simplest. If you choose to use this method, it is recommended that you use the direct-path-insert method, using the `APPEND` hint. In addition, since this statement will generate a large amount of undo and redo logs, it is recommended to perform the insert with the `NOLOGGING` option and change it back to `LOGGING` after you are done.

As part of the insert, records will go to the correct partitions after creating the partitioned table.

This procedure could further be enhanced by executing it in parallel.

The steps for this method can be summarized as follows:

1. Create an empty partitioned table using the partitioned clause and with the parallel option. The table name must be of a **different name** to the non-partitioned table. The assumption is that having performed the analysis, the partition key would be known as well as the exact number of partitions that need to be created.
2. Populate data for required partition from the nonpartitioned table.
3. There are a number of performance enhancing features you can apply to your `INSERT` or `SELECT` SQL statement such as:
    - Consider using the `APPEND` hint with the `INSERT`, this is an easy code change and provides good performance. If logging is enabled and indexes are present then the `INSERT /*+ APPEND */` hint may not be effective.
    - To minimize the overhead of index maintenance, drop indexes prior to migration and recreate them once the partitioned table has been populated.
4. Rename the partitioned table to the original table or change the synonym.
5. Build table indexes for partitioned table.
6. Gather statistics for the new objects.

Oracle Data Pump was introduced in Oracle 10gR1. It allows very high-speed movement of database and forms the basis of Oracle's new data movement utilities, Data Pump Export and Data Pump Import. Data Pump also has the added benefit of allowing you to specify whether to move a subset of data using data filters which are specified as part of Export and Import parameters.

Oracle Data Pump

Data Pump is a new utility that is available in Oracle 10gR1 and replaces the existing *import* (`imp`) and *export* (`exp`) utilities. Note that both of the existing utilities are still shipped with the latest versions of the database and are still fully supported. Data Pump does not work with utilities older than the 10g R1 and therefore database Dump files generated by the new Data Pump Export utility are not compatible with dump files generated by the original Export utility. Therefore, files generated by the original Export `(exp)` utility cannot be imported with the Data Pump Import `(impdp)` utility.

Any job submitted by Data Pump is run within the database and this feature makes any job independent of the standalone client that was used to submit the import or export process. This also allows Database Administrators to submit and monitor jobs independently.

The Data Pump command line clients, `expdp` and `impdp`, invoke the Data Pump Export utility and Data Pump Import utility and have a similar interface to the original export (`exp`) and import (`imp`) utilities. Oracle Data Pump offers many features over and above the existing import and export utility such being able to specify multiple threads to execute the Data Pump job.

Data Pump Access Methods

Data Pump uses direct path and external tables access methods to load and unload table row data. Both of these methods use the same external data representation, so they can be used interchangeably. Therefore, data unloaded with one method can be loaded using the other. Data Pump automatically chooses the fastest method appropriate for each table. By default, Data Pump uses direct path for loading and unloading data, when the table structure allows it. However, there are certain cases where Data Pump uses *external tables* instead of *direct path inserts*:

- A global index on a multi-partitioned tables exists during a single-partition load. This includes object tables that are partitioned.
- The table into which data is being imported is a pre-existing table which is partitioned.

For Additional Information:
Refer to Oracle® Database Utilities10g Release 2 (10.2): *Part 1 Oracle Data Pump* for further information.

Method 2 – Import/Export using Data Pump

This method makes use of the Data Pump Import and Data Pump Export as provided with the Oracle Database. It requires additional space to hold the Data Pump export file. It follows a similar procedure as the first method. Note that this method has the added overhead of requiring you to perform an Data Pump Export

and then an Data Pump Import and will therefore take up twice the amount of space. The Data Pump Export/Import can be done in parallel.

The steps for this method can be summarized as follows:

1. Since the import/export job is inside the database, if you want to export to a file, the first thing that you must do is create a database `DIRECTORY` object for the output directory and grant access to users who will be doing exports and imports:
2. Populate the data for the required partition from the non-partitioned table.
    - Create an empty partitioned table, .This must have a different name to the non-partitioned table with the parallel option.
    - Create a directory and grant READ and WRITE permissions on the directory to other users.
    - Grant READ or WRITE permissions to a directory object to a particular user, this will allow the Oracle database to read this file.
    - Once the directory access is granted to a user, this user can export their database objects.
3. Export the non-partitioned table using Data Pump Export.
4. Rename existing non-partitioned table to <table-name_OLD>.
5. Rename the partitioned table to the original table or change the synonym.
6. Using Data Pump Import the non-partitioned table data into the partitioned table.
7. Build table indexes.
8. Gather table statistics.
9. Remove the parallel option from newly partitioned table.

Step 7 – Maintenance Step

Once you have successfully created your partitioned table, there will be a number of maintenance operations you will need to perform as and when the need arises. For example, you may need to split or add partitions if new partition key values are added or partitions are merged. You can also drop unused partitions or partitions containing, historical data, which are no longer required.

PARTITION MAINTENANCE OPERATIONS

There are several SQL commands dedicated to partition maintenance.

In general, some of the most common maintenance operations involve the following:

- Adding Partitions/Subpartitions
- Dropping a Partition
- Moving a Partition
- Splitting and Merging Partitions
- Exchanging a Partition with a table
- Renaming a Partition

Changes made to most of the attributes of a partitioned table or index can be done even after it has been created and populated. This is not the case with nonpartitioned tables and indexes where most of the attribute changes are applied to the entire object at object creation time.

Modifying the logical structure of a partitioned table can be done in the same manner as for a non-partitioned table. For example, you can change the name of a partition or modify one the physical properties of the partition such as the storage attribute.

Similarly, the partition key definition of a partitioned table or index can be altered by adding, dropping, merging, or splitting partitions of the table or index.

Maintenance operations can also be applied to individual partitions, such as exchanging, moving, truncating, or dropping.

Adding Partition/Subpartition

Adding a partition will have different effects, depending on the type of partitioning.

Range and List

For Range and List partitions, you can use the `ALTER TABLE ... ADD PARTITION` statement. By default, this creates an empty partition segment after the last existing partition (as the partition key range is not known). To add a partition at the beginning or in the middle of a table, use the `SPLIT`

PARTITION statement. Adding a new partition does affect global indexes and this means that **existing** global and local indexes remain usable.

For list partitions, the literal value has to be specified and this describes the contents of the partition; it must not exist in any other partition of the table.

Hash and Hash Subpartition

When adding a new partition to an existing hash-partitioned table, the addition of another hash partition causes the rehashing of rows. Rows from the existing partition are then distributed into the new partition. These rows are determined by the database based upon a hashing function. Depending on the type of table (regular Heap Organized or Index-Organized), global and local index partitions may become UNUSABLE or invalidated. These are described below:

- **Regular (Heap) Organized Table** - Indexes will be marked UNUSABLE unless you specify the UPDATE INDEXES clause as part of the ALTER TABLE statement. Local indexes for all the partitions are marked UNUSABLE and must be rebuilt. Global indexes or all partitions of partitioned global indexes are marked UNUSABLE and must be rebuilt.
- **Index-Organized Table** - Local indexes behave in the same way as with heap tables and will be marked UNUSABLE. Global indexes remain USABLE. Any new local index partitions are stored in the same tablespace as the table partition, unless the index has a default storage tablespace defined at index level.

Dropping Partitions

A partition can be dropped by using the ALTER TABLE ... DROP PARTITION/SUBPARTITION command.

Table Partitions

Partitions can be dropped from range, list, or composite range-list partitioned tables. For hash-partitioned tables or hash subpartitions of range-hash partitioned tables, a coalesce operation has to be performed instead.

When a partition is dropped, the equivalent of a DDL statement is run to discard all the rows stored in that partition. These rows cannot be rolled back. If a table only contains a single partition, then the partition cannot be dropped; instead, the table must be dropped.

To preserve the data in the partition, use the MERGE PARTITION statement instead of the DROP PARTITION statement. Only the owner of the table or a user with the DROP ANY TABLE privilege can use the DROP_TABLE_PARTITION or TRUNCATE_TABLE_PARTITION clause.

Index Partitions

Local indexes cannot be dropped directly, instead when the table partition is dropped, the corresponding local partition will also be dropped regardless of its status.

When the partition of a global index that contains data is dropped, the next highest partition is marked as unusable. On rebuild, the index entries are recreated in the next highest partition, which is then marked as valid. The highest partition of a global index cannot be dropped.

Moving Partitions

Given the tablespace model for Oracle Applications (OATM) – why would you want to move a partition? OATM, although highly recommended, is still optional. Secondly, if Information Lifecycle Management (ILM) is adopted using the move partition command then it is possible to move the partitions that need to be archived onto cheaper disks.

This can be done using the `ALTER TABLE ... MOVE/MODIFY PARTITION` command.

Table Partitions

Unlike a nonpartitioned table that can be moved in a single step, table partitions can only be moved one partition at a time. You can modify the physical storage attribute of a partition by using the `MOVE PARTITION` clause of the `ALTER TABLE` statement.

Some physical attributes such as `TABLESPACE` cannot be modified using the `MOVE PARTITION` clause, but can be changed using the `MODIFY PARTITION` clause. Modifying some other attributes, such as table compression, affects only future storage and **not existing** data. Therefore, always make a point to check.

Index Partitions

Depending on the type of database table, when a partition containing data is moved, the indexes may be invalidated and marked `UNUSABLE` as follows:

Regular (Heap) Organized Table

All partitions will be invalidated/marked as `UNUSABLE` unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement as follows:

- **Global Indexes** – All global indexes or all partitions of partitioned global indexes are marked `UNUSABLE` and must be rebuilt.
- **Local indexes** – Each index partition is marked `UNUSABLE` and must be rebuilt.

Index-Organized Table – Any local or global index defined for the partition being moved remains `USABLE`.

To move a subpartition of a composite partitioned table or index, the keyword `SUBPARTITION` is used instead of `PARTITION`. Typically, tables and indexes are not moved but they are rebuilt. When moving a table partition or rebuilding an

index partition, all storage attributes can be specified, thus causing a change during the move or rebuild.

Splitting Partitions

When a partition becomes too large it may take longer to backup, recover or maintain. It is possible to split the partition, using the `SPLIT PARTITION` clause of the `ALTER TABLE` or `ALTER INDEX` statement. This will cause the redistribution of the contents of the partition into two new partitions. Hash partitions or sub-partitions cannot be split.

Table Partitions

Table partitions are done if a partition that is range partitioned is split. This will result in the creation of two new partitions, containing all the rows of the original partitions. The first partition will contain all the values *less* than the partition key value. The second partition will contain rows that are *greater than* or equal to the partition key value.

When a list partition is split, the list of literals that are specified in the `SPLIT PARTITION` clause will determine into which partitions the existing rows will be inserted. Rows matching the partition key value will be placed into the first partition, the remaining rows from the original partition will be placed into a second partition.

Index Partitions

Depending on the type of database table, when a partition containing data is split, two index partitions are created. These indexes may be invalidated/marked `UNUSABLE`. This is summarized below:

Regular (Heap) Organized Table

All new partitions will be invalidated/marked as `UNUSABLE` unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement. This is summarized as follows:

- **Global Indexes -** All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
- **Local indexes** – Each new index partition is marked `UNUSABLE` and must be rebuilt.

Index-Organized Table – The database marks local indexes as `UNUSABLE`. Global indexes remain `USABLE`.

Merging Partitions

The `ALTER TABLE ... MERGE PARTITION` statement can be used to merge the contents of two partitions into a single partition. In addition, the two original partitions are dropped, along with any corresponding local indexes. Due to the

nature of hash partitioning, it is not possible to merge hash-partitioned tables or hash subpartitions of a range-hash partitioned table.

Table Partitions

When merging two adjacent range partitions into one partition, the resulting partition inherits the higher or upper boundary of the two merged partitions. It is not possible to merge nonadjacent range partitions.

Any two list partitions can be merged, the two partitions do not need to be adjacent, as there is no ordering of list partitions. The resulting single merged partition consists of all of the data, from the original two partitions. If a default list partition is merged with any other partition, the resulting partition will still be the default partition.

Index Partitions

Depending on the type of database table, when partitions containing data are merged into a single partition, the indexes may be invalidated marked UNUSABLE. This is summarized below:

Regular (Heap) Organized Table

All new partitions will be invalidated/marked as UNUSABLE unless you specify UPDATE INDEXES as part of the ALTER TABLE statement otherwise:

- **Global Indexes -** All global indexes or all partitions of partitioned global indexes, are marked UNUSABLE and must be rebuilt.
- **Local Indexes** – All local index partitions and subpartitions are marked UNUSABLE and must be rebuilt.

Index-Organized Table – The database marks local indexes as UNUSABLE. Global indexes remain USABLE.

Exchanging a Partition with a Table

Using the ALTER TABLE ... EXCHANGE PARTITION statement, range list and hash partitioned tables can be exchanged with non-partitioned tables.

Exchanging a partition does not move any rows and the tables can be populated or empty. Local indexes partitions are exchanged with matching nonpartitioned indexes defined on the non-partitioned table. Global indexes on the partitioned table will be marked UNUSABLE.

Indexes on the non-partitioned table, which are not exchanged with a local index, are marked UNUSABLE and will only be maintained/valid if the UPDATE GLOBAL INDEXES clause has been specified.

When exchanging a partition with a table, it is important to ensure that the partition key values must be valid values of the partition. This will be verified before the

actual exchange takes place by scanning the nonpartitioned table rows. Using the NOVALIDATE clause will skip this validation.

Renaming Partitions

When renaming partitions or sub-partitions, the partition name must be unique within the affected table or index.

To rename a partitioned/nonpartitioned table or index, use one of these two statements:

```
RENAME OE_ORDER_LINES_ALL TO OE_ORDER_LINES_ALL_BKP;

ALTER TABLE OE_ORDER_LINES_ALL RENAME TO OE_ORDER_LINES_ALL_BKP;
```

New names can also be assigned to subpartitions of a table by using the ALTER TABLE ... RENAME PARTITION command.

Index partitions and subpartitions can be renamed in a similar fashion but the ALTER INDEX syntax is used.

To rename an index partition, use the ALTER INDEX ... RENAME PARTITION statement.

EXAMPLE OF HOW TO PARTITION A TABLE

The objective of partitioning a table is to ensure that the optimizer returns the rows that satisfy the query criteria in the most efficient manner. The optimizer should use *partition pruning* to minimize I/O and use partition-wise joins as necessary.

The purpose of analyzing the partitioning method and partition key selection is to determine which flows within the Oracle E-Business Suite can potentially gain the most from the partitioning functionality.

There are a number of approaches than can be used to determine the most common access path used to access a particular table and hence, the partition key.

Example: Partitioning - AP_INVOICE_DISTRIBUTIONS_ALL (Oracle Payables)

For a larger Oracle E-Business Suite customer this table can grow upwards of 48GB in size, with over 94 million rows.

Functional Analysis

This table belongs to the Oracle Payables product. From a functionality perspective, this table holds the invoice distribution information that is either manually entered or system-generated. An invoice can comprise of one or more distributions, and like other detail tables, this table can become very large.

```
TABLE_NAME                       NUM_ROWS      BYTES         GB
------------------------------ ---------- ---------- ----------
AP_INVOICE_DISTRIBUTIONS_ALL   94,870,050 5.0869E+10     48.512
```

How is this table used?

This is one of the main Oracle Payables tables. The tables supporting product workbenches, such as the Invoice Workbench form, tend to grow very large and are therefore ideal candidates for partitioning. Typically, the information is accessed by concurrent programs and reports within the same module, but can also be accessed by other interrelated products.

The diagram on the below shows how the A`P_INVOICE_DISTRIBUTIONS_ALL` table is used by other products, such as Oracle Purchasing, Oracle Bill of Materials, and Oracle Project Accounting.

Foreign Key	Mand	Table	Join Condition
AIL_AIL_FK	Y	AP_INVOICE_LINES_ALL	AIL_ID2 = *AP_INVOICE_LINES_ALL*.ID
AP_INVOICE_DISTRIBUTIONS_FK1	Y	AP_INVOICES_ALL	INVOICE_ID = *AP_INVOICES_ALL*.INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_FK10	Y	FND_CURRENCIES	RECEIPT_CURRENCY_CODE = *FND_CURRENCIES*.CURRENCY_CODE
AP_INVOICE_DISTRIBUTIONS_FK11	Y	AP_ACCOUNTING_EVENTS_ALL	ACCOUNTING_EVENT_ID = *AP_ACCOUNTING_EVENTS_ALL*.ACCOUNTING_EVENT_ID
AP_INVOICE_DISTRIBUTIONS_FK12	N	AP_INCOME_TAX_REGIONS	INCOME_TAX_REGION = *AP_INCOME_TAX_REGIONS*.REGION_SHORT_NAME
AP_INVOICE_DISTRIBUTIONS_FK13	N	GL_CODE_COMBINATIONS	RATE_VAR_CODE_COMBINATION_ID = *GL_CODE_COMBINATIONS*.CODE_COMBINATION_ID
AP_INVOICE_DISTRIBUTIONS_FK14	N	GL_CODE_COMBINATIONS	PRICE_VAR_CODE_COMBINATION_ID = *GL_CODE_COMBINATIONS*.CODE_COMBINATION_ID
AP_INVOICE_DISTRIBUTIONS_FK15	N	AP_INVOICES_ALL	PARENT_INVOICE_ID = *AP_INVOICES_ALL*.INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_FK16	Y	AP_INVOICE_DISTRIBUTIONS_ALL	PREPAY_DISTRIBUTION_ID = *AP_INVOICE_DISTRIBUTIONS_ALL*.INVOICE_DISTRIBUTION_ID
AP_INVOICE_DISTRIBUTIONS_FK17	N	GL_USSGL_TRANSACTION_CODES	USSGL_TRANSACTION_CODE = *GL_USSGL_TRANSACTION_CODES*.USSGL_TRANSACTION_CODE
AP_INVOICE_DISTRIBUTIONS_FK18	N	GL_BC_PACKETS	PACKET_ID = *GL_BC_PACKETS*.PACKET_ID
AP_INVOICE_DISTRIBUTIONS_FK19	N	AP_AWT_GROUPS	AWT_GROUP_ID = *AP_AWT_GROUPS*.GROUP_ID
AP_INVOICE_DISTRIBUTIONS_FK2	Y	AP_TAX_CODES_ALL	TAX_CODE_ID = *AP_TAX_CODES_ALL*.TAX_ID
AP_INVOICE_DISTRIBUTIONS_FK20	N	AP_AWT_TAX_RATES_ALL	AWT_TAX_RATE_ID = *AP_AWT_TAX_RATES_ALL*.TAX_RATE_ID
AP_INVOICE_DISTRIBUTIONS_FK22	N	PA_TASKS	TASK_ID = *PA_TASKS*.TASK_ID
AP_INVOICE_DISTRIBUTIONS_FK23	N	PA_EXPENDITURE_TYPES	EXPENDITURE_TYPE = *PA_EXPENDITURE_TYPES*.EXPENDITURE_TYPE
AP_INVOICE_DISTRIBUTIONS_FK24	N	PA_EXP_ORGS_IT	EXPENDITURE_ORGANIZATION_ID = *PA_EXP_ORGS_IT*.ORGANIZATION_ID
AP_INVOICE_DISTRIBUTIONS_FK25	N	RCV_TRANSACTIONS	RCV_TRANSACTION_ID = *RCV_TRANSACTIONS*.TRANSACTION_ID
AP_INVOICE_DISTRIBUTIONS_FK26	Y	AP_INVOICES_ALL	AWT_INVOICE_ID = *AP_INVOICES_ALL*.INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_FK27	Y	AP_AWT_GROUPS	AWT_ORIGIN_GROUP_ID = *AP_AWT_GROUPS*.GROUP_ID
AP_INVOICE_DISTRIBUTIONS_FK28	N	AP_CREDIT_CARD_TRXNS_ALL	CREDIT_CARD_TRX_ID = *AP_CREDIT_CARD_TRXNS_ALL*.TRX_ID
AP_INVOICE_DISTRIBUTIONS_FK29	Y	AP_INVOICES_ALL	COMPANY_PREPAID_INVOICE_ID = *AP_INVOICES_ALL*.INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_FK3	Y	AP_INVOICE_PAYMENTS_ALL	AWT_INVOICE_PAYMENT_ID = *AP_INVOICE_PAYMENTS_ALL*.INVOICE_PAYMENT_ID
AP_INVOICE_DISTRIBUTIONS_FK30	N	AP_INVOICES_ALL	PRICE_CORRECT_INV_ID = *AP_INVOICES_ALL*.INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_FK4	N	AP_BATCHES_ALL	BATCH_ID = *AP_BATCHES_ALL*.BATCH_ID
AP_INVOICE_DISTRIBUTIONS_FK5	Y	GL_CODE_COMBINATIONS	DIST_CODE_COMBINATION_ID = *GL_CODE_COMBINATIONS*.CODE_COMBINATION_ID
AP_INVOICE_DISTRIBUTIONS_FK6	Y	PA_PROJECTS_ALL	PROJECT_ID = *PA_PROJECTS_ALL*.PROJECT_ID
AP_INVOICE_DISTRIBUTIONS_FK7	Y	GL_SETS_OF_BOOKS_11I	SET_OF_BOOKS_ID = *GL_SETS_OF_BOOKS_11I*.SET_OF_BOOKS_ID
AP_INVOICE_DISTRIBUTIONS_FK8	N	PO_DISTRIBUTIONS_ALL	PO_DISTRIBUTION_ID = *PO_DISTRIBUTIONS_ALL*.PO_DISTRIBUTION_ID
AP_INVOICE_DISTRIBUTIONS_FK9	N	GL_DAILY_CONVERSION_TYPES	EXCHANGE_RATE_TYPE = *GL_DAILY_CONVERSION_TYPES*.CONVERSION_TYPE

If customizations are present or if further clarification of the access paths and keys used is needed, then also review the SQL contained in the statspack/AWR repositories. The advantage of this approach is that often different sets of users will query or interact with the same information in different ways and then it is important to be aware of all the different ways a particular table is accessed.

Index Analysis

The next step is to check whether the indexed columns are being used across the majority of SQL statements. The Statspack/AWR repository can be useful in some cases to identify which particular SQL is accessing a table. In some cases the output from the Statspack/AWR report may be truncated depending on the length of the SQL, in which case you use the `V$SQL` table to get the complete SQL based upon the hash value shown in the Statspack/AWR report and then use the explain plan tool to generate an explain plan. In most cases it is better to take several traces of different flows that make use of the table in question.

The following table shows that composite indexes that include the `INVOICE_ID` column on the leading edge in most cases have the highest number of distinct keys (DK).

Index Name	Rows	DK	Column Name
AP_INVOICE_DISTRIBUTIONS_N13		15.82m	133562 PROJECT_ID TASK_ID
AP_INVOICE_DISTRIBUTIONS_N14		92.32m	40893 PA_ADDITION_FLAG PROJECT_ID REQUEST_ID
AP_INVOICE_DISTRIBUTIONS_N15		85389	64627 AWT_INVOICE_PAYMENT_ID
AP_INVOICE_DISTRIBUTIONS_N16		250673	250664 AWT_INVOICE_ID
AP_INVOICE_DISTRIBUTIONS_N17		10	8 RCV_TRANSACTION_ID
AP_INVOICE_DISTRIBUTIONS_N18		91.04m	2.19m ACCOUNTING_EVENT_ID
AP_INVOICE_DISTRIBUTIONS_N19 INVENTORY_TRANSFER_STATUS		66.2m	1
AP_INVOICE_DISTRIBUTIONS_N2	97.41m	2	POSTED_FLAG
AP_INVOICE_DISTRIBUTIONS_N20		62518	27319 PREPAY_DISTRIBUTION_ID
AP_INVOICE_DISTRIBUTIONS_N21		0	0 AWARD_ID
AP_INVOICE_DISTRIBUTIONS_N23		92.16m	92.16m RELATED_ID
AP_INVOICE_DISTRIBUTIONS_N24 CORRECTED_INVOICE_DIST_ID		0	0
AP_INVOICE_DISTRIBUTIONS_N25		734737	82280 PARENT_REVERSAL_ID
AP_INVOICE_DISTRIBUTIONS_N26		103.16m	10.28m OLD_DISTRIBUTION_ID
AP_INVOICE_DISTRIBUTIONS_N27		**99.52m**	**71.18m INVOICE_ID** **OLD_DIST_LINE_NUMBER**
AP_INVOICE_DISTRIBUTIONS_N28 CHARGE_APPLICABLE_TO_DIST_ID		29.24m	1.55m
AP_INVOICE_DISTRIBUTIONS_N29		14.68m	14.68m DETAIL_TAX_DIST_ID
AP_INVOICE_DISTRIBUTIONS_N3	94.73m	201701	DIST_CODE_COMBINATION_ID
AP_INVOICE_DISTRIBUTIONS_N30		0	0 BC_EVENT_ID
AP_INVOICE_DISTRIBUTIONS_N4	93.25m	6759	ACCOUNTING_DATE
AP_INVOICE_DISTRIBUTIONS_N5	87.08m	157980	BATCH_ID
AP_INVOICE_DISTRIBUTIONS_N6	95.78m	88	ASSETS_ADDITION_FLAG ASSETS_TRACKING_FLAG POSTED_FLAG ORG_ID
AP_INVOICE_DISTRIBUTIONS_N7	5.52m	316273	PO_DISTRIBUTION_ID
AP_INVOICE_DISTRIBUTIONS_U1	**93.75m**	**93.75m**	**INVOICE_ID** **INVOICE_LINE_NUMBER** **DISTRIBUTION_LINE_NUMBER**
AP_INVOICE_DISTRIBUTIONS_U2	95.24m	95.24m	INVOICE_DISTRIBUTION_ID

An example of SQL that accesses this table is as follows:

```
select sum(--decode(aphd.pay_dist_lookup_code,
                    --'EXCHANGE RATE VARIANCE', -1 * aphd.amount,
                    aphd.amount)
      from   ap_payment_hist_dists aphd,
             ap_invoice_distributions_all aid,
             ap_payment_history_all aph
      where  aid.invoice_id = p_inv_rec.invoice_id
      and    aid.invoice_distribution_id = aphd.invoice_distribution_id
      and    aphd.pay_dist_lookup_code in ('CASH', 'DISCOUNT', 'AWT')
      and    aph.posted_flag = 'Y'
      and    aph.payment_history_id = aphd.payment_history_id
      and    aph.transaction_type in
             ('PAYMENT CLEARING', 'PAYMENT UNCLEARING',
```

```
                                                        'PAYMENT CLEARING ADJUSTED')
```

the explain plan is shown in the following table.

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                     |  Name                          | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                                |     1 |    63 |   211   (1)|
|   1 |  SORT AGGREGATE               |                                |     1 |    63 |            |
|   2 |   NESTED LOOPS                |                                |    44 |  2772 |   211   (1)|
|   3 |    NESTED LOOPS               |                                |    44 |  1672 |   123   (1)|
|   4 |     TABLE ACCESS BY INDEX ROWID| AP_INVOICE_DISTRIBUTIONS_ALL  |    44 |   616 |    14   (0)|
|*  5 |      INDEX RANGE SCAN         | AP_INVOICE_DISTRIBUTIONS_U1    |    44 |       |     4   (0)|
|*  6 |     TABLE ACCESS BY INDEX ROWID| AP_PAYMENT_HIST_DISTS         |     1 |    24 |     3   (0)|
|*  7 |      INDEX RANGE SCAN         | AP_PAYMENT_HIST_DISTS_N2       |     1 |       |     2   (0)|
|*  8 |    TABLE ACCESS BY INDEX ROWID | AP_PAYMENT_HISTORY_ALL        |     1 |    25 |     2   (0)|
|*  9 |     INDEX UNIQUE SCAN         | AP_PAYMENT_HISTORY_U1          |     1 |       |     1   (0)|
-------------------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
   5 - access("AID"."INVOICE_ID"=TO_NUMBER(:B0))
   6 - filter("APHD"."PAY_DIST_LOOKUP_CODE"='AWT' OR "APHD"."PAY_DIST_LOOKUP_CODE"='CASH' OR
              "APHD"."PAY_DIST_LOOKUP_CODE"='DISCOUNT')
   7 - access("AID"."INVOICE_DISTRIBUTION_ID"="APHD"."INVOICE_DISTRIBUTION_ID")
   8 - filter(("APH"."TRANSACTION_TYPE"='PAYMENT CLEARING' OR "APH"."TRANSACTION_TYPE"='PAYMENT
              CLEARING ADJUSTED' OR "APH"."TRANSACTION_TYPE"='PAYMENT UNCLEARING')
                                 AND "APH"."POSTED_FLAG"='Y')
   9 - access("APH"."PAYMENT_HISTORY_ID"="APHD"."PAYMENT_HISTORY_ID")
```

The choice for partition key and partition strategy is determined by analyzing how the data is accessed. This analysis provides clues to the partitioning approach that will provide the best performance.

Conclusion

The `INVOICE_ID` is a good column to use a partition key, given its functional importance and frequent usage. In addition, it is fairly unique and has a high degree of selectivity so the optimizer can make use of partition pruning. We now have to decide on what partitioning method to use. List partitioning can be ruled out as it doesn't make sense to use it in this scenario. This only leaves hash or range partitioning. Since there is a natural partitioning key and given that the vast majority of access to this table will be made by `INVOICE_ID`, we use range partitioning.

Examples of Partition Keys for Oracle Applications Tables

Table name	Table Description	Suggested Partitioning Method	Suggested Partition Key	Description
AP_INVOICES_ALL	Contains one row for each invoice.	Range	INVOICE_ID	This is the primary key for this table and most queries against this table will have this column included, as this is the only way to uniquely identify an invoice.
RA_CUST_TRX_LINE_GL_DIST_ALL	Holds accounting records for revenue, unearned revenue, and unbilled receivables for each invoice or credit memo line. Oracle Receivables creates one row for each accounting distribution. At least one accounting distribution must exist for each invoice or credit memo line.	Range or List	SET_OF_BOOKS_ID	The financial calendar is broken into a fixed number of periods. Although this is a very unselective column due to the low number of distinct values, this column allows us to naturally segregate the data based on a time/period component. This will be especially advantageous for concurrent programs, which access a particular SET_OF_BOOKS_ID.
GL_IMPORT_REFERENCES	Stores individual transactions from sub-ledgers that have been summarized into Oracle General Ledger journal entry lines through the Journal Import process.	Range	JE_HEADER_ID	The GL data model is comprised of Batches, Headers, and Lines. Typically when joining to these tables, either directly or top-down (e.g. from batches) or bottom-up (e.g. from line level), in the majority of cases this column will be used to uniquely identifies a header record.
AP_AE_LINES_ALL	Stores the accounting representation for a particular transaction in terms of balanced, debits or credits entries. These are both in transaction currency as well as functional currency. Additionally, other important information such as the account and other reference information pointing to the original transaction is also stored.	Range or Hash	AE_LINE_ID	This is the primary key for this table. This column will uniquely identify a line record in the table. In most cases this column is used to join to this table and certainly can benefit from partitioning as CBO could partition prune based on AE_LINE_ID.
PA_EXPENDITURE_ITEMS_ALL	Stores the smallest categorized expenditure units charged to projects and tasks.	Range	EXPENDITURE_ITEM_ID	This is a system generated number that uniquely identifies the expenditure item and is used a common access and filter predicate of this table.

Table name	Table Description	Suggested Partitioning Method	Suggested Partition Key	Description
PA_EXPENDITURE_ITEMS_ALL		Hash	ORGANIZATION_ID	This column identifies the organization that owns the non–labor resource that was utilized as the work was performed. This column is only populated for usage items. Given that each row in this table will be in the context of an ORGANIZATION_ID, this column provides a way to naturally segregate the data.
GL_BALANCES	Stores actual, budget and encumbrance balances for detail and summary accounts. Also ledger currency, foreign currency, and statistical balances for each accounting period that has ever been opened.	Range	PERIOD_NAME	The PERIOD_NAME name column is the most commonly used to join to this table, as all the rows in this table are always in the context of a GL_PERIOD. **Note:** An alternative way of partitioning this table would be to use list partitioning and still use the PERIOD_NAME column as the partition key. This will of course mean there will be more partitions as each PERIOD_NAME will have its own unique partition.
OE_ORDER_LINES_ALL	Stores information for all order lines in Oracle Order Management	List	OPEN_FLAG	Popular filter condition used on this table as functionally orders can be OPEN or CLOSED.
		Hash	LINE_ID	This column will uniquely identify an order line and therefore any joins to this table will include this column.
PA_COST_DISTRIBUTION_LINES_ALL	Stores information about the cost distribution of expenditure items. When a cost distribution program processes an expenditure item, it creates one or more corresponding cost distribution lines to hold the cost amounts and the General Ledger account information to which the cost amounts will post. Cost distribution lines amount are implicitly debit amounts.	Range	EXPENDITURE_ITEM_ID	This column is often used as an access or filter predicate; by using range partitioning the CBO can ignore those partitions which don't contain the value of the EXPENDITURE_ITEM_ID.
			ORGANIZATION_ID	Functionally all the expenditure items will belong to a particular organization, and therefore this is a natural way of segregating the data.
RA_CUSTOMER_TRX_LINES_ALL	Stores information about invoice, debit memo, credit memo, bills receivable, and commitment lines. For example, an invoice can have one line for	Range	CUSTOMER_TRX_LINE_ID	System generated Invoice line identifier that uniquely identifies a customer transaction line. This column is used for common access &

Table name	Table Description	Suggested Partitioning Method	Suggested Partition Key	Description
	Product A and another line for Product B. Each line requires one row in this table.			filter predicates and could be used by the optimizer to disregard partitions that do not contain the value of the CUTOMER_TRX_LINE_ID.
WF_ITEM_ATTRIBUTE_VALUES	Contains the data for the attributes defined in the WF_ITEM_ATTRIBUTES table.	Range/Hash	ITEM_TYPE/ITEM_KEY	Queries against this table will usually be specific to an ITEM_TYPE. Typically, customers will only use 5 different values for ITEM_TYPE. However when combined with the ITEM_KEY column that has a high number of distinct values and selectivity, we get a more efficient partitioning key that gives us the best of both partitioning methods.

PRACTICAL PARTITIONING CASE STUDY

In this section two customer product specific examples are given to demonstrate further how of a particular table is partitioned.

Oracle General Ledger

The customer's Financial Management System (FMS) database is currently 340 GB in size and the General Ledger represents 90% of this data. The database is dominated by three large GL tables and their associated indexes: GL_JE_LINES, GL_BALANCES, and GL_DAILY_BALANCES.

The customer wants to get significant benefits in terms of manageability and performance by partitioning these objects. This case study reviews the partitioning strategy for the large GL tables such as GL_JE_LINES, GL_BALANCES and GL_DAILY_BALANCES.

Background – Current Table Volumes

GL_JE_LINES

119 GB table – approx. 360,000,000 records (50 % of database)

GL_BALANCES

20 GB table – approx. 170,000,000 records (8% of database)

GL_DAILY_BALANCES

16 GB table – approx. 55,000,000 records (6% of database)

Strategy

Customer identified the partition key as PERIOD_NAME due to the selectivity of the column. Customer's column PERIOD_NAME is in the format MON-RR, e.g. 'MAY-00'

The following statements show the partition range values necessary to achieve a workable range of values for the character column PERIOD_NAME.

```
create table GL_JE_LINES (
….
….
)
partition by range ( period_name )
(
partition APR00 values less than ('APR-01') tablespace data_apr00,
partition APR_OTHER values less than ('APR-50') tablespace
data_apr_other,
partition APR98 values less than ('APR-99') tablespace data_apr98,
partition APR99 values less than ('APR-AA') tablespace data_apr99,
partition AUG00 values less than ('AUG-01') tablespace data_aug00,
partition AUG_OTHER values less than ('AUG-50') tablespace
data_aug_other,
partition AUG98 values less than ('AUG-99') tablespace data_aug98,
partition AUG99 values less than ('AUG-AA') tablespace data_aug99,
partition DEC00 values less than ('DEC-01') tablespace data_dec00,
partition DEC_OTHER values less than ('DEC-50') tablespace
data_dec_other,
```

```
partition DEC98 values less than ('DEC-99') tablespace data_dec98,
partition DEC99 values less than ('DEC-AA') tablespace data_dec99,
..
..
..
partition SEP99 values less than ('SEP-AA') tablespace data_sep99,
partition OTHER values less than MAXVALUES tablespace data_other
);
```

As an example, the partition called 'APR00' will contain all the rows where period_name = 'APR-00', and the partition name 'APR99' using the literal value 'APR-AA' is required to capture rows where period_name = 'APR-99'. Note that the partition called 'APR_OTHER' will contain any future April periods, i.e. 'APR-01', 'APR-02', etc. In fact, each period has a partition (<MON>_OTHER) to catch any future rows for that period.

The tables GL_BALANCES and GL_DAILY_BALANCES will be partitioned on PERIOD_NAME using the same strategy as used with GL_JE_LINES.

Partition Maintenance

1. At month-end, the customer needs to create a new partition prior to the Open Period process.

   In the following example, the customer creates a new partition for APR-01 data:

   ```
   ALTER TABLE GL_JE_LINES
   split partition APR_OTHER
   at ('APR-02')
   into (partition APR01 tablespace data_apr01,
         partition APR_OTHER1 tablespace data_apr_other1 );
   ```

   Due to invalidation, the index will need to be rebuilt.

   Note with the SPLIT command, the 'bucket' partition APR_OTHER cannot retain its name hence it would be split into two (2) different names: APR_OTHER1, which will become the new 'bucket' for future April periods.

   This process would also take place for GL_BALANCES and GL_DAILY_BALANCES.

2. For purging data (based on PERIOD_NAME), just use the DROP partition command after the data has been archived.

   ```
   alter table gl_je_lines
   drop partition SEP-98;
   ```

   This would also drop any local indexes associated with this partition.

Index Strategy

The indexes associated with these tables are also very large and need to be created as either local or global indexes depending upon their usage.

GL_JE_LINES Indexes

- **Global Index**:
  GL_JE_LINES_N1 – 17GB (CODE_COMBINATION, PERIOD_NAME)
- **Local Index:**
  GL_JE_LINES_U1 – 12GB (JE_HEADER_ID, JE_LINE_NUM)

GL-BALANCES Indexes

- **Global Index:**
  GL_BALANCES_N1 – 5GB (CODE_COMBINATION, PERIOD_NAME)
- **Local Index:**
  GL_BALANCES_N2 – 4GB (PERIOD_NAME)
- **Global Index:**
  GL_BALANCES_N3 – 4GB (PERIOD_NUM, PERIOD_YEAR)
- **Global Index**:
  GL_BALANCES_N4 – 590MB (TEMPLATE_ID)

GL_DAILY_BALANCES Indexes

- **Global Index**:
  GL_DAILY_BALANCES_N1 – 2GB
  (CODE_COMBINATION, PERIOD_NAME, CURRENCY_CODE)
- **Local Index**:
  GL_DAILY_BALANCES_N2 – 1.2GB (PERIOD_NAME)
- **Global Index**:
  GL_DAILY_BALANCES_N3 – 1.2GB
  (PERIOD_YEAR, PERIOD_NUM)
- **Global Index**:
  GL_DAILY_BALANCES_N4 – 152MB (TEMPLATE_ID)

The Benefits of This Partitioning Strategy

The following points summarize the main benefits:

1. Maintenance - Operations can take place on individual partitions.
2. Purge of historic data - Prior to partitioning the purge process for GL_JE_LINES took approximately 36-48 hours to purge one month of data. Partitioning means that it is possible to simply drop the partition and deal with any global indexes.
3. Performance - Performance improvements of SQL through partition pruning. Much of the Oracle E-Business Suite code accesses these large tables by PERIOD_NAME. Some large process such as summary accounts and open period should benefit from partitioning by period.

4. Read-only partitions – This customer placed old period partitions in read only tablespaces as this reduced the cost of storage. This also has other advantages as we don't need to backup read-only tablespaces.

Oracle Payables

Background

A customer wanted to load multiple payment batches and transfer them to GL in parallel. The main strategy was to partition by payment batch from loading to accounting, this meant partitioning the main transaction tables by the BATCH_ID column or equivalent. The requirement was to have independent parallel processing in AP for payment batches. Given the current functionality of the product, for the transfer to GL the customer needed to investigate the possibility of running multiple batch posting or a single days post. In addition, Oracle Cash Manager was used to for check reconciliation.

Payment batch postings allow parallel continuous runs of the load-pay-account-transfer process. The single batch approach needs to wait until all processes have finished the AP accounting stage before running a single 'AP transfer to GL' program. The functional requirement was that the AP/GL transfer logic must be able to scan all partitions to summarize by account code during the transfer to GL. To improve performance, it would be possible to accommodate the single transfer to GL in the partitioning strategy without affecting the performance of the load-pay-account phases. Since the load-pay-account is critical, this took precedence.

Data Analysis

The `CHECK_ID` and `INVOICE_ID` columns or related columns make up all or part of the key of the tables partitioned by `BATCH_ID` . These are split across 64 hash partitions, with the option of moving towards 128 or beyond. Initially, the customer considered using a sequence dispersal technique for the appropriate `CHECK_ID` and `INVOICE_ID` columns, but instead decided to use Global Indexes, which are available in 10gR2. This allowed an alternative partitioning strategy using the `CHECK_ID` and `INVOICE_ID` columns with a number of range partitions (98). As a number of concurrent processes insert into this table at once and select from the same sequence, the indexes get very hot as new entries are being inserted into the end section of the index. Rather than using a technique to disperse the sequence, Oracle 10g2 and use Global Partitioned Indexes were proposed as a solution.

Results

The following table shows the results of partitioning the Payables tables:

Object Name	Type	Column(s)	Partition Type & Size
AP_ACCOUNTING_EVENTS_ALL	TABLE	ACCOUNTING_EVENT_ID	Replace with HASH partition * 64
AP_ACCOUNTING_EVENTS_U1	INDEX	ACCOUNTING_EVENT_ID	LOCAL Hash 64
AP_ACCOUNTING_EVENTS_N1	INDEX	SOURCE_TABLE, SOURCE_ID	Make GLOBAL HASH Partition * 64
AP_ACCOUNTING_EVENTS_N2	INDEX	ACCOUNTING_DATE	LOCAL Hash 64
AP_ACCOUNTING_EVENTS_N3	INDEX	REQUEST_ID	LOCAL Hash 64
AP_ACCOUNTING_EVENTS_N4	INDEX	EVENT_STATUS_CODE	LOCAL Hash 64
AP_AE_HEADERS_ALL	TABLE	REQUEST_ID	Hash 64
AP_AE_LINES_ALL	TABLE	REQUEST_ID	Hash 64
AP_CHECKS_ALL	TABLE	CHECKRUN_NAME	Hash 64
AP_CHECKS_N3	INDEX	CHECKRUN_NAME	LOCAL Hash 64
AP_CHECKS_N9	INDEX	CHECK_NUMBER	LOCAL Hash 64
AP_CHECKS_N4	INDEX	PAYMENT_TYPE_FLAG	LOCAL Hash 64
AP_CHECKS_N8	INDEX	CHECKRUN_ID	LOCAL Hash 64
AP_CHECKS_U1	INDEX	CHECK_ID	Make GLOBAL HASH partitioned index * 64
AP_INVOICES_ALL	TABLE	BATCH_ID	Hash 64
AP_INVOICES_N1	INDEX		LOCAL Hash 64
AP_INVOICE_DISTRIBUTIONS_ALL	TABLE	BATCH_ID	Hash 64
AP_INVOICE_DISTRIBUTIONS_N14	INDEX		LOCAL Hash 64
AP_INVOICES_INTERFACE	TABLE	SOURCE	Range 0001..0064 (64 partitions) + 1 Catch all for 65 to 1024
AP_INVOICES_INTERFACE_N1	INDEX	STATUS	Index Dropped, need to check if required by Oracle Custom layer.
AP_INVOICES_INTERFACE_N2	INDEX	SOURCE GROUP_ID	Index Dropped, need to check if required by Oracle Custom layer.
AP_INVOICES_INTERFACE_U1	INDEX	INVOICE_ID	Make GLOBAL HASH, 64
AP_INVOICE_PAYMENTS_ALL	TABLE	INVOICE_PAYMENT_ID	Consider increasing to 128 partitions
AP_INVOICE_PAYMENTS_N1	INDEX	INVOICE_ID	Make GLOBAL HASH partitioned index * 64

CONCLUSION

Partitioning is a very powerful database feature that can return significant gains when applied using an appropriate strategy.

The use of *custom partitioning* within the Oracle E-Business Suite is fully supported and is something that should be explored. This paper introduced partitioning for customers of Oracle's E-Business Suite using practical examples from the Oracle E-Business Suite.

Partitioning is a very powerful database feature . This paper has explained database partitioning and provided examples based from the Oracle E-Business Suite. Partitions can be thought of as a layer between tables/indexes and tables spaces. The advantage of partitioning is that optimizer is partition aware and excludes unnecessary partitions from SQL transactions, therefore resulting in a performance increase. The paper has also described the different functions and limitations of each partitioning strategy and also reviewed compression as an complementary strategy.

The "bigger, better, and faster" hardware approach used by many customers to address the challenges of increasing data volumes provides short-term relief, but increases the total cost of ownership of an Oracle E-Business Suite implementation. Upgrades may also require more careful planning due to volumes of data involved.

Predicting how large a particular Oracle E-Business Suite implementation is going to be is generally undertaken as a part of a standard sizing exercise. When very large volumes of data are predicted, it makes sense to additionally consider partitioning, and which strategy may be the most appropriate.

This paper has shown that in order to address both the space management and performance needs, consider the normal daily business requirements to specific subsets of data and define how that data is accessed. This will help define an approach to effective deployment of the partitioning features, where tables and indexes are broken down into smaller components. Partitioning is a very powerful feature that can be implemented to reduce the disk costs by allowing you to distribute tablespaces across a range of storage devices. Data that is rarely updated can then be compressed to further reduce storage requirements.

In the second part of this paper , examples of partitioning tables and indexes in the Oracle E-Business Suite have been given. The criteria for correct partition key and partitioning method selection and other aspects such as migrating data into the newly partitioned tables/indexes have also been described. It has also discussed the additional partitioning maintenance operations that need to be considered.

This paper has described the following:

- Different database partitioning methods available
- When to use database partitioning for the Oracle E-Business Suite
- The steps need to be adhered to when partitioning
    - Including data migration
- Examples of partitioning methods for some of the common product tables
- Customer partitioning case studies

REFERENCES

[1] Ahmed Alomari & Mohsin Sameen Oracle OpenWorld (2006):
Partitioning and Purging Best Practices for Oracle E-Business Suite
Available from: http://blogs.oracle.com/schan/2006/11/17

[2] Oracle Corporation:
Oracle® Database Concepts 10g Release 2 (10.2)
Chapter 18 Part Number B14220-02

[3] Oracle Corporation:
Oracle® Database Administrator's Guide 10g Release 2 (10.2)
Chapter: 17 Part Number B14231-02

[4] Metalink: 248857.1
Oracle Applications Tablespace Model Release 11i - Tablespace Migration Utility

REVISIONS

This white paper was first issued in February 2008.

APPENDIX A - ORACLE DATA DICTIONARY TABLE & VIEWS FOR PARTITIONING

Summary of ALTER TABLE & ALTER INDEX clauses used for maintenance operation for table and index partitions:-

- `Add Partition` – To add a new partition after the highest partition
- `Drop Partition` – To drop a partition including its indexes
- `Exchange Partition` – To change a partition into a non-partitioned table and vice versa
- `Merge Partition` – To merge two adjacent partitions
- `Modify Partition` – To modify the storage parameters of a partition
- `Modify default attributes` – To modify the storage parameters of all partitions
- `Move Partition` – To move a partition generally to defrag or to change tablespace
- `Rename Partition` – Change name of a partition
- `Split Partition` – Split a partition into two
- `Truncate Partition` – Remove data of a partition

The following clauses are allowed with `ALTER INDEX`

- `Drop Partition` – Only a global index partition can be dropped
- `Modify Partition` – To modify the storage parameters of a partition
- `Modify default attributes` – To modify the storage parameters of all partitions
- `Rebuild Partition` – To rebuild a partition generally to defrag or to change
- `Rename Partition` – Change name of a partition
- `SPLIT PARTITION` – Split a partition into two
- `UNUSABLE` – To set some or all index partition as unusable

APPENDIX B - USEFUL DATABASE VIEWS

The following database view can be used to view partitioned tables and index information.

View	Description
DBA_TABLES	Table structure, Partition Y/N
DBA_PART_TABLES	DBA view displays partitioning information for all partitioned tables in the database. Partition type, default values
ALL_PART_TABLES USER_PART_TABLES	ALL view displays partitioning information for all partitioned tables accessible to the user. USER view is restricted to partitioning information for partitioned tables owned by the user
DBA_TAB_PARTITIONS ALL_TAB_PARTITIONS USER_TAB_PARTITIONS	Display partition-level partitioning information, partition storage parameters, and partition statistics generated by the DBMS_STATS package or the ANALYZE statement.
DBA_TAB_SUBPARTITIONS ALL_TAB_SUBPARTITIONS USER_TAB_SUBPARTITIONS	Display subpartition-level partitioning information, subpartition storage parameters, and subpartition statistics generated by the DBMS_STATS package or the ANALYZE statement.
DBA_PART_KEY_COLUMNS ALL_PART_KEY_COLUMNS USER_PART_KEY_COLUMNS	Display the partitioning key columns for partitioned tables.
DBA_SUBPART_KEY_COLUMNS ALL_SUBPART_KEY_COLUMNS USER_SUBPART_KEY_COLUMNS	Display the subpartitioning key columns for composite-partitioned tables (and local indexes on composite-partitioned tables).
DBA_PART_COL_STATISTICS ALL_PART_COL_STATISTICS USER_PART_COL_STATISTICS	Display column statistics and histogram information for the partitions of tables.
DBA_SUBPART_COL_STATISTICS ALL_SUBPART_COL_STATISTICS USER_SUBPART_COL_STATISTICS	Display column statistics and histogram information for subpartitions of tables.
DBA_PART_HISTOGRAMS ALL_PART_HISTOGRAMS USER_PART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table partitions.
DBA_SUBPART_HISTOGRAMS ALL_SUBPART_HISTOGRAMS USER_SUBPART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table subpartitions.
DBA_PART_INDEXES ALL_PART_INDEXES USER_PART_INDEXES	Display partitioning information for partitioned indexes.
DBA_IND_PARTITIONS ALL_IND_PARTITIONS USER_IND_PARTITIONS	Display the following for index partitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_IND_SUBPARTITIONS ALL_IND_SUBPARTITIONS USER_IND_SUBPARTITIONS	Display the following information for index subpartitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_SUBPARTITION_TEMPLATES ALL_SUBPARTITION_TEMPLATES USER_SUBPARTITION_TEMPLATES	Display information about existing subpartition templates.

Partitioning for the Oracle E-Business Suite
February 2008
Author: Mohsin Sameen
Primary Contributors: Andy Tremayne
Reviewers: Lester Gutierrez, Isam Alyousfi, Hadi Alatasi, Jin Soo Eo, Olcay Sarioglu, Dimas Chabane

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2008, Oracle. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.