Oracle Partitioning – an Introduction

Hermann Bär
Partitioning Product Management
Oracle USA
Redwood Shores, CA USA

Schlüsselworte
Partitioning, performance, data management, large data volumes

Introduction

Oracle Partitioning enables large tables and indexes to be subdivided in smaller pieces, improving the performance, manageability, and availability for tens of thousands of applications. Queries and maintenance operations are sped up by an order of magnitude for mission critical systems of any shape – OLTP, data warehousing, or mixed workloads – and any size – from hundreds of Gigabytes to Petabytes.

Partitioning is a key tool for building large systems or systems with extreme high availability requirements. Moreover, partitioning can greatly reduce the total cost of data ownership, using a "tiered archiving" approach of keeping older relevant information still online on low cost storage devices in the most optimal compressed format. When used together with Automatic Data Optimization and Heat Map, new functionality introduced in Oracle Database 12*c*, Partitioning provides a simple and automated way to implement an Information Lifecycle Management (ILM) strategy.

Concept of Partitioning

Partitioning enables tables and indexes to be subdivided into individual smaller pieces. Each piece of the database object is called a partition. A partition has its own name, and may optionally have its own storage characteristics.

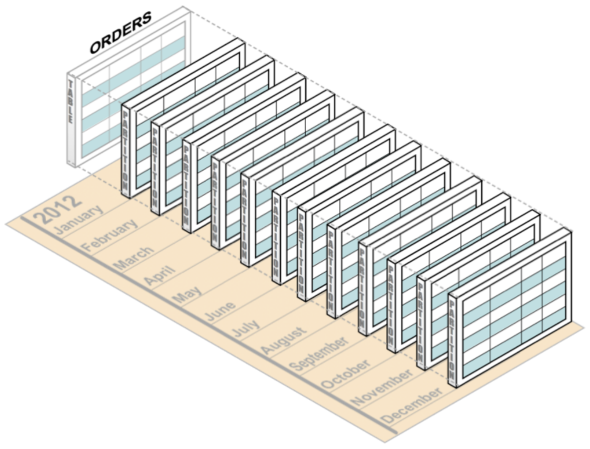

Figure 1: Application and DBA view of a partitioned table

From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives the administrator considerable flexibility in managing a partitioned object. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands. Logically, it is still only one table.

Database objects – tables and indexes - are partitioned using a **partitioning key**, a set of columns that determine in which partition a given row will reside (in case of a composite partitioned table, a partition is further subdivided into subpartitions, using a second set of columns for further subdivision; the data placement of a given row is then determined by both partitioning key criteria and placed in the appropriate subpartition [1)]. For example the Orders table shown in Figure 1 is range-partitioned on order date, using a monthly partitioning strategy; the table appears to any application as a single, 'normal' table. However, the database administrator can manage and store each monthly partition individually, potentially using different storage tiers, applying table compression to the older data, or store complete ranges of older data in read only tablespaces. Application developers generally do not have to worry about whether or not a table is partitioned, but they also can leverage partitioning to their advantage: for example a resource intensive DML operation to purge data from a table can be implemented using partition maintenance operations, improving the runtime dramatically while reducing the resource consumption significantly.

Partitioning Functionality at a Glance

The following table shows all available basic partitioning strategies in Oracle Database 12*c*:

BASIC PARTITIONING IN ORACLE DATABASE 12C

PARTITIONING STRATEGY	**DATA DISTRIBUTION**	**SAMPLE BUSINESS CASE**
Range Partitioning	Consecutive ranges of values.	Orders table range partitioned by order_date
List Partitioning	Unordered lists of values.	Orders table list partitioned by country
Hash Partitioning	Internal hash algorithm	Orders table hash partitioned by customer_id
Composite Partitioning • Range-Range • Range-List • Range-Hash • List-List • List-Range • List-Hash • Hash-Hash • Hash-List • Hash-Range	Combination of two of the above-mentioned basic techniques of Range, List, and Hash	Orders table is range partitioned by order_date and sub-partitioned by hash on customer_id Orders table is range partitioned by order_date and sub-partitioned by range on shipment_date Orders table is list partitioned by country and sub-partitioned by range on order_date Orders table is list partitioned by country and sub-partitioned by hash on customer_id

[1] For simplicity reasons we will refer to partitions only for the rest of this document

In addition to the available partitioning strategies, Oracle Database 12*c* provides the following partitioning extensions:

PARTITIONING EXTENSIONS IN ORACLE DATABASE 12C

PARTITIONING EXTENSION	**DESCRIPTION**	**SAMPLE BUSINESS CASE**
Interval Partitioning • Interval • Interval-Range • Interval-List • Interval-Hash	Extension to Range Partition. Defined by an interval, providing equi-width ranges. With the exception of the first partition all partitions are automatically created on-demand when matching data arrives.	Orders table partitioned by order_date with a predefined daily interval, starting with '01-Jan-2013'
Reference Partitioning	Partitioning for a child table is inherited from the parent table through a primary key – foreign key relationship. The partitioning keys are not stored in actual columns in the child table.	(Parent) Orders table range partitioned by order_date and inherits the partitioning technique to (child) order lines table. Column order_date is only present in the parent orders table
Virtual column based Partitioning	Defined by any partition techniques where the partitioning key is based on a virtual column. Virtual columns are not stored on disk and only exist as metadata.	Orders table has a virtual column that derives the sales region based on the first three digits of the customer account number. The orders table is then list partitioned by sales region.

Indexing of partitioned table

Irrespective of the chosen table partitioning strategy, any index of a partitioned table is either coupled or uncoupled with the underlying partitioning strategy of its table. Oracle Database 12*c* differentiates between three types of indexes.

A **local index** is an index on a partitioned table that is coupled with the underlying partitioned table; the index 'inherits' the partitioning strategy from the table. Consequently, each partition of a local index corresponds to one - and only one - partition of the underlying table. The coupling enables optimized partition maintenance; for example, when a table partition is dropped, Oracle simply has to drop the corresponding index partition as well. No costly index maintenance is required since an index partition is by definition only tied to its table partition; a local index segment will never contain data of other partitions. Local indexes are most common in data warehousing environments.

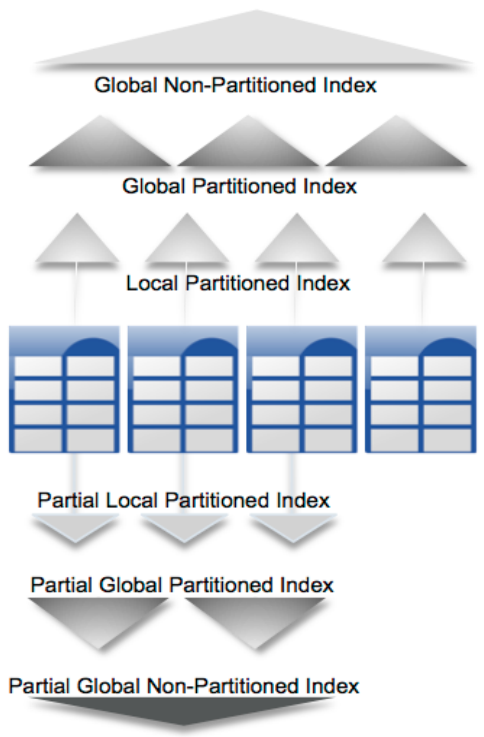
Global Non-Partitioned Index
Global Partitioned Index
Local Partitioned Index
Partial Local Partitioned Index
Partial Global Partitioned Index
Partial Global Non-Partitioned Index

Figure 2: Indexing on partitioned tables

A **global partitioned index** is an index on a partitioned or non-partitioned table that is partitioned using a different partitioning-key or partitioning strategy than the table. Global-partitioned indexes can be partitioned using range or hash partitioning and are uncoupled from the underlying table. For example, a table could be range-partitioned by month and have twelve partitions, while an index on that table could be hash-partitioned using a different partitioning key and have a different number of partitions. Decoupling an index from its table automatically means that any partition maintenance operation on the table can potentially cause index maintenance operations. Global partitioned indexes are more common for OLTP than for data warehousing environments.

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

All of the before-mentioned index types can be either created on all partitions of a partitioned table – so-called **full indexing** – or, beginning with Oracle Database 12*c*, created only on a subset of the partitions of a partitioned table – so-called **partial indexing**.

Partial indexing is an index attribute only applicable to indexes on partitioned tables and is complementary to the standard full indexing available for partitioned and non-partitioned tables. Whether or not a particular partition will be indexed or not is determined on a per-partition base and applied to all partial indexes. With partial indexing you can for example decide not to index older partitions where the data access is mostly or exclusively without filtering predicates. You can also choose not to index the most recent partition to avoid any index maintenance work at data insertion time, therefore maximizing data load speed.

The appropriate indexing strategy is chosen based on the business requirements and access patterns, making partitioning well suited to support any kind of application.

Partitioning for Performance

The placement of a given row is determined by its value of the partitioning key. How the data of a table is subdivided across the partitions is stored as partitioning metadata of a table or index. This metadata is used to determine for every SQL operation – queries, DML, and partition maintenance operations - what partitions of a table are relevant, and the database automatically only touches relevant partitions. By limiting the amount of data to be examined or operated on, partitioning provides a number of performance benefits.

Partition pruning (a.k.a. partition elimination) is the simplest and also the most effective means to improve performance. It can often improve query performance by several orders of magnitude by leveraging the partitioning metadata to only touch the data of relevance for a SQL operation. For example, suppose an application contains an Orders table containing an historical record of orders, and that this table has been partitioned by day. A query requesting orders for a single week would only access seven partitions of the Orders table. If the table had 2 years of historical data, this query would access seven partitions instead of 730 partitions. This query could potentially execute 100x faster simply because of partition pruning. Partition pruning works with all of Oracle's other performance features. Oracle will utilize partition pruning in conjunction with any indexing technique, join technique, or parallel access method.

Partitioning can also improve the performance of multi-table joins, by using a technique known as **partition-wise joins**. Partition-wise joins can be applied when two tables are being joined together, and at least one of these tables is partitioned on the join key. Partition-wise joins break a large join into smaller joins of 'identical' data sets for the joined tables. 'Identical' here is defined as covering exactly the same set of partitioning key values on both sides of the join, thus ensuring that only a join of these 'identical' data sets will produce a result and that other data sets do not have to be considered. Oracle is using either the fact of already (physical) equi-partitioned tables for the join or is transparently redistributing (= "repartitioning") one table – the smaller one - at runtime to create equi-partitioned data sets matching the partitioning of the other table, completing the overall join in less time, using less resources. This offers significant performance benefits both for serial and parallel execution.

Partitioning for Manageability

By partitioning tables and indexes into smaller, more manageable units, database administrators can use a "divide and conquer" approach to data management. Oracle provides a comprehensive set of SQL commands for managing partitioning tables. These include commands for adding new partitions, dropping, splitting, moving, merging, truncating, and exchanging partitions.

With partitioning, maintenance operations can be focused on particular portions of tables. For example, a database administrator could compress a single partition containing say the data for the year 2012 of a table, rather than compressing the entire table; as part of the compression operation, this partition could also being moved to a lower cost storage tier, reducing the total cost of ownership for the stored data even more. With Oracle Database 12*c* this partition maintenance operation can be done in a completely online fashion, allowing both queries and DML operations to occur while the data maintenance operation is in process.

Oracle Database 12*c* further allows partition maintenance operations on multiple partitions as single atomic operation: for example, you can merge the three partitions 'January 2012', 'February 2012', and 'March 2012' into a single partition 'Q1 2012' with a single merge partition operation.

Another typical usage of partitioning for manageability is to support a 'rolling window' load process in a data warehouse. Suppose that a DBA loads new data into a table on daily basis. That table could be range-partitioned so that each partition contains one day of data. The load process is simply the addition of a new partition. Adding a single partition is much more efficient than modifying the entire table, since the DBA does not need to modify any other partitions.

Removing data in a very efficient and elegant manner is another key advantage of partitioning. For example, to purge data from a partitioned table you simply drop or truncate one or multiple partitions, a very cheap and quick data dictionary operation, rather than issuing the equivalent delete command, using lots of resources and touching all the rows to being deleted. The common operation of removing data with a partition maintenance operation such as drop or truncate is optimized in Oracle Database 12*c*: these operations do not require any immediate index maintenance to keep all indexes valid, making it fast metadata-only operations.

Information Lifecycle Management with Partitioning

Today's challenge of storing vast quantities of data for the lowest possible cost can be optimally addressed by using Oracle Partitioning with Automatic Data Optimization and Heat Map, new functionality in the Advanced Compression option of Oracle Database 12*c*. The independence of individual partitions, together with efficient and transparent data maintenance operations for partitions, are key enablers for addressing the online portion of a “tiered archiving” strategy. Specifically in tables containing historical data, the importance - and access pattern – of the data heavily relies on the age of the data; Partitioning enables individual partitions (or groups of partitions) to be stored on different storage tiers, providing different physical attributes – such as compression - and price points. For example, in an Orders table containing 2 years worth of data, you could store only the most recent quarter on an expensive high-end storage tier and keep the rest of the table (almost 90% of the data) on an inexpensive low cost storage tier.

The addition of Automatic Data Optimization – ADO – allows you to define policies that specify when storage tiering and compression tiering should be implemented for a given partition, based on the usage statistics automatically collected by Heat Map. ADO policies are automatically evaluated and executed by the Oracle Database without any manual intervention required, making it possible to experience the cost savings and performance benefits of storage tiering and compression without creating complex scripts and jobs.

Conclusion

Oracle Partitioning is for everybody. Partitioning can greatly enhance the manageability, performance, and availability of almost any database application. Since partitioning is transparent to the application, it can be easily implemented for any kind of application because no costly and time-consuming application changes are required.

Kontaktadresse:
Hermann Bär
Oracle USA
400 Oracle Parkway
Redwood Shores, CA 94065, USA

Telefon:	+1 (0) 650.506.6833
Fax:	+1 (0) 650.506.6833
E-Mail	hermann.baer@oracle.com
Internet:	www.oracle.com