Servicio Andaluz de Salud
CONSEJERÍA DE SALUD

Oficina Técnica para la Gestión y Supervisión de Servicios TIC
Subdirección de Tecnologías de la Información

Implementación de particionamiento en Oracle. Buenas Prácticas para desarrolladores.

Referencia documento: InfV5_JASAS_Partitioning_Development_BestPractices_V920.doc
Fecha: 13 de noviembre de 2018
Versión: 9.2.0

Registro de Cambios

Fecha	Autor	Versión	Notas
17 de Junio de 2013	Isidro Granados	4.2.0	Versión inicial
17 de Octubre de 2013	Isidro Granados	4.3.0	Versión correspondiente a la entrega de Octubre de 2013
16 de Diciembre de 2015	Isidro Granados	6.2.0	Revisión de Diciembre de 2015, contrato 2014-2016
16 de Junio de 2016	Isidro Granados	7.1.0	Revisión de Junio de 2016, contrato 2014-2016
14 de octubre de 2016	Isidro Granados	7.2.0	Revisión de Noviembre de 2016, contrato 2014-2016
14 de junio de 2017	Isidro Granados	8.1.0	Revisión de Junio de 2017, contrato 2016-2018
16 de noviembre de 2017	Isidro Granados	8.2.0	Revisión de Noviembre de 2017, contrato 2016-2018
16 de junio de 2018	Isidro Granados	9.1.0	Revisión de junio de 2018, contrato 2016-2018
13 de noviembre de 2018	Isidro Granados	9.2.0	Revisión de noviembre de 2018, contrato 2016-2018

Revisiones

Nombre	Role
Jonathan Ortiz	Advanced Support Engineer
Gregorio Adame	Advanced Support Engineer
Jose María Gómez	TAM

Distribución

Copia	Nombre	Empresa
1	Subdirección de Tecnologías de la Información	Servicio Andaluz de Salud, Junta de Andalucía
2	Servicio de Coordinación de Informática de la Consejería de Innovación	Consejería de Innovación, Junta de Andalucía

Índice de Contenidos

Control de cambios

Cambio	Descripción	Página
1	No se realizan cambios en esta versión del documento.	N/A

Introducción

Una base de datos Oracle puede ser realmente grande y no es difícil encontrar tablas que son capaces de ocupar varios gigabytes o incluso varios terabytes de datos. Particionamiento es la opción que ofrece el RDBMS Oracle para dividir lógicamente en trozos una tabla y un índice para facilitar el procesamiento y rendimiento de consultas y operaciones DML así como el manejo y mantenimiento de una base de datos con estas circunstancias.

Particionamiento es una opción que se ofrece como opción licenciable para Enterprise Edition. El particionamiento aparece por primera vez en la versión 8 del servidor de Oracle y ha ido evolucionando con cada nueva versión.

Esta opción nació pensada fundamentalmente para entornos Data Warehouse, en los que se manejan tablas de un tamaño considerable. Pero cualquier tipo de base de datos que contenga tablas de tamaño elevado se puede beneficiar de la utilización del particionamiento de Oracle. Estos entornos que manejan tan grandes cantidades de datos presentan serios problemas tanto de rendimiento como de mantenimiento, la opción de particionamiento de Oracle es la solución más adecuada para estos dos problemas.

Sin embargo, no todas las tablas y situaciones se pueden beneficiar del particionamiento. Un particionamiento inadecuado o innecesario puede suponer un empeoramiento en el rendimiento, además de un consumo innecesario de tiempo de trabajo del administrador de la base de datos.

La opción de Particionamiento viene ligada a concepto de Base de datos tipo VLDB (very large database). Una VLDB no tiene un mínimo absoluto de tamaño y aunque no deja de ser una base de datos Oracle hay varios aspectos específicos que aplican principalmente a una VLDB provocados por su gran tamaño y por el coste de realizar operaciones en un sistema de este tamaño.

Varias tendencias han sido responsables del crecimiento constante del tamaño de las base de datos:

- Durante mucho tiempo los sistemas han sido desarrollados de forma independiente. Los departamentos de IT han empezado a ver los beneficios de la consolidación, esto es, combinar estos sistemas y habilitar bases de datos interdepartamentales para reducir los costes de mantenimiento de estos sistemas independientes. La consolidación de base de datos y aplicaciones es un factor clave para el crecimiento continuo del tamaño de las bases de datos.
- Muchas compañias se enfrentan a Regulaciones que obligan a guardar datos durante un mínimo tiempo, cuyo resultado es que muchos datos son guardados durante largos periodos de tiempo.
- La compañias crecen expandiendo ventas u operaciones o a través de fusiones o adquisiciones provocando que la cantidad de datos generados y procesados se incremente.

El Particionamiento es una característica clave en el manejo de una VLDB. Particionamiento permite implementar la técnica de "Divide y Venceras" para tablas o indices muy grandes, especialmente aquellas en continuo crecimiento. Con esta técnica es posible en una base de datos mantener consistentemente el rendimiento y concurrencia a la vez de que se sigue creciendo en tamaño, esto sin tener que incrementar los recursos hardware o administrativos.

El particionamiento permite dividir en trozos más pequeños tablas, índices y tablas organizadas por índices (IOTs) de forma que puedan ser accedidos a un nivel más granular. Oracle dispone de una gran variedad de técnicas y extensiones de particionamiento de forma que se pueda adaptar a cualquier requerimiento de negocio.

El particionamiento es totalmente transparente, por tanto puede ser implementado en casi cualquier aplicación sin necesidad de realizar cambios en la aplicación.

Objetivos de este documento

Este documento trata de mostrar en detalle los conceptos y las técnicas necesarias para utilizar correctamente la opción de particionamiento en Oracle y poder maximizar su beneficio dentro de una base de datos así como minimizar los riesgos de una inadecuada implantación o uso de esta opción en el desarrollo de aplicaciones sobre bases de datos Oracle.

Una buena implantación de particionamiento tendrá los siguientes beneficios en nuestras aplicaciones:

- Mejoras en el Rendimiento
- Mejoras en el Mantenimiento
- Mejoras en la Disponibilidad

El documento se centra principalmente en la versión Oracle RDBMS 12cR2 (12.2.0.1), aunque muchos de los conceptos y recomendaciones son igualmente aplicables a las versiones anteriores. El documento irá indicando las versiones en las que las opciones disponibles han sido introducidas.

El documento mostrará con ejemplos distintos acercamientos para identificar que tablas e índices se podrían beneficiar del uso de particionamiento, así como recomendaciones o buenas prácticas a tener en cuenta en su implantación dependiendo del tipo de entorno o aplicación, esto es, tanto a entornos transaccionales OLTP como entornos Data Warehouse.

Conceptos de Particionamiento

Visión general particionamiento

El particionamiento consiste básicamente en dividir una tabla, índice o IOT en trozos más pequeños, cada trozo de un objeto particionado se llamará partición, o incluso subpartición.

Desde la perspectiva de un DBA, un objeto particionado podrá ser manejado de forma global o individualmente. Esto nos da una gran flexibilidad a la hora de utilizar el particionamiento y posibilita una serie de operaciones que facilitan notablemente las labores de mantenimiento de las bases de datos

Desde el punto de vista de la aplicación, una tabla particionada es idéntica a una no particionada, y no habrá que hacer cambios en consultas SQL o sentencias DML.

Cada partición tendrá su propio nombre, y se corresponderá físicamente con un segmento. Estos segmentos opcionalmente podrán tener sus propias características de almacenamiento, por ejemplo las particiones de un objeto particionado podrán residir en tablespaces distintos, con la restricción de que todos los tablespaces tengan el mismo tamaño de bloque.

Para tablas normales, una tabla es ambas cosas un objeto y un segmento. Para una tabla particionada, una tabla es sólo un objeto pero sin segmento. La tabla contendrá particiones que serán objetos y segmentos. Si la tabla es particionada compuesta, entonces las particiones solo son objetos sin un segmento asignado y las subparticiones serán los segmentos físicos.

Cada partición en una tabla o índice deberá tener los mismos atributos lógicos, como nombres de columnas, tipos de datos y "constraints". Por ejemplo, todas las particiones de una tabla compartirán la definición de las columnas, aunque cada partición podrá tener atributos físicos diferentes como el "tablespace" o el "pctfree".

Clave de Particionamiento (Partition Key)

El criterio para particionar la tabla se definirá sobre una o más columnas que la forman (esta columna o columnas formarán lo que se denomina como **clave de particionamiento**). Cada fila en una tabla particionada será asignada de forma inequívoca a una única partición. Para determinar en qué partición se guarda una fila se usará esta clave de particionamiento. Oracle automáticamente redirige los comandos insert, update y delete a la partición apropiada usando esta clave de particionamiento.

Por ejemplo en una tabla `ventas`, se podría usar la columna `time_id` como clave de particionamiento por rango. La base de datos asignará automáticamente las filas a las particiones basándose cuando una fecha está en un rango específico.

Beneficios del particionamiento.

El concepto de partición es interno y prácticamente invisible al usuario. El usuario se referirá a la tabla completa como si fuera un único objeto al igual que si la tabla fuera no particionada. Una tabla podría pasar de no particionada a particionada y viceversa sin que los usuarios finales tuvieran conocimiento de este hecho o hubiese que cambiar alguna SQL o aplicaciones. Sin embargo, el hecho de particionar una tabla ofrece diversos beneficios:

Mejoras en el Rendimiento

El particionamiento ofrece una serie de mejoras en cuanto al rendimiento ya que dada una consulta sobre la tabla particionada, el optimizador basado en costes puede eliminar de forma transparente las particiones que no contienen datos que satisfagan la consulta. Este mecanismo llamado "partition pruning" es transparente a la aplicación y mejorará considerablemente los tiempos de respuesta de la base de datos.

Otra ventaja es el hecho de que al tratarse cada partición de un segmento de datos diferenciado, se puede ubicar de forma de se eviten contenciones de E/S.

En algunos sistemas tipo OLTP, las particiones pueden ayudar a descender la contención de un recurso compartido. Por ejemplo, operaciones tipo DML se pueden distribuir sobre muchos segmentos en lugar de uno solo.

En un entorno Data Warehouse, el particionamiento puede incrementar la velocidad del procesamiento de consultas grandes ya que las operaciones podrán paralelizarse y ejecutarse concurrentemente en cada partición. Estas particiones podrán incluso localizarse en físicamente en diferentes sitios mejorando la concurrencia.

Mejoras en el Mantenimiento y Disponibilidad

Cada partición se puede administrar separadamente. Esto significa dos cosas:

- Las operaciones se realizarán sobre un segmento de tamaño considerablemente inferior al de la tabla, por lo que los tiempos de indisponibilidad serán significativamente menores.
- En caso de pérdida de una partición, si se encuentra en un tablespace separado del resto de particiones, las labores de restauración y recuperación afectarán sólo a esta partición, quedando el resto de la tabla accesible.

Igualmente se mejorará la disponibilidad ya que en caso de que una partición no esté disponible, el optimizador automáticamente borrará particiones que no sean referenciadas en el plan de acceso y así consultas no serán afectadas cuando haya particiones no disponibles.

Se han creado nuevas opciones de mantenimiento en herramientas como SQL*Loader, Datapump, etc. que permiten estas operaciones de mantenimiento sobre particiones individuales sin alterar la disponibilidad de resto de la tabla.

Un beneficio adicional es que al ocuparnos de segmentos de tamaño reducido, minimizamos los tiempos de recuperación y los recursos utilizados para ello.

Oracle dispone de gran variedad de métodos y comandos para el manejo de las particiones. Por ejemplo:

- Una partición podrá ser movida de un tablespace a otro.
- Una partición puede ser dividida en varias particiones
- Una partición puede ser borrada, añadida o truncada.
- Aunque las operaciones `SELECT`,`UPDATE`, `INSERT` y `DELETE` no necesitan cambiarse por el hecho de tener una tabla particionada, se pueden restringir a nivel de partición en lugar de que apliquen a la tabla completa, añadiendo la cláusula `PARTITION` en el `FROM`.

Las operaciones de mantenimiento que se pueden realizar sobre las tablas particionadas son:

- Add table partition
- Modify partition
- Move table partition
- Rename partition
- Drop partition
- Truncate partition
- Split partition

Por ejemplo el `DROP` y el `TRUNCATE` de una partición evitarán la ejecución de muchas sentencias DML `DELETE`.

Estratégias de Particionamiento de Tablas.

El particionamiento en Oracle ofrece varias estrategias para indicar el método en que la base de datos pone en el lugar adecuado los datos en particiones. Dependiendo del método de distribución de datos Oracle ofrece tres modelos fundamentales de particionamiento:

- Particionamiento por rangos (y su extensión por intervalos desde 11g)
- Particionamiento hash (desde Oracle8i)
- Particionamiento por listas (desde Oracle9i)

Además se podrá elegir entre particionamiento de un solo nivel o particionamiento compuesto. Desde 11g se añaden varias extensiones de particionamiento que aumentarán las opciones de elegir una estrategia de particionamiento. También en versión 12c y 12cR2 se añaden nuevas mejoras y opciones de particionamiento que serán indicadas en los siguientes apartados.

Cada estrategia tiene diferentes ventajas y consideraciones de diseño. Así cada estrategia será más adecuada a una situación particular.

Particionamiento de un sólo nivel (Single-Level Partitioning)

En este caso la tabla se crea especificando sólo uno de los siguientes métodos de distribución usando una o más columnas como clave de particionamiento:

- Particionamiento por rangos
- Particionamiento hash
- Particionamiento por listas

Particionamiento por rangos

Este es el primer tipo de particionamiento disponible desde Oracle 8. Es el que más se suele aplicar, fundamentalmente sobre campos de tipo fecha.

Este tipo de particionamiento distribuye los datos entre las particiones en base a rangos continuos de valores de las claves de particionamiento. Los rangos de los valores de las particiones se definen mediante comparaciones de tipo "menos estricto que".

Se ha de seguir las siguientes reglas:

- Cada partición tiene una cláusula `VALUES LESS THAN`, que específica límite superior para la partición. Cualquier valor de la clave de partición igual o mayor a este valor límite se incluye en la siguiente partición.
- Todas las particiones, salvo la primera, tienen un límite inferior implícito que es el indicado en la cláusula `VALUES LESS THAN` de la partición que le precede.
- Se puede definir el valor `MAXVALUE` como límite superior de la última partición. `MAXVALUE` representa un valor que es superior al más alto que pueda tomar la clave de partición, incluyendo el valor `NULL`. (se puede asimilar al concepto de valor infinito en matemáticas.)

Ejemplo simple de una tabla particionada por rango:

```
CREATE TABLE time_range_sales
   ( prod_id        NUMBER(6)
   , cust_id        NUMBER
   , time_id        DATE
   , channel_id     CHAR(1)
   , promo_id       NUMBER(6)
   , quantity_sold  NUMBER(3)
   , amount_sold    NUMBER(10,2)
   )
PARTITION BY RANGE (time_id)
 (PARTITION SALES_1998 VALUES LESS THAN (TO_DATE('01-JAN-
1999','DD-MON-YYYY')),
  PARTITION SALES_1999 VALUES LESS THAN (TO_DATE('01-JAN-
2000','DD-MON-YYYY')),
  PARTITION SALES_2000 VALUES LESS THAN (TO_DATE('01-JAN-
2001','DD-MON-YYYY')),
  PARTITION SALES_2001 VALUES LESS THAN (MAXVALUE)
 );
```

Particionamiento por listas

Este tipo de particionamiento está disponible desde Oracle 9i.

El particionamiento por lista permite controlar de forma explícita como se asignan los registros a las particiones. Esto se consigue especificando una lista de valores discretos de la clave de particionamiento.

La ventaja de este método de particionamiento es que permite agrupar y organizar de una forma natural conjuntos de datos desordenados (no existe un criterio de ordenación natural) y no relacionados.

La partición `DEFAULT` permite evitar tener que especificar todos los posibles valores en la definición de una tabla particionada por listas, ya que usando la partición default todas las filas que no satisfagan los valores de la lista irán a esta partición sin que se genere error.

Ejemplo simple de una tabla particionada por lista:

```
CREATE TABLE list_sales
   ( prod_id        NUMBER(6)
   , cust_id        NUMBER
   , time_id        DATE
   , channel_id     CHAR(1)
   , promo_id       NUMBER(6)
   , quantity_sold  NUMBER(3)
   , amount_sold    NUMBER(10,2)
   )
PARTITION BY LIST (channel_id)
 (PARTITION even_channels VALUES (2,4),
  PARTITION odd_channels VALUES (3,9)
 );
```

Particionamiento hash

Este tipo de particionamiento está disponible desde Oracle 8i.

El particionamiento hash es muy sencillo de implementar ya que su sintaxis es muy elemental. No asigna rangos a las particiones sino que utiliza un algoritmo hash sobre la clave de particionamiento para distribuir equitativamente los registros entre las particiones que se hayan definido. Este particionamiento generará particiones de aproximadamente el mismo tamaño.

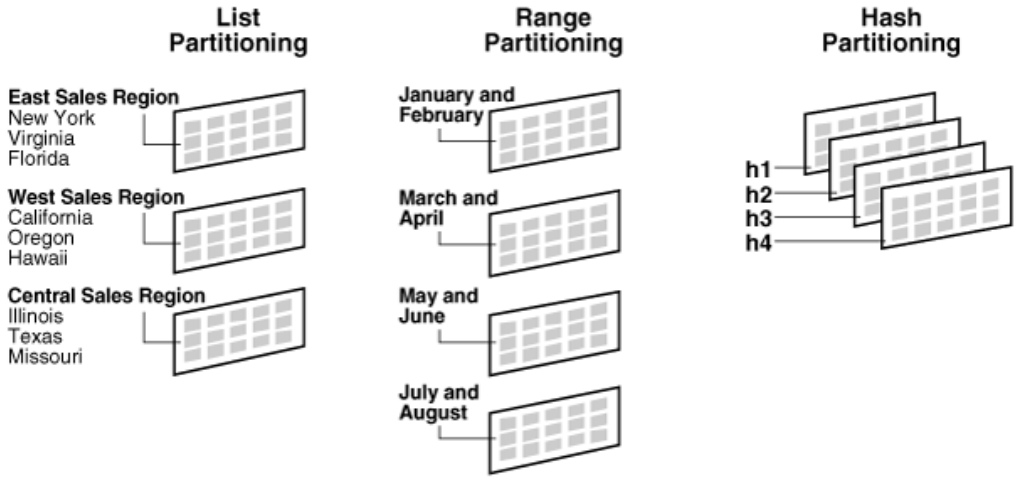
No se podrá manipular los algoritmos hash usados por el particionamiento.

El particionamiento tipo Hash es ideal para distribuir datos entre dispositivos físicos. También es usado como alternativa al particionamiento por rango, especialmente cuando los datos a particionar no son históricos o cuando no se detecta una clave de particionamiento clara.

Ejemplo simple de una tabla particionada por hash con 2 particiones:

```
CREATE TABLE hash_sales
   ( prod_id        NUMBER(6)
   , cust_id        NUMBER
   , time_id        DATE
   , channel_id     CHAR(1)
   , promo_id       NUMBER(6)
   , quantity_sold  NUMBER(3)
   , amount_sold    NUMBER(10,2)
   )
PARTITION BY HASH (prod_id)
PARTITIONS 4;
```

La siguiente gráfica muestra una visión física de las estrategias de particionamiento de un solo nivel:

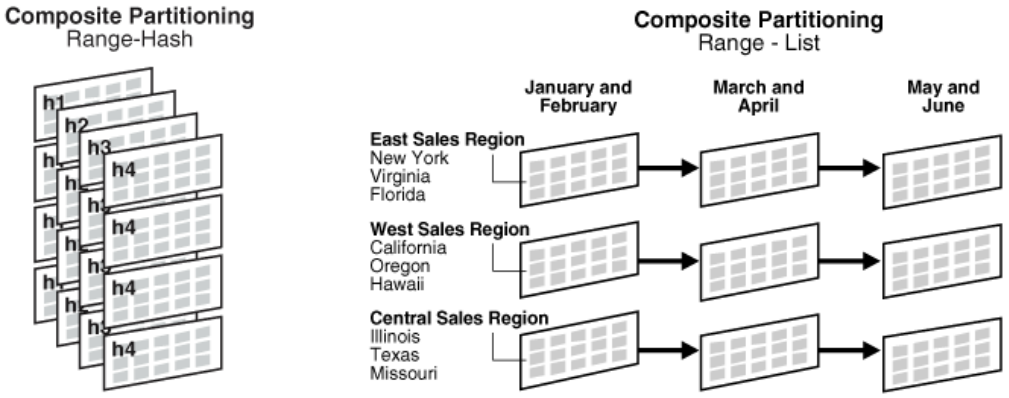


Particionamiento compuesto

El particionamiento compuesto es una combinación de los métodos de distribución fundamentales. Se generará una tabla particionada por un método de distribución que a su vez cada partición será subdividida en subparticiones usando un segundo método de distribución.

Con el particionamiento compuesto se podrá tener un grado más fino de granularidad en la distribución de los datos a través de las subparticiones.

En la siguiente gráfica se muestra una tabla particionada compuesta tipo Rango-Hash y otra tipo Rango-Lista, en este caso la tabla será particionada por rangos en la que cada partición, a su vez, estará particionada en subparticiones por hash o por una lista.



Los tipos de particionamiento compuesto son:

- Particionamiento compuesto rango-rango (Desde 11g)

  Particionamiento compuesto rango-rango permite el particionamiento lógico de rangos en dos dimensiones, por ejemplo, particionar por la columna order_date y subparticionar por la columna order_value

- Particionamiento compuesto rango-hash (Desde 8i)

En el particionamiento compuesto rango-hash se particionará usando el método de rango, y dentro de cada partición, las subparticiones usarán el método hash.

Este particionamiento compuesto rango-hash proporciona la manejabilidad del particionamiento por rango y las mejoras en distribución de los datos y paralelismo del particionamiento hash.

- Particionamiento compuesto rango-lista (Desde 9i)

  En el particionamiento compuesto rango-lista se particionará usando el método de rango, y dentro de cada partición, las subparticiones usarán el método lista.

  El particionamiento compuesto rango-lista proporciona la manejabilidad del particionamiento por rango y el control explícito del particionamiento por lista para las subparticiones.

- Particionamiento compuesto lista-rango (Desde 11g)

  El particionamiento compuesto lista-rango permite subparticionar por rango dentro de una estrategia de particionamiento por lista, por ejemplo, particionar por lista por country_id y subparticionar por rango por order_date.

- Particionamiento compuesto lista-hash (Desde 11g)

  El particionamiento compuesto lista-hash permite subparticionar por hash dentro de una estrategia de particionamiento por lista.

- Particionamiento compuesto lista-lista (Desde 11g)

  Particionamiento compuesto lista-lista permite el particionamiento lógico de listas en dos dimensiones, por ejemplo, particionar por country_id y subparticionar por sales-channel.

- Particionamiento compuesto hash-hash (Desde 11gR2)

  Particionamiento compuesto hash-hash permite el particionamiento hash en dos dimensiones. Esto puede ser útil para favorecer el partition-wise joins en dos dimensiones.

- Particionamiento compuesto hash-lista (Desde 12c)

  El particionamiento compuesto hash-lista permite subparticionar por lista dentro de una estrategia de particionamiento por hash.

- Particionamiento compuesto hash-rango (Desde 12c)

  El particionamiento compuesto hash-rango permite subparticionar por rango dentro de una estrategia de particionamiento por hash.

  También las extensiones por intervalo que se verán más adelante podrán utilizarse en particionamiento compuesto.

Extensiones de Particionamiento (A partir de 11g)

Además de las estrategias básicas de particionamiento, en versión 11g Oracle añade las siguientes extensiones de Particionamiento que se agrupan por:

- Manageability Extensions. Mejoran el manejo de tablas particionadas:

  - Particionamiento por Intervalos
  - Partition Advisor

- Partitioning Key Extensions. Añaden más opciones a la hora de elegir la clave de particionamiento:
  - Reference Partitioning
  - Particionamiento sobre columnas virtuales

- System Partitioning

Particionamiento por Intervalos

El particionamiento por intervalos es una extensión del particionamiento por rangos que consiste en indicar a la base de datos automáticamente que cree particiones de un específico intervalo cuando se inserten datos que exceden las particiones por rango existentes, por ejemplo particiones mensuales, que automáticamente se crean el primer día del mes.

Se deberá especificar al menos una partición tipo rango. El valor de la clave de particionamiento determinará el valor mayor del rango de particiones, que se llamará punto de transición, y la base de datos creará particiones de intervalos para datos cuyos valores son superiores al punto de transición.

Cuando se usan particiones tipo intervalo existirán restricciones:

- Sólo se puede especificar una clave de particionamiento y esta deberá ser del tipo `NUMBER, FLOAT, DATE` o `TIMESTAMP`.
- El particionamiento por intervalo no está soportado para IOTs.
- No se podrá crear un índice domain en una tabla particionada por intervalos.
- No está soportado el particionamiento por intervalos a nivel de subpartición.

Se pueden crear tablas particionadas por intervalo single-level y las siguientes tablas particionadas compuestas:

Interval-rango

Interval-hash

Interval-lista

Partition Advisor

El Partition Advisor es una funcionalidad incluida en el SQL Access Advisor. El Partition Advisor puede recomendar de forma autónoma una estrategia de particionamiento para una tabla, basándose en el estudio de un conjunto de SQL indicadas y que pueden venir de la SQL Area, un SQL Tuning Set o definidas por el usuario.

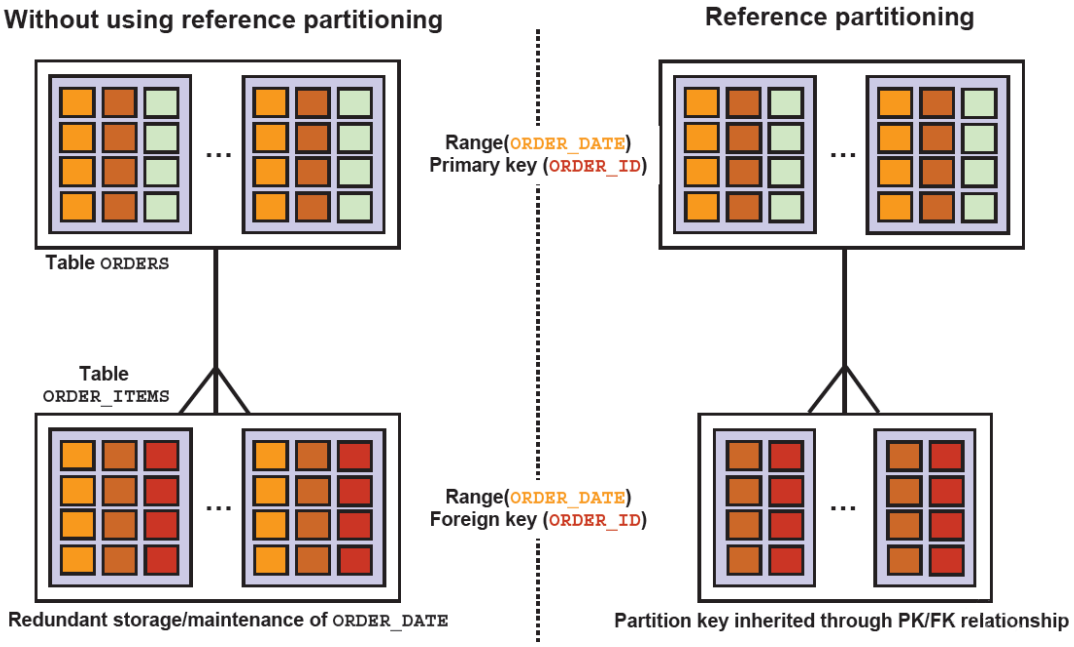
Reference Partitioning

Reference partitioning permite que una tabla sea particionada basándose en el método de particionado de la tabla referenciada por una "constraint". La clave de

particionamiento se resuelve a través de la relación padre-hijo creada a través de las constraints primary key y foreign key.

El gran beneficio de esta extensión es que las tablas con una relación padre-hijo puede estar lógicamente equiparticionadas (esto es, que tienen las mismas particiones) a través de heredar la clave de particionamiento desde la tabla padre sin tener que duplicar columnas.

Un ejemplo de Reference Partitioning se muestra en la gráfica siguiente



En la parte de la derecha se muestra la situación en la que se tienen 2 tablas `ORDERS` y `ORDERS_ITEMS` que están equiparticionadas por la columna `ORDER_DATE`. En este caso, ambas tablas necesitan definir la columna `ORDER_DATE`. Sin embargo, definir `ORDER_DATE` en la tabla `ORDER_ITEMS` es redundante porque hay una relación primary key/foreign key entre las dos tablas. En la parte de la derecha se muestra la situación cuando se usa reference partitioning. En este caso no hace falta definir la columna `ORDER_DATE` en la tabla `ORDER_ITEMS`. La clave de particionamiento en la tabla `ORDER_ITEMS` se hereda automáticamente gracias a la relación existente de primary key /foreigh key.

Desde versión 12c también es posible usar particionamiento por intervalos en las tablas padre para reference partitioning. Las particiones por intervalo serán creadas también automáticamente en la tabla hija cuando se inserten filas referenciadas que tienen particiones en la tabla padre.

Particionamiento sobre columnas virtuales

Desde versión 11g es posible definir columnas virtuales derivadas de aplicar una función o expresión a otras columnas de la propia tabla. Estas columnas podrán ser utilizadas como clave de particionamiento, esto será el particionamiento sobre columnas virtuales.

Por ejemplo si una tabla tiene una columna `ACCOUNT_ID` que consiste en 10 dígitos de los cuales los 3 primeros indican el número de oficina. La tabla se podría extender

con una columna virtual ACCOUNT_BRANCH, que venga derivada de sacar los 3 primeros dígitos de la columna ACCOUNT_ID y usando el Particionamiento sobre columnas virtuales, usar esta columna ACCOUNT_BRANCH como clave de particionamiento de la tabla.

Está soportado el uso de todas las estrategias de particionamiento de un solo nivel y compuesto en la definición del particionamiento sobre columnas virtuales.

Ejemplo:

```
create table t (c1 int,
                c2 varchar2(10),
                c3 date,
                c3_v char(1)
                generated always as
                (to_char(c3,'d')) virtual
                )
 partition by list (c3_v)
  (
       partition p1 values ('1'),
       partition p2 values ('2'),
       partition p3 values ('3'),
       partition p4 values ('4'),
       partition p5 values ('5'),
       partition p6 values ('6'),
       partition p7 values ('7')
       );
```

System Partitioning

Permite controlar el particionamiento desde el nivel de aplicación sin que la base de datos controle el lugar donde se localizan los datos. La base de datos simplemente proporciona la habilidad de dividir la tabla en particiones pero sin conocer para que van a ser usadas estas.

Todos los aspectos de particionamiento son controlados desde la aplicación. Por ejemplo si se intenta insertar en una tabla system partitioned sin indicar explícitamente la especificación de partición fallará.

System partitioning proporciona los beneficios del particionamiento (escalabilidad, disponibilidad, manejabilidad), pero el particionamiento y la localización de los datos serán controlados por la aplicación.

No existirá clave de particionamiento.

Ejemplo de creación de una System Partition:

```
CREATE TABLE syspart_example (c1 number,  c2 varchar2(10), c3
date)
PARTITION BY SYSTEM
( PARTITION p1 ,
  PARTITION p2 ,
  PARTITION p3 );
```

Si realizáramos una inserción convencional sobre la tabla se producirá un error del tipo:

```
SQL> insert into syspart_example values (1,'A',sysdate);
           *
ERROR en línea 1:
ORA-14701: Se debe utilizar el nombre de partición ampliada o la
variable ligada para DML en tablas particionadas
```

Para realizar la inserción deberá por lo tanto utilizarse, una sentencia del tipo siguiente:

```
SQL> insert into syspart_example partition (p3) values
(1,'A',sysdate);
insert into syspart_example values (1,'A',sysdate);
```

Las sentencias `delete` y `update` no requieren la sintaxis de partition.

Este particionado puede ser útil en aplicaciones donde es posible gestionar la forma en la que se realiza el particionado (lógica de aplicación).

Para más información sobre este tipo de particionamiento ver la nota de MOS: 452447.1 11g Partitioning Enhancements.

Nuevas opciones de particionamiento a partir de 12c

Además de las estrategias básicas de particionamiento, en versión 12c Oracle añade las siguientes nuevas opciones en particionamiento:

- Particionamiento en External Tables (Desde 12.2.0.1)

  Esta funcionalidad permite mejorar el rendimiento con el uso de partition pruning y partition wise join en sentencias sql sobre tablas externas. Adicionalmente permite que se capturen estadísticas a nivel de partición e incrementales en tablas externas lo que permitirá al optimizador optar por mejores planes de ejecución.

- Range Partitioning for Hash Clusters (Desde 12.1.0.2)

  El particionamiento de hash clusters está soportado por rango y sólo para particionamiento single-level.

- Oracle XML DB and Domain Index Support of Hash Partitioned Tables (Desde 12.1.0.2)

  Oracle XML DB y otras aplicaciones que usan Domain indexes pueden usar el particionamiento por hash a partir de 12.1.0.2.

  En versión 12c para XMLIndex está soportado el uso de particionamiento hash, list, y range. En versión 12.2.0.1 también interval y reference partitioning están soportados para columnas y tablas XMLType y XMLIndex.

- Interval-Reference Partitioning (Desde 12.1.0.1)

  Desde versión 12c también es posible usar particionamiento por intervalos en las tablas padre junto con la estrategia de reference partitioning. Las particiones por intervalo serán creadas también automáticamente en la tabla hija cuando se inserten filas referenciadas que tienen particiones en la tabla padre.

- Multi-column List Partitioning (Desde 12.2.0.1)

  Desde versión 12cR2 es posible particionar una tabla basándose en una lista de valores para múltiples columnas. Es similar al particionamiento por lista de una sola columna pero en este caso cada partición contendrá como lista un conjunto de valores.

- Automatic List Partitioning (Desde 12.2.0.1)

  Desde versión 12cR2 es posible el particionamiento por lista de forma automática. Una tabla particionada auto-list es similar a una tabla particionada por lista, excepto que es más fácil de manejar. Se puede crear una tabla particionada auto-list usando sólo los valores conocidos en ese momento de la clave de particionada. Cuando se vayan cargando datos a la tabla, la base de datos automáticamente irá creando nuevas particiones si la clave de particionamiento insertada no está en la lista definida de particiones existentes. Conceptualmente es similar al método de particionamiento por intervalos ya que las particiones son automáticamente creadas.

Indices Particionados. Tipos.

Un índice particionado es un índice que, como una tabla particionada, se descompone en trozos más pequeños y manejables. Al igual que con tablas particionadas, los índices particionados mejorarán la gestión, la disponibilidad, el rendimiento y la escalabilidad en nuestra base de datos.

Habrá diferentes tipos de índices con el uso de particionamiento:
- Índice noparticionado en una tabla particionada, también llamado índice global noparticionado.
- Índice particionado, que se divide en:
  - Índice global. Estará particionado independientemente de la tabla donde esté creado.
  - Índice local. Estará particionado pero las particiones estarán enlazadas con el método de particionado de una tabla.

Además los índices particionados a su vez se categorizarán en:
- Prefixed
- Nonprefixed.

Se pueden crear índices tipo bitmap en tablas particionadas. La única restricción es que solo se permiten índices bitmap de tipo local. No pueden ser índices globales. Los índices bitmap globales sólo se soportan en tablas no particionadas.

En versión 12c se añade la funcionalidad de "Partial Indexes" que permite que los índices locales o globales puedan ser creados en un subconjunto de particiones de la tabla, lo que proporciona más flexibilidad en la creación del índice.

Índices particionados locales

En un índice particionado local, el índice estará particionado por las mismas columnas, con el mismo número de particiones y con los mismos límites de particionamiento que la tabla.

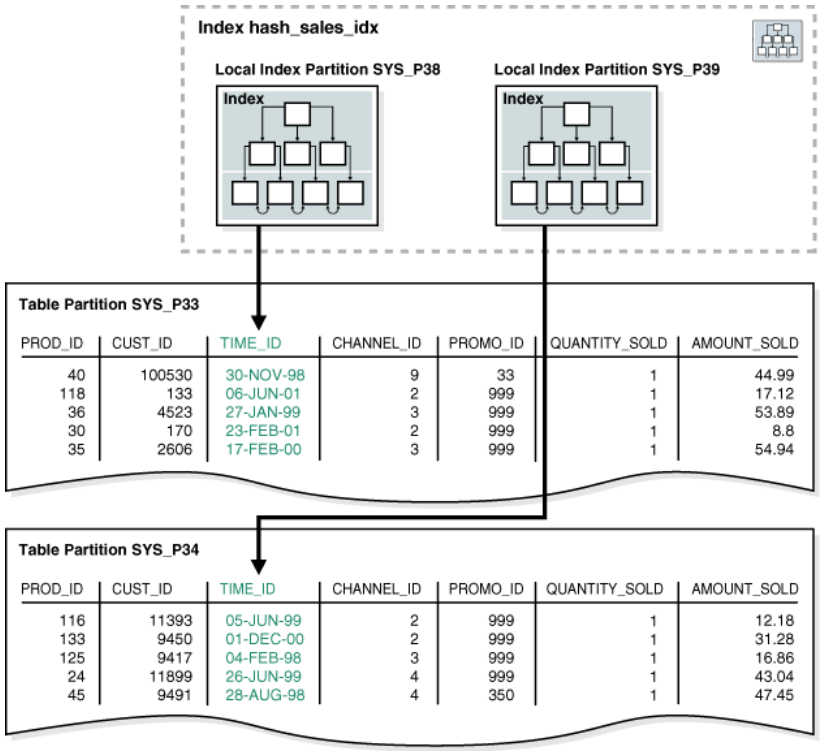
Cada partición del índice se asociará exactamente a una partición de la tabla donde se cree, y todas las claves en una partición del índice apuntarán a sólo filas de una única partición de la tabla. De esta forma, la base de datos sincronizará automáticamente las particiones del índice con su partición en la tabla.
No se podrá explícitamente añadir ni borrar una partición en un índice local, esto se hará automáticamente cuando se añada o se borre en la tabla particionada.

Los índices particionados locales son comunes en entornos Data Warehouse. La disponibilidad es mayor ya que en caso de que no haya datos disponibles en una partición sólo afectará a esa partición.

Imaginemos el siguiente ejemplo de una tabla `hash_sales` particionada en 2 particiones por hash en la columna `product_id`. Podremos crear un índice local en la columna `time_id` con en el siguiente comando:

```
CREATE INDEX hash_sales_idx ON hash_sales(time_id) LOCAL;
```

Gráficamente podremos ver que el índice se divide en dos particiones cada una enlazada con una partición en la tabla. La partición del índice `SYS_P38` indexará las filas de la partición de la tabla `SYS_P33`, y la partición del índice `SYS_P39` indexará las filas de la partición `SYS_P34`:



Los índices particionados locales se dividen a su vez en:

- Local prefixed indexes

En este caso, la clave de particionamiento aparece en la parte inicial de definición del índice. En el ejemplo anterior, un índice local prefixed sería si tiene en la primera columna del índice la columna `product_id`.

- Local nonprefixed indexes

  En este caso, la clave de particionamiento no aparece en la parte inicial de definición del índice y no tiene incluso que estar en la definición del índice. En el ejemplo anterior es un caso de índice local nonprefixed.

Ambos tipos tienen la ventaja de poder utilizar la eliminación de particiones (partition pruning), que ocurre cuando el optimizador mejora el rendimiento en el acceso a datos excluyendo particiones que no son necesarias. Esta eliminación de particiones dependerá del predicado de la sentencia sql. Un sentencia sql que usa un índice local prefixed siempre permitirá la eliminación de particiones en cambio un local noprefixed no siempre.

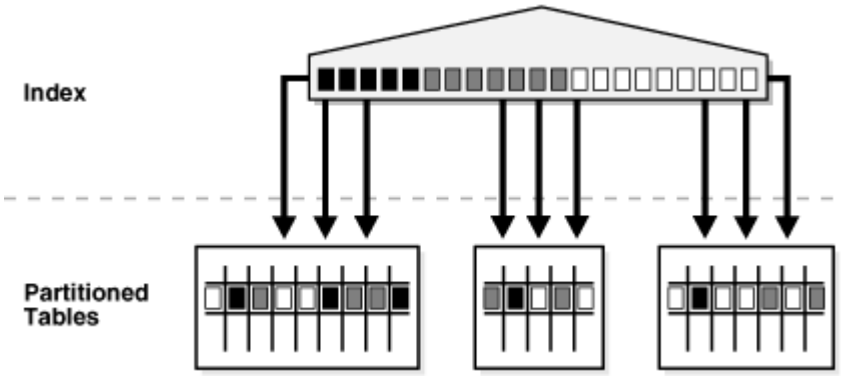
Índices particionados y no particionados globales

Un índice particionado global es un índice B-tree que está particionado independientemente del método de particionamiento de la tabla donde se cree y de su clave de particionamiento. Una única partición del índice podría apuntar a cualquier o todas las particiones de la tabla.

Se puede particionar un índice global por rango y por hash. Por hash mejorará el rendimiento bajando la contención cuando un índice crece de forma continua por un lado, esto es, las inserciones en el índice ocurren solo en el lado derecho del índice.

Un índice noparticionado global es un índice normal creado en una tabla particionada y tiene las mismas características.

En la siguiente imagen muestra una visión gráfica de un índice global noparticionado:



En general, los índices globales se usan en aplicaciones OLTP, donde el acceso rápido, la integridad de los datos y la disponibilidad son importantes. En un sistema OLTP, una tabla podría estar particionados por una clave, por ejemplo, la columna `employees.department_id`, pero una aplicación podría necesitar acceder a los

datos por muchas otras diferentes claves, por ejemplo `employee_id` o `job_id`. Índices globales podrían ser útiles en este escenario.

Algunas restricciones con el uso de particionamiento

Hay una serie de restricciones que se tiene que tener en cuenta a la hora de plantearse el particionamiento de una tabla:

- Tipos de datos. No se puede particionar una tabla que contenga alguna columna de tipo `LONG o LONG RAW`. Desde Oracle8i, se admite la presencia de campos de tipo `LOB (BLOB, CLOB, NCLOB y BFILE).`
- Aunque las tablas con `LOBs` pueden particionarse, las columnas con tipo `LOBs` no pueden usarse en la clave de particionamiento. Los segmentos de los `LOBs` estarán equiparticionados con la tabla, esto es, habrá tantas particiones de los `LOBs` como particiones en la tabla, además las características de los `LOBS` (storage in row, tablespace, etc) se podrán especificar a nivel de cada partición.
- No se puede particionar una tabla que forman parte de un cluster.
- Índices Bitmap. Los índices bitmap tienen como única restricción el que tienen que particionarse de forma local.
- Optimizador de costes. El optimizador de reglas no soporta particionamiento. Una aplicación que utilice reglas no se beneficiará de la eliminación de particiones, aunque las ventajas de mantenimiento seguirán estando presentes. En cualquier caso el optimizador basado en reglas no está soportado desde versión 10g.
- Restricciones físicas. Una tabla particionada no puede dispersarse en varias bases de datos. Todas sus particiones deben estar en la misma base de datos.
- Todas las particiones deberán residir en tablespaces con el mismo tamaño de bloque. Los índices u otros segmentos como segmentos `LOBs` podrán residir en tablespaces con otro tamaño de bloque.

Implementando Particionamiento.

Estrategia óptima de particionamiento

Hay dos principales criterios a considerar cuando se elige una estrategia de particionamiento, el rendimiento y una más facil administración o gestión. La estrategía de particionamiento debería seguir siempre criterios de rendimiento. Esto significa que se buscará que las sentencias, reportes, etc sean optimizados para mejorar rendimiento y escabilidad, a menudo a costa de la facilidad de uso o gestión.

Como ejemplo, los datos de la tabla ventas son a menudo consultados por la aplicación por la localización geográfica y después por fecha (por ejemplo las ventas de mi región el último mes), mientras que los procesos ETL cargarán los datos basándose en la fecha (por ejemplo, las ventas de la semana son cargadas en un único batch para todas las regiones geográficas). El conflicto en este ejemplo es que los datos podrían particionarse por región geográfica o por fecha. En un caso se favorecerá el acceso del usuario, mientras que en otro se optimizarán las cargas ETL. Dependiendo de la prioridad del negocio se primará una u otra, pero lo más habitual y recomendable es primar el usuario final de la aplicación.

En un RAC otro factor que puede ayudar a decidir un criterio puede ser reducir la contención del tráfico entre nodos. Para esto se puede particionar la carga de trabajo en la misma linea que el particionamiento de los datos. De esta forma se reduciría la probabilidad de que grandes cantidades de datos viajen continuamente entre nodos. En el ejemplo anterior, este acercamiento se haría haciendo que los reports de una región geográfica se procese en nodos específicos, mientras que otras regiones en otros nodos, así los datos de ciertas particiones serán con más probabilidad accedidos exclusivamente por ciertos nodos, reduciendo en gran medida el tráfico interconnect.

Tablas/Índices candidatas a ser particionadas

Algunas sugerencias para determinar cuando una tabla debería particionarse:

- Oracle recomienda que tablas superiores a 2GB deberían particionarse. Incluso otras tablas de menor tamaño también podrían beneficiarse de su particionamiento.
- Tablas grandes que crecen rápidamente.
- Tablas resumen o agregados tipo DW (summarized tables, aggregate tables, summarized historical tables)
- Cuando el contenido de la tabla se quiera distribuir entre diferentes tipos de almacenamiento (por ejemplo discos rápidos para datos más accedidos)

Algunas sugerencias para determinar cuando un índice debería particionarse:

- Cuando se quiera evitar la reconstrucción de un índice entero si se borran datos.
- Realizar tareas de mantenimiento de partes de los datos sin tener que invalidar el índice completamente.

- Cuando se quiera bajar contención en un índice en una columna que monótonamente incrementa el valor y que crece de forma continua por un lado.

Escenarios de Particionamiento

Cuando se defina la lógica en el diseño del particionamiento, dependiendo de la naturaleza del dato habrá diferentes escenarios relativos a cómo las tablas podrían ser particionadas. Los escenarios más comunes son:

- Particionamiento por periodo de tiempo:
  Particionando por periodo de tiempo es muy común en entorno Data Warehouse. En este escenario cada partición contendrá las transacciones de un periodo de tiempo concreto (meses, años, etc).

- Particionamiento por división lógica:
  Este escenario es más común en entornos de bases de datos OLTP o transaccionales. En este escenario, una tabla como aquella que guarde registros de empleados es dividida entre particiones por regiones geográficas o divisiones de la compañia. Tareas administativas en unas regiones podrían realizarse sin afectar a otras.

- Particionamiento por la columna consultada:
  Debido a las mejoras de rendimiento al introducir el particionamiento, un criterio de particionamiento podria ser por una columna que es muy accedida. Por ejemplo, en la tabla CUSTOMER la mayoría de las consultas y updates buscan por la primary key CustomerID. Particionando usando como clave este CustomerID generalmente distribuirá las sentencias entre todas las particiones.

- Particionamiento por el tipo de acceso a la tabla (read-only/read-write):
  Si existe una porción importante de la tabla que es de solo lectura, se podría particionar los datos por el tipo de acceso. Así solo las particiones read-write necesitarán backup&recovery y la complejidad de la administración se reduce mientras que la disponibilidad se incrementa. Esta es menos común como estrategia lógica de particionamiento, pero es legítima bajo las circunstancias apropiadas.

Particionamiento de Tablas. Estrategias y usos. Ejemplos.

Habiendo considerado una visión global conceptual de lo que son las tablas particionadas, en la siguiente sección se revisará la sintaxis requerida para la creación de tablas particionadas así como las estrategias y los métodos de partición más habituales recomendados para su implementación, así como indicar las situaciones en los que estos métodos serán más o menos adecuados.

Comando CREATE TABLE con particionamiento

Basta añadir la clausula de particionamiento a un script de creación de una tabla para implementar particionamiento. Las clausulas de almacenamiento TABLESPACE, STORAGE (INITIAL, NEXT), PCTFREE son iguales que en una tabla normal.

```
CREATE TABLE (… columns …)
      PARTITION BY RANGE ( column_list)
      PARTITION BY HASH ( column_list)
      PARTITION BY LIST ( column)
      PARTITION BY SYSTEM
      PARTITION BY RANGE ( column)
            INTERVAL (NUMTOYMINTERVAL(1,'MONTH')
  (PARTITION specifications) ;
```

La declaración de creación de la tabla particionada se compone de tres elementos:

- La estructura lógica de la tabla
- La estructura de particionamiento definiendo el tipo y columnas
- La estructura de cada partición o subpartición que se compondrá de 2 partes:
    - Límites lógicos
    - Atributos físicos de almacenamiento

Ejemplo:

```
CREATE TABLE example
             (idx NUMBER, txt VARCHAR2(20))
 PARTITION BY RANGE (idx)
      (PARTITION VALUES LESS THAN (0)
            TABLESPACE DATA01
      , PARTITION VALUES LESS THAT (MAXVALUE));
```

En el ejemplo, se muestra una tabla particionada por rango cuyas filas que tenga un valor negativo en la columna idx serán guardadas en la primera partición que físicamente estará en el tablespace DATA01.

Atributos lógicos y físicos de una tabla particionada:

Cuando se especifica una tabla o índice particionado, se podrán especificar varios atributos. Los atributos se dividen fácilmente entre aquellos que declaran algo lógico sobre la tabla o partición y aquellos que indican algo sobre la forma de almacenamiento físico de las particiones.

Generalmente los atributos lógicos son comunes para toda la tabla, mientras que los atributos físicos aplicarán a cada partición. Los atributos físicos declarados para la tabla son usados como los valores por defecto para cada partición si no se indica otra cosa.

Atributos lógicos:

- Estructura normal de la tabla (columnas, constraints)
- Partition type
- Clave de particionamiento y valores
- Row movement

Atributos físicos:

- Tablespace
- Tamaños de extensión y bloques

Notas generales sobre tablas particionadas:

- Una tabla particionada también puede crearse con la cláusula AS SELECT.
- Podrán crearse hasta un total de 1024K-1 = 1048575 particiones y subparticiones.
- Se podrá crear una tabla particionada con una única partición. Una tabla con una única partición será diferente a una no particionada. Por ejemplo, no se le pueden añadir particiones a una tabla no particionada.
- Si no se especifica el nombre de la partición explícitamente, serán nombradas con los nombres SYS_Pnnnn.

Creando una tabla particionada por Rango

Como vimos anteriormente, este tipo de particionamiento distribuye los datos entre las particiones en base a rangos continuos de valores de las claves de particionamiento. Es el que más se suele aplicar, fundamentalmente sobre campos de tipo fecha.

Se indicarán rangos de valores para la columna o columnas que formen la clave de particionamiento para determinar en qué partición las filas deberán ser guardadas.

Este tipo de particionamiento es el más adecuado cuando hay datos que tienen un rango lógico por el que se pueden distribuir. El rendimiento es mejor cuando los datos se distribuyen de forma equitativa a lo largo del rango. Si el particionamiento por rango causa que haya grandes variaciones de tamaño en las particiones porque hay una distribución muy desigual, se debería considerar otros métodos de particionamiento.

Por ejemplo, en una tabla EMPLEADOS, un particionamiento por rango en la columna EmpID podría ser una configuración aceptable ya que los ID de empleados se asignan secuencialmente y uno podría concluir que a lo largo del tiempo se podría tener una distribución bastante equilibrada de filas entre particiones. Por otro

lado, podría ocurrir que con el tiempo aquellos empleados que primero fueron contratados y con un menor EmpID serán los primeros en dejar la compañía. Si este fuera el caso, las primeras particiones tendrían menos filas que las últimas, produciendo un peor particionamiento.

El escenario ideal para el particionamiento por rango es datos históricos los cuales se distribuyen de forma uniforme a lo largo del tiempo. Por ejemplo, en la tabla VENTAS, si uno particiona por mes o trimestre las ventas, y los volúmenes de ventas son relativamente consistentes entre meses o trimestres, entonces la columna FechaVenta podría ser un buen ejemplo de columna para la clave de particionamiento por rango.

Particionando por rango basado en una columna fecha tendremos varios beneficios:

- Se podrán identificar claramente el número total de particiones que necesitaremos crear en un periodo de tiempo concreto. Por ejemplo, si los datos son particionados trimestralmente y queremos tener un periodo de retención de 5 años, podemos determinar que necesitaremos 20 particiones.
- Cuando un trimestre expira, la partición más antigua podría ser borrada con una simple operación.
- Las consultas contra datos históricos normalmente trabajan con rangos de fechas. Cualquier partición que no entre en el criterio de búsqueda será internamente eliminada (partition pruning)
- Ya que las particiones estarían basadas en la edad del dato, ciertos atributos de los segmentos donde se guarden podrían ser personalizados. Por ejemplo, si los datos más antiguos no se modifican, estos podrían localizarse en tablespace de READ-ONLY, o añadir clausulas COMPRESS para particiones antiguas.

Nota: Si el rango es una columna tipo caracteres (char, varchar2, etc), las comparaciones serán usando los códigos ASCII en el primer byte. Esto es, si se usa VALUES LESS THAN 'A', cualquier palabra que empieza por A será localiza en la siguiente partición.

Cuando usar Particionamiento por Rango:

- Particionamiento por Rango (o intervalo), es ideal cuando se hacen búsquedas de datos en los que se referencian un rango de valores de la clave de particionamiento, como pueden ser operaciones BETWEEN o LIKE, o expresiones regulares. Por ejemplo tablas muy grandes que son frecuentemente escaneadas con un predicado por rangos en columnas como ORDER_DATE o PURCHASE_DATE. En estos casos el partition pruning mejorará el rendimiento tiempo de respuesta de las consultas.
- Se quiere mantener una ventana de borrado de datos.
- No se pueden completar tareas administrativas, como por ejemplo un backup y restore, en tablas muy grandes en una franja determinada de tiempo, pero sí se puede dividir en trozos lógicos más pequeños basados en el particionamiento por rango de una columna.

Ejemplo 1:

Este primer ejemplo se creará la tabla EMPLEADOS particionada por rango por la clave primaria. Se usará una secuencia que para cargar los valores de la columna EmpID que será la clave primaria.

```
CREATE SEQUENCE EmpIDSeq;
```

```
CREATE TABLE empleados (
       EmpID  NUMBER(8) PRIMARY KEY,
       Nombre VARCHAR2(20),
       Apellido1 VARCHAR2(20),
       Apellido2 VARCHAR2(20),
       Salario      NUMBER(6),
       Deptno NUMBER(2))
PARTITION BY RANGE (EmpID)
  (PARTITION p_empleado1 VALUES LESS THAN (250)
   TABLESPACE DATOS1,
   PARTITION p_empleado2 VALUES LESS THAN (500)
   TABLESPACE DATOS2,
   PARTITION p_empleado3 VALUES LESS THAN (750)
   TABLESPACE DATOS3,
   PARTITION p_empleado4 VALUES LESS THAN (1000)
   TABLESPACE DATOS4);
```

Insertamos un conjunto de datos de ejemplo usando la secuencia EmpIDSeq.

```
begin
       for i in 1..900 loop
              insert into empleados (    EmpID,Nombre,   Apellido1,
Apellido2,    Salario, Deptno)
                     values
(EmpIDSeq.NEXTVAL,'JUAN','PEREZ','PEREZ',1000+i,1);
       end loop;
       commit;
end;
/
```

Confirmamos el particionamiento

Como se comentó anteriormente, el usuario se referirá a la tabla completa como si fuera un único objeto al igual que si la tabla fuera no particionada. El usuario no necesita indicar en qué partición se encuentra una fila, internamente Oracle resolverá esto. Aún así, es posible incluir en cualquier sentencia el nombre de una partición si queremos restringir a esta, añadiendo la cláusula PARTITION tras el nombre de la tabla.

Por ejemplo, en nuestro caso, revisamos como se han distribuidos las filas por partición:

```
SELECT COUNT(*), MAX(EmpID)
FROM empleados PARTITION (p_empleado1);

  COUNT(*) MAX(EMPID)
---------- ----------
       249        249

SELECT COUNT(*), MAX(EmpID)
FROM empleados PARTITION (p_empleado2);

  COUNT(*) MAX(EMPID)
---------- ----------
       250        499

SELECT COUNT(*), MAX(EmpID)
FROM empleados PARTITION (p_empleado3);

  COUNT(*) MAX(EMPID)
---------- ----------
       250        749

SELECT COUNT(*), MAX(EmpID)
```

```
FROM empleados PARTITION (p_empleado4);

  COUNT(*) MAX(EMPID)
---------- ----------
       151        900
```

Notas sobre los rangos usados:

- Cada partición tiene una cláusula VALUES LESS THAN, que específica límite superior para la partición. Cualquier valor de la clave de partición igual o mayor a este valor límite se incluye en la siguiente partición. Todas las particiones, salvo la primera, tienen un límite inferior implícito que es el indicado en la cláusula VALUES LESS THAN de la partición que le precede.
- Si se realiza un INSERT que incluye una valor de la clave de particionamiento que está por encima del límite superior de la última partición se generará el error:

  ORA-14400: inserted partition key does not map to any partition
- Para evitar este error, se puede usar el valor MAXVALUE como límite superior de la última partición. MAXVALUE representa un valor que es superior al más alto que pueda tomar la clave de partición, incluyendo el valor NULL (se puede asimilar al concepto de valor infinito en matemáticas.)
- En este caso, al crear la primary key en la definición de la tabla, el índice que crea, será un índice global noparticionado. Ya que la definición de la primary key es sobre la columna que es la clave de particionamiento, se podría haber creado previamente el índice como local y usar este como índice de la primary key, o en la definición de la creación de la tabla especificar las características del índice que se quiere para la primary key.

Ejemplo 2:

Consideremos otro ejemplo más práctico de particionamiento por rango en la tabla VENTAS

```
CREATE TABLE VENTAS (
       CustomerID    NUMBER(4),
       ProductID     NUMBER(4),
       SaleDate DATE,
       Qty                  NUMBER(4),
       UnitPrice     NUMBER(6,2),
       PRIMARY KEY (CustomerID, ProductID, SaleDate))
       PARTITION BY RANGE (SaleDate)
       (PARTITION sales1 VALUES LESS THAN
       (TO_DATE('01/01/2011','DD/MM/YYYY'))
              TABLESPACE DATOS1,
       PARTITION sales2 VALUES LESS THAN
       (TO_DATE('01/01/2012','DD/MM/YYYY'))
              TABLESPACE DATOS2,
       PARTITION sales3 VALUES LESS THAN
       (TO_DATE('01/01/2013','DD/MM/YYYY'))
       TABLESPACE DATOS3,
       PARTITION sales4 VALUES LESS THAN (MAXVALUE)
       TABLESPACE DATOS4);
```

Varias notas sobre esta tabla:

- El índice en la clave primara será creado como un índice global no particionado. Al tener dentro de la definición del índice la clave de particionamiento (`SaleDate`) se permitiría crear el índice previamente definido como LOCAL, y posteriormente asignarle a la tabla la primary key, o en la definición de la creación de la tabla especificar las características del índice que se quiere para la primary key. Si la primary key fuese sólo en (`CustomerID, ProductID`) el índice sólo puede ser creado global (particionado prefixed o no-particionado).
- Cuando la clave de partición es tipo DATE, las expresiones en VALUES pueden incluir la función TO_DATE.
- La última partición asegura que todas las filas podrán ser insertadas. También las filas con valor NULL en la clave de particionamiento se guardarán en esta partición.

Examinando los objetos físicos particionados

Podemos confirmar la estructura del objeto particionado examinando varias vistas de diccionario:

USER_PART_TABLES - Esta vista mostrará como la tabla está particionada, cuantas particiones existen y cuáles son las columnas que componen la clave de particionamiento.

USER_PART_KEY_COLUMNS - Esta vista mostrará detalles sobre la clave de particionamiento.

USER_TAB_PARTITIONS- Esta vista muestra las particiones individualmente y los detalles de los rangos usados en las particiones

```
SELECT table_name, partitioning_type "Type", partition_count "Count",
partitioning_key_count  "Key  count"  from  user_part_tables  WHERE
table_name='EMPLEADOS';

TABLE_NAME                     Type           Count  Key count
------------------------------ --------- ---------- ----------
EMPLEADOS                      RANGE              4          1

SELECT * FROM user_part_key_columns WHERE name='EMPLEADOS';
NAME                           OBJEC COLUMN_NAM COLUMN_POSITION
------------------------------ ----- ---------- ---------------
EMPLEADOS                      TABLE EMPID                    1


SELECT table_name, partition_name, high_value, partition_position
                   FROM user_tab_partitions
                  WHERE table_name='EMPLEADOS'
ORDER BY 2;

TABLE_NAME    PARTITION_NAME    HIGH_VALUE  PARTITION_POSITION
------------- ----------------- ----------- ------------------
EMPLEADOS     P_EMPLEADO1       250                          1
EMPLEADOS     P_EMPLEADO2       500                          2
EMPLEADOS     P_EMPLEADO3       750                          3
EMPLEADOS     P_EMPLEADO4       1000                         4
```

Sobre la clave de particionamiento. Multicolumnas y Operaciones UPDATE.

Algunas notas relativas a la clave de particionamiento y los rangos indicados en la especificación de la partición:

- La clave de particionamiento puede ser una única columna o una lista de hasta 16 columnas.
- La columna o columnas que forman la clave de particionamiento no podrán ser cambiadas sin recrear y volver a cargar la tabla por completo, que tratándose de tablas muy grandes puede llevarse bastante tiempo. Por tanto, habrá que elegir con cuidado la columna o columnas que formen la clave de particionamiento.
- El valor de la clave de partición debe ser expresado como un literal o como una expresión usando la función TO_DATE en columnas fechas.

  Incluso expresiones simples como 3+5, TO_NUMER('65') o ASCII('G') no son permitidas. La única excepción es usar TO_DATE con el propósito de especificar el formato NLS usado. Si no se especifica el CREATE TABLE cogerá el NLS del entorno de la sesión que lanza el comando.

Multicolumnas en la clave de particionamiento:

Cuando se eligen varias columnas para una clave de particionamiento, el orden es significativo. La segunda columna será examinada sólo después de que los valores de la primera columna sean iguales a la especificación del límite.

Esto es, cuando se comparan los valores de una fila con los puntos finales de la partición, para determinar en qué partición la fila debería localizarse Oracle seguirá estos pasos:

- Si la primera columna "es menor que" el primer valor clave de la partición, entonces la fila pertenece a esa partición. Esto significa que la segunda columna podría contener NULLs. El valor de la clave de particionamiento para la segunda y siguientes columnas son simplemente ignorados.
- Si la primera columna es igual al primer valor clave de la partición entonces la segunda columna es comparada con el valor de la segunda partición:
  - Si la segunda columna "es menor que" al valor clave de la primera partición, entonces la fila pertenece a esa partición.
  - Si la segunda columna es "mayor o igual" al valor clave de la primera partición, la tercera columna será examinada. Si no hay tercera columna, la fila pertenecerá a la siguiente partición.
  - Si no hay siguiente partición la fila será rechazada.

Ejemplo de multicolumna:

Supongamos que esta es la definición de la partición:

```
create table multicol
       (unit NUMBER (1), subunit CHAR(1) )
       PARTITION BY RANGE ( unit, subunit)
       (      PARTITION P_2b VALUES LESS THAN (2,'B')
       ,      PARTITION P_2c VALUES LESS THAN (2,'C')
       ,      PARTITION P_3b VALUES LESS THAN (3,'B')
       ,      PARTITION P_4x VALUES LESS THAN (4,'X') );
```

Imaginamos que insertamos los siguientes valores, se indica en qué partición se guardarán:

Datos	Partición:
1,'A'	P_2B
2,'A'	P_2B
2,'D'	P_3B
4,'A'	P_4X
1,'Z'	P_2B
2,'B'	P_2C
2,NULL	P_3B
4,'Z'	ORA-14400: inserted partition key does not map any partition
1,NULL	P_2B
2,'C'	P_3B
3,'Z'	P_4X
4,NULL	ORA-14400: inserted partition key does not map any partition

Podemos usar la siguiente consulta para determinar dónde va cada fila:

```
select d.subobject_name as "Partition", m.unit, m.subunit from
multicol m, dba_objects d where dbms_rowid.rowid_object (m.rowid)
= d.object_id;

Partition           UNIT S
------------ ---------- -
P_4X                 3 Z
P_4X                 4 A
P_3B                 2 C
P_3B                 2
P_3B                 2 D
P_2C                 2 B
P_2B                 1
P_2B                 1 Z
P_2B                 2 A
P_2B                 1 A
```

Varios casos de multicolumna que habría que resaltar:

- Fechas con otros tipos de datos:

  Hay que tener cuidado cuando se definen fechas usando otros tipos de datos y particiones en múltiples columnas. Ejemplo

```
CREATE TABLE… ( year NUMBER(4), month NUMBER(2), day NUMBER
(2) …)
 PARTITION BY RANGE (year, month, day)
 (PARTITION VALUES LESS THAN (2011,01,32)
 ,PARTITION VALUES LESS THAN (2011,02,29)
    ,PARTITION VALUES LESS THAN (2011,03,32)
```

Se supone que sólo se esperarán que se inserten fechas válidas, pero una fila con valores (2010,13,88) es válida y se insertará en la partición inicial. Por tanto habría que añadir un constraint CHECK que evitara esa posibilidad.

El valor para día tiene que ser un día superior que el último día del mes, ya que los valores no son inclusivos. Si se define la clave de partición como (day, month, year) con esos valores las particiones serían totalmente distintas y no tendrían el mismo sentido ya que las filas no estarían particionadas por meses.

Lo mejor será particionar directamente por una columna tipo DATE.

- Se recomienda en la definición de la clave de particionamiento se use el TO_DATE, donde se especifique claramente el formato para evitar ambigüedades.
- Uso de palabras como columna inicial de la clave de particionamiento

  El uso palabras como columna inicial de la clave de particionamiento puede dar resultados no esperados. Por ejemplo supongamos que tenemos la tabla DBA_SOURCE particionada. Hay pocos procedures DBMS que tienen miles de líneas de código y la mayoría tienen sólo pocos cientos de filas. Un particionamiento inicial podría pensarse así:

```
PARTITION BY (NAME, LINE)
( PARTITION DBA_SOURCE_P1
              VALUES LESS THAN ('DBMS',1000)
, PARTITION DBA_SOURCE_P2
              VALUES LESS THAN ('DBMS', MAXVALUE)
,PARTITION DBA_SOURCE_P3
              VALUES LESS THAN (MAXVALUE, MAXVALUE)
```

  El resultado de este particionamiento que nos esperamos es que aquellos procedures que tienen más de 1000 líneas estén en la partición DBA_SOURCE_P2. La realidad será que la partición DBA_SOURCE_P2 no tiene ni una sola fila, todo el código DBMS* estará guardado en la partición DBA_SOURCE_P3. El problema en la comparación es que una palabra DBMRxxxxx será menor que DBMS para la primera partición, y DBMS_xxxx será mayor que DBMS para la partición DBA_SOURCE_P1 y DBA_SOURCE_P2. Así sólo un procedure que se llame exactamente DBMS que tenga 1000 o más líneas será localizado en la partición DBA_SOURCE_P2. En este ejemplo para conseguir el resultado esperado la alternativa será a utilizar particionamiento compuesto RANGE-RANGE.

Operaciones UPDATE y la clave de particionamiento

Como es conocido, si la clave primaria de una tabla es modificada, no solo los cambios se producirán en el valor de la columna si no que también requiere el cambio correspondiente en el índice de la clave primaria. Cuando los cambios ocurren en la clave de particionamiento, el impacto puede ser incluso mayor si el nuevo valor hace que la fila tenga que moverse a otra partición. Por defecto, las tablas son creadas con la opción DISABLE ROW MOVEMENT. Esto significa que cuando el valor de la clave de partición cambie de forma que la fila tiene que moverse de partición, se produzca el siguiente error:

ORA-14402: update partition key column would cause a partition change

Este mensaje puede evitarse haciendo un alter table y especificando la clausula ENABLE ROW MOVEMENT. De esta forma la fila será migrada a otra partición.

Esto puede ser una operación costosa, especialmente si ocurre frecuentemente, ya que generará una considerable actividad de redo. Por tanto, se debería **evitar elegir claves de particionamiento de columnas que sufren a menudo modificaciones**. Esto aplicará normalmente en bases de datos transaccional tipo OLTP, menos en bases de datos tipo Data Warehouse donde la información generalmente es más estática.

Aunque la tabla esté creada con el DISABLE ROW MOVEMENT, si el update no requiere que la fila se tenga que mover de partición, este será válido.

Creando una tabla particionada por Intervalos

Desde versión 11g se introduce la posibilidad del particionamiento por intervalos que es una extensión de particionamiento por Rangos en el que se le indica a la base de datos que automáticamente cree particiones de un intervalo específico cuando se inserten datos en la tabla que exceden los rangos de las particiones existentes. Se deberá especificar al menos un rango de partición. El valor del rango de la clave de particionamiento determina el valor más alto de los rangos de las particiones, el cual es llamado el punto de transición. La base de datos creará particiones de intervalo para los datos que sobrepasen el punto de transición.

El uso de particionamiento por intervalos puede venir bien para automatizar la creación de manual de particiones en el caso de que estas sean muy habituales, por ejemplo particiones diarias.

Cuando se usan particiones tipo intervalo existirán restricciones:

- Sólo se puede especificar una clave de particionamiento y esta deberá ser del tipo NUMBER o DATE (TIMESTAMP y TIMESTAMP WITH LOCAL TIME ZONE también).
- El particionamiento por intervalo no está soportado para IOTs.
- No se podrá crear un índice domain en una tabla particionada por intervalos.
- No está soportado el particionamiento por intervalos a nivel de subpartición.

Ejemplo:

```
CREATE TABLE VENTAS_INTERVAL (
       CustomerID    NUMBER(4),
       ProductID     NUMBER(4),
       SaleDate DATE,
       Qty                  NUMBER(4),
       UnitPrice     NUMBER(6,2),
       PRIMARY KEY (CustomerID, ProductID, SaleDate))
       PARTITION BY RANGE (SaleDate)
       INTERVAL    (NUMTOYMINTERVAL(1,    'month'))    store    in
(DATOS1,DATOS2,DATOS3,DATOS4)
(
PARTITION          P1          VALUES          LESS          THAN
(TO_DATE('01/01/2013','DD/MM/YYYY')) TABLESPACE DATOS1);
```

En este ejemplo, a partir del punto de transición 01/01/2013 las particiones se crearán un rango de un mes. Estas particiones se crearán cuando se inserten valores superiores a 01/01/2013. Por ejemplo, la partición Pi1 se creará automáticamente cuando se inserte un valor correspondiente a Enero 2013. Con la opción STORE IN en la clausula INTERVAL se podrá especificar que las particiones se vayan localizando en distintos tablespaces mediante round-robin.

Otro ejemplo:

```
CREATE TABLE interval_sales
    ( prod_id NUMBER(6)
    , cust_id NUMBER
    , time_id DATE
    , channel_id CHAR(1)
    , promo_id NUMBER(6)
    , quantity_sold NUMBER(3)
    , amount_sold NUMBER(10,2) )
PARTITION BY RANGE (time_id)
    INTERVAL(numtodsinterval(7,'day'))
    ( PARTITION p0 VALUES LESS THAN (TO_DATE('1-1-2013', 'DD-
MM-YYYY')));
```

En este caso el punto de transición será el 1 de enero de 2013, y las particiones se crearán con un intervalo de una semana.

Se puede cambiar una tabla particionada por rango para que sea particionada por intervalo usando el ALTER TABLE y definiendo el valor del intervalo:

ALTER TABLE table_name SET INTERVAL (interval_value);

Igualmente con ALTER TABLE table_name SET INTERVAL (); se deshabilita la creación automática de particiones y convierto todas las particiones en particiones por rango. Cuando se cambia el intervalo de una tabla particionada por intervalo, las particiones por intervalo existentes no cambiarán, sólo afectará a las nuevas.

Creando una tabla particionada por Hash

El método de particionamiento por lista es similar al de Rango con la excepción de que cada partición incluye una lista de valores en lugar de un límite del rango.

```
…
PARTITION BY HASH (PartitionKey)
PARTITIONS x
STORE IN ( TABLESPACE1, TABLESPACE2, … TABLESPACEN)
```

En este caso x indicará el número de particiones hash que se irán distribuyendo por los tablespaces indicados alternativamente. Se podrán indicar los nombres de las particiones, en este caso la cláusula PARTITIONS será ignorada.

Recomendaciones cómo y cuándo usar particionamiento por Hash:

- Se recomienda que el número de particiones sea un potencia de 2, esto es, 2, 4, 8, 16, 32 y así. Esto es debido al algoritmo hash de particionamiento usado internamente que no es modificable, si no es potencia de 2, las primeras particiones contendrán desproporcionalmente más filas.
- La cardinalidad de las columnas es muy importante cuando se elija una clave para particionamiento por hash. La función HASH funciona mejor con un número alto de valores distintos. Oracle recomienda evitar crear particionamiento por hash en columnas que tienen baja cardinalidad (columnas en las que el número de valores distintos es pequeño comparado con el número total de filas). Lo más recomendable es elegir una columna o combinación de columnas en la clave de particionamiento que sean únicas o casi únicas.

- El particionamiento tipo Hash es usado como alternativa al particionamiento por rango especialmente cuando no hay datos que se pueden distribuir por rango de valores.
- El particionamiento hash es más adecuado si las consultas contra los datos se usan condiciones de igualdad u operadores tipo IN.
- Podrá ser usado para distribuir aleatoriamente los datos entre dispositivos físicos aunque no se tendrá control sobre qué partición ocupará una fila concreta. Esto podrá ser usado para evitar cuellos de botella I/O en caso de que no se utilice una técnica de almacenamiento que haga stripe y mirror entre todos los dispositivos disponibles.

Ejemplo:

```
CREATE TABLE products
   ( prod_id   NUMBER(6)
   , name        VARCHAR2(20)
   , comments  VARCHAR2(20)
   , ListPrice   NUMBER(6,2)
   , Location  VARCHAR2(10)
   , MemberID  NUMBER(4)
   )
PARTITION BY HASH (prod_id)
PARTITIONS 4
STORE IN (DATOS1,DATOS2,DATOS3,DATOS4);


SELECT table_name, partition_name, high_value, partition_position
      FROM user_tab_partitions
     WHERE table_name='PRODUCTS'
ORDER BY 2;


TABLE_NAME     PARTITION_NAME    HIGH_VALUE  PARTITION_POSITION
-------------- ----------------- ----------- ------------------
PRODUCTS       SYS_P161                                      1
PRODUCTS       SYS_P162                                      2
PRODUCTS       SYS_P163                                      3
PRODUCTS       SYS_P164                                      4
```

En este ejemplo, los nombres de las particiones son generados por el sistema. Podríamos crear la tabla indicando el nombre de cada partición y su localización:

```
CREATE TABLE products
   ( prod_id   NUMBER(6)
   , name        VARCHAR2(20)
   , comments  VARCHAR2(20)
   , ListPrice   NUMBER(6,2)
   , Location  VARCHAR2(10)
   , MemberID  NUMBER(4)
   )
PARTITION BY HASH (prod_id)
(PARTITION products1 TABLESPACE DATOS1,
PARTITION products2 TABLESPACE DATOS2,
PARTITION products3 TABLESPACE DATOS3,
PARTITION products4 TABLESPACE DATOS4);
```

Creando una tabla particionada por Lista

El particionamiento por lista permite controlar de forma explícita como se asignan los registros a las particiones. Esto se consigue especificando una lista de valores discretos de la clave de particionamiento.

La ventaja de este método de particionamiento es que permite agrupar y organizar de una forma natural conjuntos de datos desordenados (no existe un criterio de ordenación natural) y no relacionados. Por ejemplo en la tabla CLIENTES se podrían agrupar por distintas provincias, de forma que podamos distribuir los clientes en particiones que tengan aproximadamente el mismo número de clientes.

Cuando usar el Particionamiento por Lista:

- El particionamiento por lista, como el particionamiento por rango, es ideal cuando se hacen búsquedas de datos en los que se referencian un rango de valores de la clave de particionamiento, como pueden ser operaciones BETWEEN o LIKE, o expresiones regulares.

- Su uso permitirá mapear filas a particiones basándose en valores discretos. Por ejemplo, en la tabla CLIENTES particionada por varias provincias, los managers que analicen las cuentas por región tendrán la ventaja del partition pruning.

Ejemplo1:

```
CREATE TABLE products
   ( prod_id   NUMBER(6)
   , name        VARCHAR2(20)
   , comments  VARCHAR2(20)
   , ListPrice   NUMBER(6,2)
   , Location  VARCHAR2(20)
   , MemberID  NUMBER(4)
   )
PARTITION BY LIST  (Location)
      (PARTITION east VALUES ('BOSTON','NEW YORK','MIAMI')
       TABLESPACE DATOS1,
      PARTITION mid VALUES ('CHICAGO','SEATTLE')
       TABLESPACE DATOS2,
      PARTITION west VALUES ('LOS ANGELES','SAN FRANCISCO')
       TABLESPACE DATOS3,
      PARTITION p_others VALUES (DEFAULT));
```

La partición DEFAULT permite evitar tener que especificar todos los posibles valores en la definición de una tabla particionada por listas, ya que usando la partición default todas las filas que no satisfagan los valores de la lista irán a esta partición sin que se genere error. En este caso esta partición no especifica el tablespace y por tanto se creará en el tablespace por defecto del usuario.

Cambiando la lista de valores de la partición

Ya que este método de particionamiento se refiere a valores discretos, los cambios en la localización de una compañía podrán tener un impacto en la especificación de la

partición. Se pueden añadir y borrar valores a la lista de una partición. Para añadir se usará MODIFY PARTITION ... ADD VALUES.

Por ejemplo,

```
ALTER TABLE products
      MODIFY PARTITION mid
      ADD VALUES ('DENVER');
```

Hay varias restricciones:
- No se puede añadir un valor que ya está incluido en otra partición
- Si hay alguna fila que contiene el valor en la partición por defecto, se generará un error.
- No se pueden añadir valores a la partición por defecto.

Para borrar valore se usará MODIFY PARTITION ... DROP VALUES

```
ALTER TABLE products
      MODIFY PARTITION mid
      DROP VALUES ('DENVER');
```

Igualmente, si hay filas que contienen el valor que se está borrando, el ALTER TABLE fallará. En ese caso habrá que previamente hacer DELETE o UPDATE de las filas con problemas antes de modificar la especificación de la partición.

Multi-column List-Partitioned Table

Desde versión 12cR2 es posible particionar una tabla basándose en una lista de valores para múltiples columnas.

Es similar al particionamiento por lista de una sola columna, cada partición contendrá un conjunto de una lista de valores.

Ejemplo2:

```
CREATE TABLE sales_by_region_and_channel
 (deptno           NUMBER,
  deptname         VARCHAR2(20),
  quarterly_sales  NUMBER(10,2),
  state            VARCHAR2(2),
  channel          VARCHAR2(1)
 )
 PARTITION BY LIST (state, channel)
 (
  PARTITION     q1_northwest_direct     VALUES     (('OR','D'),
('WA','D')),
  PARTITION    q1_northwest_indirect    VALUES     (('OR','I'),
('WA','I')),
  PARTITION             q1_southwest_direct              VALUES
(('AZ','D'),('UT','D'),('NM','D')),
  PARTITION q1_ca_direct VALUES ('CA','D'),
  PARTITION rest VALUES (DEFAULT)
 );
```

En este ejemplo la table estará particionada usando multi-column partitioning en las columnas state y channel.

Sólo se permite tener una partición DEFAULT para toda la tabla.

Creando una tabla particionada por Lista automática

Desde versión 12cR2 es posible el particionamiento por lista de forma automática (Automatic List-Partitioned).

Una tabla particionada auto-list es similar a una tabla particionada por lista, excepto que es más fácil de manejar. Se puede crear una tabla particionada auto-list usando sólo los valores conocidos de la clave de particionada.

Cuando se vayan cargando datos a la tabla, la base de datos automáticamente irá creando nuevas particiones si la clave de particionamiento insertada no está en la lista definida de particiones existentes. Conceptualmente es similar al método de particionamiento por intervalos ya que las particiones son automáticamente creadas.

En el comando de CREATE o ALTER TABLE SQL se añade una cláusula para especificar particionamiento por lista AUTOMATIC o MANUAL. En caso de automatic, no se podrá tener una default partition y requerirá al menos una partición.

Cuando usar el Particionamiento por Lista automática:

Su uso es similar a usar una tabla particionada por lista manual. Pero este tipo de particionamiento no es adecuado en tipos de datos que varían frecuentemente, a menos que se puedan homogenizar los datos ya que el volumen de creación de particiones serían muy grande.

Ejemplo1:

```
CREATE TABLE sales_auto_list
(
   salesman_id   NUMBER(5),
   salesman_name VARCHAR2(30),
   sales_state   VARCHAR2(20),
   sales_amount  NUMBER(10),
   sales_date    DATE
)
  PARTITION BY LIST (sales_state) AUTOMATIC
 (PARTITION P_CAL VALUES ('CALIFORNIA')
);
```

Tablas particionadas compuestas

El particionamiento compuesto es una combinación de los métodos de distribución fundamentales en dos dimensiones. Desde el punto de vista de rendimiento se podrá obtener la ventaja del partition pruning en una o dos dimensiones dependiendo de la sentencia SQL, y también se puede tener el beneficio del uso de full o partial partition-wise joins en ambas dimensiones.

Se generará una tabla particionada por un método de distribución elegido y cada partición generada a su vez será subdividida en subparticiones usando un segundo método de distribución. Cada subpartición podrá tener propiedades que difieran de las propiedades de la tabla o de la partición a la que la subpartición pertenezca.

Con el particionamiento compuesto se podrá tener un grado más fino de granularidad en la distribución de los datos a través de las subparticiones.

Las combinaciones son las siguientes:

- Range-hash. (Desde 8i)
- Range-list. (Desde 9i)
- Range-range. (Desde 11g)
- List-range. (Desde 11g)
- List-list. (Desde 11g)
- List-hash. (Desde 11g)
- Hash-hash (introducido en la versión 11gR2).
- Hash-lista (Desde 12c)
- Hash-rango (Desde 12c)

Como extensión a Range también se podrá realizar particionamiento compuesto de

- Interval-Range (Desde 11g)
- Interval-list. (Desde 11g)
- Interval-hash. (Desde 11g)

Particionado Range-Hash (O Interval-Hash)

En un particionamiento Range-hash las particiones de datos por el método rango, y dentro de cada partición, esta se subparticiona usando el método hash.

Este tipo de particionamiento Range-hash son comunes para tablas que guardan datos históricos, y que pueden ser muy grandes, y que son frecuentemente unidas con otras tablas muy grandes. Para este tipo de tablas (típicas en sistemas Data warehouse), el particionamiento compuesto Range-hash permite el beneficio del partition pruning a nivel de rango con la oportunidad de realizar parallel full o partial partion-wise joins a nivel hash.

El particionamiento range-hash también puede ser usando para tablas que tradicionalmente usan particionado hash, pero que también se quieren adaptar a una política de ventana de borrado. Cada cierto tiempo los datos pueden ser movidos desde un almacenamiento a otro, guardados en read-only tablespaces, o incluso borrarlos.

Veamos un ejemplo utilizando el tipo RANGE y el HASH. En primer lugar, hace un particionado del tipo RANGE utilizando rangos de años. En segundo lugar, para cada partición definida por cada año, hacemos un segundo particionado (subpartición) del tipo aleatorio (HASH) por el valor de otra columna:

```
CREATE TABLE TAB2 (ord_id     NUMBER(10),
     ord_day    NUMBER(2),
```

```
   ord_month  NUMBER(2),
   ord_year   NUMBER(4)
   )
PARTITION BY RANGE(ord_year)
SUBPARTITION BY HASH(ord_id)
 ( PARTITION q1 VALUES LESS THAN(2001)
   ( SUBPARTITION q1_h1 TABLESPACE TBS1,
     SUBPARTITION q1_h2 TABLESPACE TBS2,
     SUBPARTITION q1_h3 TABLESPACE TBS3,
     SUBPARTITION q1_h4 TABLESPACE TBS4
   ),
   PARTITION q2 VALUES LESS THAN(2002)
   ( SUBPARTITION q2_h5 TABLESPACE TBS5,
     SUBPARTITION q2_h6 TABLESPACE TBS6,
     SUBPARTITION q2_h7 TABLESPACE TBS7,
     SUBPARTITION q2_h8 TABLESPACE TBS8
   ),
   PARTITION q3 VALUES LESS THAN(2003)
   ( SUBPARTITION q3_h1 TABLESPACE TBS1,
     SUBPARTITION q3_h2 TABLESPACE TBS2,
     SUBPARTITION q3_h3 TABLESPACE TBS3,
     SUBPARTITION q3_h4 TABLESPACE TBS4
   ),
   PARTITION q4 VALUES LESS THAN(2004)
   ( SUBPARTITION q4_h5 TABLESPACE TBS5,
     SUBPARTITION q4_h6 TABLESPACE TBS6,
     SUBPARTITION q4_h7 TABLESPACE TBS7,
     SUBPARTITION q4_h8 TABLESPACE TBS8
   )
 );
```

```
select table_name, partition_name, subpartition_name, tablespace_name
from dba_tab_subpartitions
where table_name='TAB2';
```

TABLE_NAME	PARTITION_NAME	SUBPARTITION_NAME	TABLESPACE_NAME
TAB2	Q1	Q1_H1	DATOS1
TAB2	Q1	Q1_H2	DATOS2
TAB2	Q1	Q1_H3	DATOS3
TAB2	Q1	Q1_H4	DATOS4
TAB2	Q2	Q2_H5	DATOS5
TAB2	Q2	Q2_H6	DATOS6
TAB2	Q2	Q2_H7	DATOS7
TAB2	Q2	Q2_H8	DATOS8
TAB2	Q3	Q3_H1	DATOS1
TAB2	Q3	Q3_H2	DATOS2
TAB2	Q3	Q3_H3	DATOS3
TAB2	Q3	Q3_H4	DATOS4
TAB2	Q4	Q4_H5	DATOS5
TAB2	Q4	Q4_H6	DATOS6
TAB2	Q4	Q4_H7	DATOS7
TAB2	Q4	Q4_H8	DATOS8

Otro ejemplo de una tabla particionada por range-hash puede ser la tabla page_history de un proveedor de servicios de Internet. La definición de la tabla está optimizada para análisis históricos para específicos valores de client_ip (en los que las sentencias se beneficiarán del partition pruning) o para análisis a través de muchas direcciones IPs, en cuyo caso las sentencias podrán coger beneficio de full o partial partition-wise joins.

```
CREATE TABLE page_history
( id                NUMBER NOT NULL
, url               VARCHAR2(300) NOT NULL
, view_date         DATE NOT NULL
, client_ip         VARCHAR2(23) NOT NULL
, from_url          VARCHAR2(300)
, to_url            VARCHAR2(300)
, timing_in_seconds NUMBER
) PARTITION BY RANGE(view_date) INTERVAL
(NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH(client_ip)
SUBPARTITIONS 32
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2006','dd-MON-
yyyy')))
PARALLEL 32 COMPRESS;
```

En este ejemplo se usa el particionamiento por intervalo, que es un como un particionamiento por rango en el que las particiones son creadas automáticamente cuando los datos son insertados, en este caso cada día se creará una partición, y cada partición estará subparticionada en 32 subparticiones hash.

Particionado Range-List:

Igual que el método de particionamiento compuesto range-hash, el método de particionamiento Range-List permite el particionamiento basado en una jerarquía de dos niveles.
El primer nivel de particionamiento basado en un rango de valores, y el segundo nivel basado en valores discretos. Esta forma de particionamiento compuesto viene bien para datos históricos que permite además agrupar las filas de datos basándose en valores de columna que no están ordenados o no tienen relación.

Por ejemplo, el siguiente comando creará una tabla particionada Range-list llamada call_detail_records. Una compañía de telecomunicaciones puede usar esta tabla para analizar los tipos de llamadas a los largo del tiempo. La tabla también usará índices locales en from_number y to_number.

```
CREATE TABLE call_detail_records
( id NUMBER
, from_number        VARCHAR2(20)
, to_number          VARCHAR2(20)
, date_of_call       DATE
, distance           VARCHAR2(1)
, call_duration_in_s NUMBER(4)
) PARTITION BY RANGE(date_of_call)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY LIST(distance)
SUBPARTITION TEMPLATE
( SUBPARTITION local VALUES('L') TABLESPACE tbs1
, SUBPARTITION medium_long VALUES ('M') TABLESPACE tbs2
, SUBPARTITION long_distance VALUES ('D') TABLESPACE tbs3
, SUBPARTITION international VALUES ('I') TABLESPACE tbs4
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2005','dd-MON-
yyyy')))
PARALLEL;
```

```
CREATE INDEX from_number_ix ON
call_detail_records(from_number)
LOCAL PARALLEL NOLOGGING;

CREATE INDEX to_number_ix ON call_detail_records(to_number)
LOCAL PARALLEL NOLOGGING;
```

En este ejemplo se usa el particionamiento por intervalo, que es un como un particionamiento por rango en el que las particiones son creadas automáticamente cuando los datos son insertados. Estos índices locales estarán equiparticionados con la tabla base en el siguiente sentido:

- Contendrán tantas particiones como la tabla base
- Cada partición del índice constará de tantas subparticiones como la correspondiente partición de la tabla base. Podremos verlas en la vista dba_ind_subpartitions
- Las entradas del índice para las filas de una subpartición concreta de la tabla base estarán guardadas en la correspondiente subpartición del índice.

Particionado Range-Range:

Particionamiento Range-Range permite habilitar el particionamiento por rango en 2 dimensiones. Son útiles en aplicaciones que guardan datos basados en fecha en varias dimensiones. Por ejemplo, particionar la tabla order por rango en order_date, y subparticionar por rango por shipping_date.

Un ejemplo de una tabla particionada Range-Range puede ser la tabla `account_balance_history`. En la que un banco puede querer usar el acceso a subparticiones individuales para contactar con los clientes con un bajo balance en la cuenta o especificar promociones para ciertas categorías de clientes.

```
CREATE TABLE account_balance_history
( id                 NUMBER NOT NULL
, account_number     NUMBER NOT NULL
, customer_id        NUMBER NOT NULL
, transaction_date   DATE NOT NULL
, amount_credited    NUMBER
, amount_debited     NUMBER
, end_of_day_balance NUMBER NOT NULL
) PARTITION BY RANGE(transaction_date)
INTERVAL (NUMTODSINTERVAL(7,'DAY'))
SUBPARTITION BY RANGE(end_of_day_balance)
SUBPARTITION TEMPLATE
( SUBPARTITION unacceptable VALUES LESS THAN (-1000)
, SUBPARTITION credit VALUES LESS THAN (0)
, SUBPARTITION low VALUES LESS THAN (500)
, SUBPARTITION normal VALUES LESS THAN (5000)
, SUBPARTITION high VALUES LESS THAN (20000)
, SUBPARTITION extraordinary VALUES LESS THAN (MAXVALUE)
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-
yyyy')));
```

En este ejemplo se usa el particionamiento por intervalo, que es un como un particionamiento por rango en el que las particiones son creadas automáticamente cuando los datos son insertados en este caso cada 7 días.

Particionado List-Range:

El particionamiento List-Range proporciona un subparticionamiento por rango dentro de una estrategia de particionamiento por lista. Por ejemplo, particionar por lista basándose en country_id, y subparticionar por rango por order_date.
List-Range es menos usado para guardar datos históricos, es más habitual el Range-List. Range-List puede ser implementado usando un interval-list particionamiento, pero en cambio list-range no soporta interval partitioning.

En el siguiente ejemplo se muestra la tabla donations que guarda las donaciones en diferentes monedas particionada por List-Range. Las donaciones se categorizan en small, medium y high, dependiendo de la cantidad. Debido a las diferencias de moneda, los rangos son diferentes.

Ejemplo:

```
CREATE TABLE donations
( id             NUMBER
, name           VARCHAR2(60)
, beneficiary    VARCHAR2(80)
, payment_method VARCHAR2(30)
, currency       VARCHAR2(3)
, amount         NUMBER
) PARTITION BY LIST (currency)
SUBPARTITION BY RANGE (amount)
( PARTITION p_eur VALUES ('EUR')
  ( SUBPARTITION p_eur_small VALUES LESS THAN (8)
  , SUBPARTITION p_eur_medium VALUES LESS THAN (80)
  , SUBPARTITION p_eur_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_gbp VALUES ('GBP')
  ( SUBPARTITION p_gbp_small VALUES LESS THAN (5)
  , SUBPARTITION p_gbp_medium VALUES LESS THAN (50)
  , SUBPARTITION p_gbp_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_aud_nzd_chf VALUES ('AUD','NZD','CHF')
  ( SUBPARTITION p_aud_nzd_chf_small VALUES LESS THAN (12)
  , SUBPARTITION p_aud_nzd_chf_medium VALUES LESS THAN (120)
  , SUBPARTITION p_aud_nzd_chf_high VALUES LESS THAN
(MAXVALUE)
  )
, PARTITION p_jpy VALUES ('JPY')
  ( SUBPARTITION p_jpy_small VALUES LESS THAN (1200)
  , SUBPARTITION p_jpy_medium VALUES LESS THAN (12000)
  , SUBPARTITION p_jpy_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_inr VALUES ('INR')
  ( SUBPARTITION p_inr_small VALUES LESS THAN (400)
  , SUBPARTITION p_inr_medium VALUES LESS THAN (4000)
  , SUBPARTITION p_inr_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_zar VALUES ('ZAR')
  ( SUBPARTITION p_zar_small VALUES LESS THAN (70)
  , SUBPARTITION p_zar_medium VALUES LESS THAN (700)
  , SUBPARTITION p_zar_high VALUES LESS THAN (MAXVALUE)
  )
```

```
, PARTITION p_default VALUES (DEFAULT)
  ( SUBPARTITION p_default_small VALUES LESS THAN (10)
  , SUBPARTITION p_default_medium VALUES LESS THAN (100)
  , SUBPARTITION p_default_high VALUES LESS THAN (MAXVALUE)
  )
) ENABLE ROW MOVEMENT;
```

En este caso tendremos 3 subparticiones por cada partición, y cada una tendrá sus propios rangos de valores. Incluso una partición se podría haber definido con más subparticiones que el resto.

Particionado List-Hash:

El particionamiento List-Hash permite crear subparticiones hash en un objeto particionado por lista. Este particionamiento es útil para grandes tablas a las que normalmente se accede en una dimensión, pero (debido al tamaño) todavía se obtiene más ventaja en parallel full o partial partition-wise joins en otra dimensión en uniones con otras tablas grandes.

El siguiente ejemplo muestra la tabla credit_card_accounts. La tabla esta particionada por lista por región, de forma que los gestores de cuentas acceden rápidamente a cuentas en su región. Además está subparticionada por hash en customer_id para mejorar uniones con otras tablas, por ejemplo las consultas contra la tabla de transacciones, la cual está también subparticionada en customer_id, pueden beneficiarse de partition-wise joins. También uniones con la tabla customers que también está particionada por hash en customer_id se beneficiarían de full partition-wise joins.

```
CREATE TABLE credit_card_accounts
( account_number  NUMBER(16) NOT NULL
, customer_id     NUMBER NOT NULL
, customer_region VARCHAR2(2) NOT NULL
, is_active       VARCHAR2(1) NOT NULL
, date_opened     DATE NOT NULL
) PARTITION BY LIST (customer_region)
SUBPARTITION BY HASH (customer_id)
SUBPARTITIONS 16
( PARTITION emea VALUES ('EU','ME','AF')
, PARTITION amer VALUES ('NA','LA')
, PARTITION apac VALUES ('SA','AU','NZ','IN','CH')
) PARALLEL;
```

Particionado List-List:

El particionamiento List-List permite particionar por lista a través de dos dimensiones, por ejemplo, particionar por lista basándose en country_id, y subparticionar estar particiones por sales_channel.

En el siguiente ejemplo se muestra la tabla current_inventory que es muy accedida. La tabla es constantemente modificada con el inventario actual en cada supermercado de una cadena. Se tienen categorizados los alimentos si son o no perecederos, y se particionan en base a esto para optimizar los suministros y las entregas desde los almacenes a los supermercados de estos alimentos.

Ejemplo

```
CREATE TABLE current_inventory
( warehouse_id      NUMBER
, warehouse_region  VARCHAR2(2)
, product_id        NUMBER
, product_category  VARCHAR2(12)
, amount_in_stock   NUMBER
, unit_of_shipping  VARCHAR2(20)
, products_per_unit NUMBER
, last_updated      DATE
) PARTITION BY LIST (warehouse_region)
SUBPARTITION BY LIST (product_category)
SUBPARTITION TEMPLATE
( SUBPARTITION perishable VALUES
('DAIRY','PRODUCE','MEAT','BREAD')
, SUBPARTITION non_perishable VALUES ('CANNED','PACKAGED')
, SUBPARTITION durable VALUES ('TOYS','KITCHENWARE')
)
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

Recopilación recomendaciones en la implantación de particionamiento

Recopilando todo lo visto hasta ahora a continuación se hace un listado resumen de las recomendaciones indicadas a tener en cuenta a la hora de establecer el tipo y configuración de particionamiento:

- Sobre el uso de Particionamiento por Rango

    - Este tipo de particionamiento es el más adecuado cuando hay datos que tienen un rango lógico por el que se pueden distribuir.
    - El rendimiento es mejor cuando los datos se distribuyen de forma equitativa a lo largo del rango, por tanto se recomienda distribuir los datos en particiones de similar tamaño.
    - Este particionamiento es ideal cuando se hacen búsquedas de datos en los que se referencian un rango de valores de la clave de particionamiento, como pueden ser operaciones BETWEEN o LIKE o expresiones regulares.
    - Es el más adecuado si se quiere mantener una ventana de borrado de datos.
    - Datos históricos normalmente trabajan con rangos de fechas y son buenas candidatas a particionamiento por Rango.

- Sobre la elección de la clave de particionamiento

- Cuando se eligen varias columnas para una clave de particionamiento, el orden es significativo. La segunda columna será examinada sólo después de que los valores de la primera columna sean iguales a la especificación del límite.
- En lugar de definir fechas usando otros tipos de datos y particiones en múltiples columnas, se recomienda particionar directamente por una columna tipo DATE.
- Se recomienda en la definición de la clave de particionamiento para fechas se use la función TO_DATE, donde se especifique claramente el formato para evitar ambigüedades.
- Evitar elegir claves de particionamiento de columnas que sufren a menudo modificaciones.

- Sobre el uso de Particionamiento por Hash

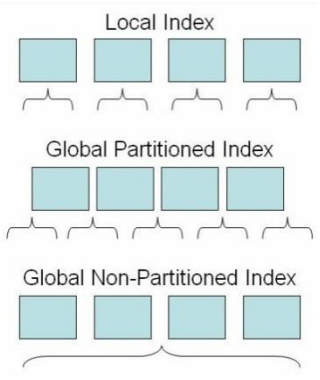
  - Se recomienda que el número de particiones sea un potencia de 2, esto es, 2, 4, 8, 16, 32 y así.
  - La cardinalidad de las columnas es muy importante cuando se elija una clave para particionamiento por hash. La función HASH funciona mejor con un número alto de valores distintos. Oracle recomienda evitar crear particionamiento por hash en columnas que tienen baja cardinalidad (columnas en las que el número de valores distintos es pequeño comparado con el número total de filas). Lo más recomendable es elegir una columna o combinación de columnas en la clave de particionamiento que sean únicas o casi únicas.
  - El particionamiento tipo Hash es usado como alternativa al particionamiento por rango especialmente cuando no hay datos que se pueden distribuir por rango de valores.
  - El particionamiento hash es más adecuado si las consultas contra los datos se usan condiciones de igualdad u operadores tipo IN.
  - Podrá ser usado para distribuir aleatoriamente los datos entre dispositivos físicos aunque no se tendrá control sobre qué partición ocupará una fila concreta. Esto podrá ser usado para evitar cuellos de botella I/O en caso de que no se utilice una técnica de almacenamiento que haga stripe y mirror entre todos los dispositivos disponibles.

- Sobre el uso de Particionamiento por Listas

  - El particionamiento por lista, como el particionamiento por rango, es ideal cuando se hacen búsquedas de datos en los que se referencian un rango de valores de la clave de particionamiento, como pueden ser operaciones BETWEEN o LIKE, o expresiones regulares.
  - Su uso permitirá mapear filas a particiones basándose en valores discretos.
  - En caso de uso de particionamiento por lista automática, no es recomendado en columnas con datos que varían frecuentemente.

Particionamiento de Indices. Estrategias y usos. Ejemplos.

Existen tres tipos diferentes de índices que pueden utilizarse en una tabla particionada, Locales, Globales Particionados y Globales no particionados.



Independientemente de la estrategia elegida para la partición de índices, un índice puede estar asociado o no con la estrategia de partición de la tabla subyacente. Podremos tener una combinación de índices globales y locales en la misma tabla particionada.

Dependiendo de las consultas y de la decisión que se tome para la creación de los índices el rendimiento global de una base de datos puede sufrir una disminución significativa.

Una operación aparentemente sencilla puede obviar la utilización de un índice, pero si el tiempo de respuesta en una aplicación depende de la utilización de dicho índice, su no utilización puede provocar un grave decremento en el rendimiento global y lo que podría ser una búsqueda de segundos podría alterarse y tardar minutos o incluso horas.

Debe tenerse en cuenta que cuando utilizamos un índice global particionado o no particionado puede resultar "traumática" (pensemos en el coste en tiempo y recursos) la recreación de un índice de este tipo de varios millones de registros cuando se realiza cualquier operación de mantenimiento sobre la tabla. Por ejemplo después de realizar el truncado o "Split" de una partición de la tabla (alter table drop partition) el índice global quedara en estado "UNUSABLE" lo que hará imposible su utilización y provocará que cualquier plan de ejecución que utilizara dicho índice experimente una pérdida de rendimiento. Para evitar situaciones de este tipo podremos utilizar el atributo "**update <global> indexes**". Como se verá más adelante esto tendrá sus implicaciones.

Se pueden crear índices particionados de distinto tipo: B*Tree, Bitmap, bitmap join y de funciones, pero hay algunas restricciones. Por ejemplo se pueden crear índices bitmap con la restricción de que estos deberán ser locales, no se pueden crear índices bitmap globales.

Veamos más a fondo alguno de las características de los índices que pueden crearse sobre tablas particionadas.

Índices Locales

Son índices de tablas particionadas que se asocian con la tabla subyacente, por lo que "**heredan su estrategia de particionamiento**". En consecuencia, cada partición de un índice

local se corresponde con una y sólo una partición de la tabla subyacente (en pocas palabras, índice y tabla están particionados en forma equivalente, o "equiparticionados"). La asociación entre índice y tabla simplifica el mantenimiento de la partición. Para dar un ejemplo, cuando se quita una partición de la tabla, Oracle sólo tiene que quitar la partición correspondiente en el índice sin que haga falta llevar a cabo otras tareas de mantenimiento sobre los índices. Los índices locales normalmente se utilizan en entornos de DW.

Un índice local puede ser creado como `UNIQUE` sólo si las columnas que forman la clave de particionamiento forman parte de un subconjunto de las columnas del índice. Esta restricción garantiza que las posibles claves idénticas del índice siempre estarán mapeadas a la misma partición, donde las violaciones de unicidad pueden ser detectadas.

Cada partición del índice se corresponde con su partición de la tabla.

Los índices locales tienen estas ventajas:

- Sólo una partición del índice deberá ser reconstruida en caso de cualquier operación de mantenimiento como mover o borrar una partición al sobrepasar la política de retención mientras que el resto de los índices locales seguirán siendo válidos. Esto es, los índices locales soportan mejor el movimiento de datos en tablas históricas.
- La duración de las tareas de mantenimiento será proporcional al tamaño de la partición si la tabla sólo tiene índices locales.
- Oracle tomará ventaja del hecho de que el índice esté equiparticionado con la tabla a la que pertenece, mejorando el plan de ejecución.
- En SQLs con predicados con la clave de particionamiento, se podrán evitar leer particiones del índice (partition pruning).
- Los índices locales simplifican las tareas de recuperación de un tablespace hasta un momento en el tiempo. La recuperación de una partición o subpartición de una tabla, también se deberá recuperar sus correspondientes entradas del índice hasta el mismo punto en el tiempo. Esto sólo se podrá realizar si es un índice local.

La creación de este tipo de índice particionado solo requiere añadir la cláusula LOCAL al final de la sentencia CREATE INDEX, por ejemplo:

```
CREATE TABLE sales
     (acct_no               NUMBER(5)     NOT NULL,
     sales_person_id        NUMBER(5)     NOT NULL,
     po_number              VARCHAR2(10) NOT NULL,
     po_amount              NUMBER(9,2),
     month_no               NUMBER(2)     NOT NULL)
     PARTITION BY RANGE (month_no)
          (PARTITION first_qtr     VALUES LESS THAN (4),
           PARTITION sec_qtr       VALUES LESS THAN (7),
           PARTITION thrd_qtr      VALUES LESS THAN(10),
           PARTITION frth_qtr      VALUES LESS THAN(13),
           PARTITION bad_qtr       VALUES LESS THAN (MAXVALUE));

CREATE INDEX pt_sales
     ON sales (month_no, sales_person_id,acct_no,po_number)
            LOCAL;
```

Puede definirse en que tablespace se almacena cada una de las particiones del índice:

```
CREATE INDEX pt_lc_sales
```

```
ON sales (month_no, sales_person_id,acct_no,po_number)
       LOCAL(
     PARTITION i_first_qtr      TABLESPACE part_ind_tbsp1,
     PARTITION i_sec_qtr        TABLESPACE part_ind_tbsp2,
     PARTITION i_thrd_qtr       TABLESPACE part_ind_tbsp3,
     PARTITION i_frth_qtr       TABLESPACE part_ind_tbsp4,
     PARTITION i_bad_qtr        TABLESPACE part_ind_tbsp5);
```

Los índices particionados locales se dividen en:

- Local prefixed indexes

  En este caso, la clave de particionamiento aparece en la parte inicial de definición del índice y la clave de subparticionamiento aparece en la clave del índice. Los índices locales prefixed pueden ser únicos o no-únicos.
  En el ejemplo anterior la clave de particionamiento month_no aparece en parte inicial de la definición del índice, por tanto será un índice prefixed.

- Local nonprefixed indexes

  En este caso, la clave de particionamiento no aparece en la parte de la izquierda de la definición del índice o si la clave de subparticionamiento no aparece en la clave del índice. No se podrá un índice local nonprefixed único a menos que la clave de particionamiento esté incluida en el índice.

El atributo prefixed o no-prefixed no puede ser directamente especificado, simplemente será el resultado de la combinación de las columnas de índice con las de la especificación de la clave de particionamiento.

Índices Prefixes versus No-Prefixed

Ambos tipos tienen la ventaja de poder utilizar la eliminación de particiones (partition pruning), que ocurre cuando el optimizador mejora el rendimiento en el acceso a datos excluyendo particiones que no son necesarias. Esta eliminación de particiones dependerá del predicado de la sentencia sql. Un sentencia sql que usa un índice local prefixed siempre permitirá la eliminación de particiones en cambio un local noprefixed no siempre.

El coste de la utilización de índices "nonprefixed" es mayor que el de la utilización de los "prefixed". Utilizar índices locales "prefixed" cuando sea posible ya que asegura que el optimizador ignore aquellas particiones que no coinciden con las operaciones vinculadas a la clave del índice.

Aún así el uso de índices locales no-prefixed es útil si es importante tener acceso rápido a una columna que no es parte de la clave de particionamiento (por ejemplo búsquedas por un valor de la columna account_number, cuando esta columna es la columna inicial del índice y la clave de partición es otra columna).

Índices Particionados Globales

Son índices de tablas particionadas o no particionadas, que utilizan una clave o estrategia de particionamiento diferente de la que se usó en las tablas. Los índices de

este tipo pueden particionarse por rango o por hash y no están equiparticionados con respecto a la tabla subyacente.

Aunque los índices sólo pueden particionarse por rango o hash, podrán ser definidos en cualquier tipo de tabla particionada.

Las operaciones de mantenimiento de las particiones tienen un impacto mayor en este tipo de índices que en los locales dado que dichas operaciones pueden afectar a más de una partición.

Los índices globales permiten definir particiones diferentes que las especificadas en las particiones de la tabla, esto significa que los rangos de un índice global pueden contener particiones con valores diferentes de los de su tabla asociada.

Podría definirse un índice global con las mismas particiones y los límites que la tabla, pero esto no tiene mucho sentido ya que debe mantenerse la conexión entre las particiones de índice y las particiones de la tabla, es decir estaremos obligados a realizar el mantenimiento del índice cuando se añada o se suprima una partición en la tabla, además de que Oracle no tendría la ventaja de que tabla e índice estuviera equiparticionados cuando se generan planes de ejecución.

Un índice global no puede ser subparticionado.

Un índice particionado global es prefixed si está particionado por lo que aparece en la parte de la izquierda del índice, no-prefixed si no aparece. Oracle sólo soporta índices globales prefixed, si no, se generará un error ORA-14038 al intentar crearlo.

Los índices globales pueden ser únicos o no únicos.

Sólo los índices B*tree pueden ser particionados globales.

Los índices particionados globales pueden ser útiles si hay sentencias que usan un acceso a la tabla a través de un índice y devuelve pocas filas, y particionando el índice se puede eliminar grandes porciones del índice en la mayoría de las sentencias. En una tabla particionada, se consideraría un índice global particionado si la columnas o columnas para poder hacer un partition pruning no incluyen la clave de particionamiento de la tabla.

El uso de índices globales puede ser útil en entorno OLTP en los que una tabla está particionada por una clave, por ejemplo la columna employees.department_id, pero una aplicación necesita acceso a los datos con diferentes clave por ejemplo employee_id o job_id. En estos casos el uso de un índice global podría tendrá mejor rendimiento que un índice local no-prefixed porque se minimizará el número de búsquedas en las particiones del índice.

Cuando se define un índice global, han de especificarse las particiones en la sentencia CREATE INDEX el atributo GLOBAL y la definición del particionamiento. En el siguiente ejemplo se crea un índice global particionado por hash en la tabla sales:

```
CREATE INDEX cust_id_prod_id_global_ix
ON sales(cust_id,prod_id)
GLOBAL PARTITION BY HASH (cust_id)
```

```
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
)
PARALLEL NOLOGGING;
```

O en el siguiente para una particionado por Rango:

```
CREATE INDEX idxB ON tabA(colB) GLOBAL PARTITION BY RANGE (colB)
(
PARTITION VALUES LESS THAN 10,
PARTITION VALUES LESS THAN 100,
PARTITION VALUES LESS THAN (MAXVALUE)
);
```

La palabra clave MAXVALUE establece un límite superior ilimitado. Puede utilizar MAXVALUE para la última partición en un índice o una tabla.

Los índices globales particionados por hash podrían mejorar el rendimiento si hay contención porque un índice crece de forma monótona. Esto es, si la mayoría de las inserciones ocurren solo a la derecha del índice. Es muy común en entornos OLTP el uso de secuencias para crear claves, lo que puede producir que muchos procesos de inserción compitan por los mismos bloques hojas del índice. Por ejemplo, si se crean 4 particiones hash en el índice, habrá 4 segmentos en los que se insertarán datos, reduciendo la concurrencia con un factor de 4.

En el siguiente ejemplo se muestra la creación de un índice único en la columna order_id de la tabla orders. El order_id de una aplicación OLTP se rellena usando una secuencia numérica. El índice único usa un particionamiento hash para reducir la contención en los insert de valores de order_id que monótonamente van creciendo.

La clave única es luego usada como restricción de primary key.

```
CREATE UNIQUE INDEX orders_pk
ON orders(order_id)
GLOBAL PARTITION BY HASH (order_id)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
) NOLOGGING;

ALTER TABLE orders ADD CONSTRAINT orders_pk
PRIMARY KEY (order_id)
USING INDEX;
```

Índices Globales No Particionados

Un índice noparticionado global es un índice normal en una tabla particionada y se comporta de igual forma que un índice no particionado.

Se utilizan comúnmente en entornos OLTP y ofrecen un acceso eficiente a cualquier registro individual. Se podrán usar un índice global noparticionado para forzar una clave primaria o única. También será útil en sentencias sql que devuelven pocas filas

porque usan una clausula where con igualdades o listas IN en una columna o un conjunto de columnas que no están en la clave de particionamiento. En estos casos es más rápido escanear un único índice que tener que escanear muchas diferentes particiones para buscar las filas que cumplen las condiciones.

Como se dijo anteriormente, índices únicos en columnas que no contienen las claves de particionamiento (no-prefixed), no podrán ser locales, y por tanto tendrán que ser globales. Esto normalmente no aplica a Data Warehouse, donde las claves únicas a veces no son creadas debido a la pérdida de rendimiento en los procesos de carga para forzar el cumplimiento de la restricción de unicidad. Además los índices globales pueden ser muy grandes en tablas de billones de filas.

El siguiente ejemplo muestra la creación de un índice único global no particionado en la tabla sales:

```
CREATE UNIQUE INDEX sales_unique_ix

ON sales(cust_id, prod_id, promo_id, channel_id, time_id)

PARALLEL NOLOGGING;
```

Considerar no crear este tipo de índices en bases de datos con pequeñas ventanas de mantenimiento, ya que la mayoría de las operaciones de mantenimiento de las particiones de la tabla invalidan los índices noparticionados, y forzará un index rebuild.

Otras consideraciones sobre la creación de índices en tablas particionadas:

- Se pueden crear índices bitmap en tablas particionadas, con la restricción de que estos deberán ser locales, no se pueden crear índices bitmap globales.

- Cuando se crean índices particionados en tablas particionadas compuestas hay que tener en cuenta que:
  - Estos índices subparticionados locales estarán equiparticionados con la tabla base en el siguiente sentido:
    - Contendrán tantas particiones como la tabla base
    - Cada partición del índice consistirá de tantas subparticiones como la correspondiente partición de la tabla base. Podremos verlas en la vista dba_ind_subpartitions
    - Las entradas del índice para las filas de una subpartición concreta de la tabla base estarán guardadas en la correspondiente subpartición del índice.
  - Se podrán especificar la cláusula tablespace en ambos niveles de particionamiento o subparticionamiento del índice.

Índices parciales en tablas particionadas

En versión 12c se añade la funcionalidad de "Partial Indexes" que permite que los índices locales y globales sobre tablas particionadas puedan ser creados en un

subconjunto de particiones de una tabla, permitiendo más flexibilidad en la creación del índice.

Esta operación está soportada usando la cláusula INDEXING ON/OFF en la definición de la tabla o en sus particiones en el create o alter de la tabla. Para que el índice sea parcial el índice se creará utilizando la cláusula PARTIAL, en otro caso será FULL (por defecto).

Cuando se indica la cláusula PARTIAL en la creación de un índice en una tabla:

- En local indexes:
  Una partición del índice es creada usable si la indexación es indicada como activa a nivel de partición de la tabla, en otro caso la partición será unusable. Este comportamiento puede ser sobreescrito especificando USABLE/UNUSABLE a nivel del índice o a nivel de partición del índice.

- En global indexes: Indexará sólo aquellas particiones en las que la indexación es indicada como activa, y excluye las otras.

Esta funcionalidad no está soportada para índices únicos, o índices usados para forzar una constraint unique.

Ejemplo:

```
CREATE TABLE orders (
  order_id NUMBER(12),
  order_date DATE CONSTRAINT order_date_nn NOT NULL,
  order_mode VARCHAR2(8),
  customer_id NUMBER(6) CONSTRAINT order_customer_id_nn NOT NULL,
  order_status NUMBER(2),
  order_total NUMBER(8,2),
  sales_rep_id NUMBER(6),
  promotion_id NUMBER(6),
  CONSTRAINT     order_mode_lov     CHECK     (order_mode     in
('direct','online')),
  CONSTRAINT order_total_min CHECK (order_total >= 0))
   INDEXING OFF
   PARTITION BY RANGE (ORDER_DATE)
   (PARTITION ord_p1 VALUES LESS THAN (TO_DATE('01-MAR-1999','DD-
MON-YYYY'))
     INDEXING ON,
   PARTITION ord_p2 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-
MON-YYYY'))
     INDEXING OFF,
   PARTITION ord_p3 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-
MON-YYYY'))
     INDEXING ON,
   PARTITION ord_p4 VALUES LESS THAN (TO_DATE('01-MAR-2000','DD-
MON-YYYY')),
   PARTITION ord_p5 VALUES LESS THAN (TO_DATE('01-MAR-2010','DD-
MON-YYYY')));
```

Esta SQL DDL crea una tabla con estas características:

- Las particiones ORD_P1 y ORD_P3 son incluidas en todos los índices parciales globales.
- Las particiones de un índice local (para índices creados PARTIAL) correspondientes a las 2 particiones de la tabla indicadas serán por defecto usable.
- Las otras particiones son excluidas de todos los índices globales parciales y para índices locales creados con PARTIAL las otras particiones se crearán como unusable.

Para que un índice siga las propiedades de indexación indicadas en el SQL de creación la tabla tendrá que especificar la cláusula INDEXING PARTIAL, tanto para índice local como global, de esta forma será un índice parcial. Por ejemplo:

```
CREATE INDEX ORDERS_ORDER_TOTAL_GIDX ON ORDERS (ORDER_TOTAL)
   GLOBAL INDEXING PARTIAL;
```

El índice ORDERS_ORDER_TOTAL_GIDX será creado sólo para indexar aquellas particiones que tienen INDEXING ON, y excluye las demás particiones.

Compresión en índices particionados

El uso de compresión puede reducir el espacio de almacenamiento requerido para los índices. En versión 12c existen dos tipos de compresión:

- Key compression

  Si no se tiene licenciado la opción Advanced Compression se puede usar la compresión tradicional de índices "key compression" o también llamada "prefix compression" para comprimir algunas o todas las particiones de un índice B-tree.

  El siguiente ejemplo create un índice local particionado en todas las particiones excepto en la más reciente:

  ```
  CREATE INDEX i_cost1 ON costs_demo (prod_id) COMPRESS LOCAL
     (PARTITION costs_old, PARTITION costs_q1_2003,
      PARTITION costs_q2_2003, PARTITION costs_recent NOCOMPRESS);
  ```

  No se puede especificar COMPRESS (o NOCOMPRESS) explícitamente a nivel de subpartición. Todas las subparticiones de una partición heredan la configuración de key compression de la partición padre.

  No todos los índices son buenos candidatos a ser comprimidos con key compression, y si el número de columnas indexadas es mayor que 1 habría que añadir el rango más adecuado para que la compresión "key compression" sea efectiva.

- Advanced index compression

  Advanced index compression mejora significativamente los ratios de compresión y consigue mantener un eficiente rendimiento en el acceso al índice. Incluso en índices que no son buenos candidatos a la prefix compression pueden ser comprimidos con el advanced compression.

  Se puede especificar a nivel de partición el tipo de compresión o a nivel del índice cuyas particiones que no lo especifiquen heredarán los atributos de compresión. Ejemplos:

```
CREATE INDEX my_test_idx ON test(a, b) COMPRESS ADVANCED HIGH
LOCAL
   (PARTITION p1 COMPRESS ADVANCED LOW,
    PARTITION p2 COMPRESS,
    PARTITION p3,
    PARTITION p4 NOCOMPRESS);

CREATE INDEX my_test_idx ON test(a, b) NOCOMPRESS LOCAL
   (PARTITION p1 COMPRESS ADVANCED LOW,
    PARTITION p2 COMPRESS ADVANCED HIGH,
    PARTITION p3);
```

NOTA: "Advanced Index Compress" que está disponible a partir de versión 12.1.0.2 y requiere licenciamiento extra de la opción de Advanced Compression Option.

Mantenimiento de los índices en tablas particionadas. Estado UNUSABLE.

Por defecto, muchas operaciones en tablas particionadas marcarán UNUSABLE e invalidarán el índice global o las particiones concretas afectadas en caso de índice local. La mayoría de las operaciones de mantenimiento en la tabla invalidan los índices globales particionados, forzando a que se tenga que hacer un rebuild, por tanto son más difíciles de manejar que los índices particionados locales.

El que una operación marque una partición de un índice como UNUSABLE dependerá del tipo de partición (por ejemplo add partition no marca UNUSABLE para partición por rango, en cambio sí para HASH).

A partir del 11.2 índices en estado UNUSABLE no consumen espacio.

Cuando un índice está marcado como UNUSABLE, estos son ignorados por el optimizador, si el parámetro SKIP_UNUSABLE_INDEXES = TRUE (por defecto así). Si se configura a FALSE, se generará un error en la ejecución de la sentencia.

Por defecto las siguientes operaciones en tablas normales (no IOTs) marcan los índices globales como unusable:

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

Para ver más información sobre esto, ver las siguientes referencias:

Note 165917.1 Maintenance Commands That Cause Indexes to Become Unusable

Note 1054736.6 HOW DO INDEXES BECOME INDEX UNUSABLE?

Operations that mark global indexes unusable (VLDB and Partitioning Guide)

Se puede evitar el que un índice se marque como UNUSABLE, si se utiliza la opción "**UPDATE INDEXES**" en la operación de alter table. Especificando esta clausula se le dirá a la base de datos que modifique el índice (ambos índices locales y globales) a la vez que se ejecuta el comando DDL. El índice estará disponible y online mientras esté la operación y no se tendrá que reconstruir después de la operación.
Para modificar sólo los índices globales, se usará la clausula UPDATE GLOBAL INDEXES.

Las siguientes operaciones soportan la cláusula UPDATE INDEXES:

```
 ADD       PARTITION | SUBPARTITION
COALESCE  PARTITION | SUBPARTITION
DROP      PARTITION | SUBPARTITION
EXCHANGE  PARTITION | SUBPARTITION
MERGE     PARTITION | SUBPARTITION
MOVE      PARTITION | SUBPARTITION
SPLIT     PARTITION | SUBPARTITION
TRUNCATE  PARTITION | SUBPARTITION
```

El uso de UPDATE INDEXES tiene varias implicaciones:

- Los comandos DDL tardarán más en ejecutar, ya que los índices que se deberían marcar como UNUSABLE son ahora modificados. Se debería comparar el tiempo de realizar la operación con esta cláusula respecto a lanzar el DDL sin la cláusula y después hacer un rebuild del índice. Como regla general es más rápido el uso de la cláusula si el tamaño de la partición implicada es menor del 5% del tamaño de la tabla.
- Las operaciones de DROP, TRUNCATE y EXCHANGE dejan de ser operaciones rápidas. (A partir de versión 12c una nueva funcionalidad llamada mantenimiento asíncrono de índices hacen que `DROP` y `TRUNCATE` sí sean operaciones rápidas).
- El uso de UPDATE INDEXES generará información de undo y redo mode. En cambio cuando se hace un rebuild de un índice global, se puede usar el modo NOLOGGING. Además haciendo un rebuild del índice entero tendremos un índice más compacto y más eficiente.

Recomendaciones

Al igual que las tablas particionadas, los índices particionados mejorarán la administración, la disponibilidad, el rendimiento y la escalabilidad. Pueden dividirse de forma independiente (índices globales) o vinculados al método de partición de una tabla (índices locales).

Independientemente del tipo de índice que se cree, la creación de un índice debería estar únicamente justificada si realmente va a ser usado, esto es, si las sentencias que lo usen mejoran el tiempo de respuesta, o porque se necesita para cumplir una restricción única o clave primaria. Todas las recomendaciones genéricas que aplican a índices en tablas no particionadas aplicarán también a tablas particionadas (selectividad, clustering factor, etc.).

Pautas para decidir el tipo de particionamiento en el índice a usar:

En términos de decidir entre un índice particionado local o global pueden considerarse las siguientes pautas en el orden en que están listadas:

1. Si la clave de particionamiento de la tabla es la misma, o un subconjunto de la clave del índice, entonces siempre se debería usar un índice local.
2. Si lo anterior no aplica, considerar si el índice es único o no único. Si el índice es único y no incluye las columnas de la clave de particionamiento, habrá que utilizar un índice global.
3. Si lo anterior no aplica, habría que considerar el grado de facilidad en la gestión que se busca, esto es, si la prioridad es la gestión, considerar un índice local.
4. Finalmente, se podría considerar la naturaleza de la base de datos, bases de datos OLTP obtendrán mejores tiempos de respuesta con índices globales, y en entornos Data Warehouse obtendremos mejor rendimiento con índices locales.

Otras recomendaciones a tener en cuenta:

- Utilizar índices locales "prefixed" cuando sea posible ya que provocan que el optimizador ignore aquellas particiones que no coinciden con las operaciones vinculadas a la clave del índice.
- De forma general los índices locales son los más adecuados para los datos de aplicaciones DW, mientras que los índices globales trabajarán mejor con sistemas OLTP.
- Si las ventanas de mantenimiento son pequeñas mejor usar índices locales.
- En un entorno Data Warehouse de forma general si una tabla justifica particionamiento, normalmente el índice debería particionarse también.
- Los comandos DDL sobre tablas particionadas con la cláusula UPDATE INDEXES para evitar índices UNUSABLE tardarán más en ejecutarse. Se debería comparar el tiempo de realizar la operación con esta cláusula respecto a lanzar el DDL sin la cláusula y después hacer un rebuild del índice. Haciendo un rebuild del índice entero tendremos un índice más compacto y más eficiente.
- En sistemas OLTP el uso de índices globales será más eficiente en sentencias sql que devuelven pocas filas porque usan una clausula where con igualdades o listas IN en una columna o un conjunto de columnas que no están en la clave de particionamiento. En estos casos es más rápido escanear un único índice global que un índice local no-prefixed donde habrá que escanear muchas diferentes particiones para buscar las filas que cumplen las condiciones.
- En sistemas OLTP los índices globales particionados por hash podrían mejorar el rendimiento si hay contención porque un índice crece de forma monótona.

Partition Pruning y Partition-Wise Joins

Con el uso de particionamiento se podrá mejorar el rendimiento ya que la base de datos internamente podrá limitar la cantidad de datos a examinar y mejorará la ejecución en paralelo.

Estas mejoras en el rendimiento vendrán por el uso de las siguientes características:

- Partition Pruning
- Partition-wise Joins

Partition Pruning

Esta es la más simple técnica y a la vez la que produce una mayor ganancia de rendimiento por el uso de particionamiento. Partition Pruning consiste en que cuando se realiza una sentencia SQL sobre una tabla particionada, el optimizador analizará el FROM y el WHERE y accederá al conjunto mínimo de particiones necesarias que son relevantes para resolver la sentencia.

La base de datos realizará de forma transparente al usuario el partition pruning cuando se usen predicados con rangos, LIKE, igualdades y listas IN en columnas particionadas por rango o lista, y cuando se usen predicados con igualdades y listas IN en columnas particionadas por hash.

Beneficios del Partition Pruning

Partition pruning reduce dramáticamente la cantidad de datos a devolver desde disco y bajando el tiempo de proceso, y así mejorando el rendimiento de la sentencia y bajando la utilización de recursos.

Si se particiona el índice y la tabla en diferentes columnas (con un índice particionado global), entonces el partition pruning podría también eliminar particiones del índice aunque las particiones de la tabla no puedan ser eliminadas.

El partition pruning se divide en Static pruning y Dynamic pruning. Dependiendo de la sentencia SQL, Oracle usará Static pruning o Dynamic pruning. El Static pruning ocurre en tiempo de compilación, y la información sobre las particiones a las que acceder ya se conoce. Dynamic pruning ocurre en tiempo de ejecución, las particiones a las que acceder no se conocerá previamente.

Un ejemplo de static pruning es una sentencia SQL que contiene un WHERE con una condición que usa una constante literal en una columna de la clave de particionamiento. Un ejemplo de dynamic pruning es el uso de bind variables o funciones en la condición WHERE.

El uso de static partition pruning será mejor que el dynamic pruning.

Como identificar si se está usando Partition Pruning

Para identificar si se está usando partition pruning habrá que revisar el plan de ejecución de la sentencia, bien usando el comando EXPLAIN PLAN o revisando el plan de ejecución generado en la SQL Area si la sentencia ya ha sido ejecutada.

La información de partition pruning se refleja en las columnas PSTART (PARTITION_START) y PSTOP (PARTITION_STOP).

En la nota de MyOracleSupport How to see Partition Pruning Occurred? [ID 166118.1] podemos ver varios métodos para determinar el uso o no de partition pruning.

Static Partition Pruning

Cuando se usan predicados estáticos Oracle determinará las particiones a acceder en tiempo de compilación (parse). Si sacamos un plan de ejecución de la sentencia se verá que cuando se hace el parse de la sentencia Oracle determina las particiones contiguas a acceder, y rellenará las columnas PSTART y PSTOP con el principio y final de las particiones a acceder. En dynamic pruning mostrará el valor KEY en PSTART y PSTOP.

Ejemplo:

```
SQL> explain plan for select * from sales where time_id =
to_date('01-jan-2001', 'dd-mon-yyyy');
Explained.

SQL> select * from table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
-
Plan hash value: 3971874201
--------------------------------------------- ----------------------------
| Id | Operation              | Name  | Rows | Cost (%CPU)| Pstart| Pstop |
--------------------------------------------- ----------------------------
|  0 | SELECT STATEMENT       |       | 673  | 27      (8)|       |       |
|  1 |  PARTITION RANGE SINGLE|       | 673  | 27      (8)| 17    | 17    |
|* 2 |   TABLE ACCESS FULL    | SALES | 673  | 27      (8)| 17    | 17    |
------------------------------------------------------------------------------
--
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("TIME_ID"=TO_DATE('2001-01-01 00:00:00', 'yyyy-mm-dd
hh24:mi:ss'))
```

En este ejemplo el plan muestra que Oracle accederá a la partición 17. En la columna OPERATION muestra un PARTITION RANGE SINGLE que indica que sólo accederá a una única partición. Si OPERATION mostrara PARTITION RANGE ALL entonces todas las particiones serían accedidas sin que tenga lugar una eliminación de particiones a leer.

Dynamic Partition Pruning

El dynamic pruning ocurrirá si el pruning es posible pero no un static pruning. El partition pruning tendrá que ser calculado en tiempo de ejecución. Por ejemplo, con

el uso de bind variables en el where, con el uso de subqueries, start tranformations, Nested Loop joins se usará Dynamic Pruning.

Por ejemplo con el uso de bind variables:

```
SQL> explain plan for select * from sales s where time_id in ( :a, :b, :c,
:d);
Explained.
```

```
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 513834092
------------------------------------------------------------------------------------------
| Id | Operation                          |    Name |Rows|Cost (%CPU)| Pstart| Pstop|
------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                   |         |2517|    292 (0)|       |      |
|  1 |  INLIST ITERATOR                   |         |    |           |       |      |
|  2 |   PARTITION RANGE ITERATOR         |         |2517|    292 (0)|KEY(I) |KEY(I)|
|  3 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES |2517|    292 (0)|KEY(I) |KEY(I)|
|  4 |     BITMAP CONVERSION TO ROWIDS    |         |    |           |       |      |
|* 5 |      BITMAP INDEX SINGLE VALUE     |SALES_TIME_BIX|          |KEY(I) |KEY(I)|
------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
5 - access("TIME_ID"=:A OR "TIME_ID"=:B OR "TIME_ID"=:C OR "TIME_ID"=:D)
```

En este ejemplo el acceso será por un índice bitmap particionado local donde también se producirá al partition pruning.

Más información y ejemplos sobre Static y dynamic partition pruning en Database VLDB and Partitioning Guide 12cR2. "Partitioning for Availability, Manageability, and Performance" y en Advanced Partition Pruning Techniques

Recomendaciones en el uso de Partition Pruning

- Para las sentencias el uso de static partition pruning será mejor que el dynamic pruning especialmente para accesos a una única partición.

- Se recomienda evitar conversiones del tipo de datos. Las conversiones de tipos de datos hace que el resultado sea usar un dynamic pruning cuando se podría haber realizado un static pruning. El caso más común de conversión es con el uso de tipos DATE, y el uso de formatos distintos al de la configuración NLS del cliente. Para evitar esto se recomienda usar la función TO_DATE con el formato adecuado en las clausulas where, de esta forma la base de datos puede determinar de forma inequívoca el valor de la fecha y potencialmente usarla para realizar un static pruning.

  Ejemplo de no uso de static partition pruning:

```
alter session set nls_date_format='fmdd Month yyyy';

explain plan for select sum(amount_sold)
from sales
where time_id between '01-JAN-2000' and '31-DEC-2000' ;
```

El plan de ejecución muestra lo siguiente:

```
-------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |   525   (8)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |            |       |       |
|*  2 |   FILTER                   |       |       |            |       |       |
|   3 |    PARTITION RANGE ITERATOR|       |   230K|   525   (8)|   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL      | SALES |   230K|   525   (8)|   KEY |   KEY |
-------------------------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('01-JAN-2000')<=TO_DATE('31-DEC-2000'))
   4 - filter("TIME_ID">='01-JAN-2000' AND "TIME_ID"<='31-DEC-2000')
```

Este plan usa dynamic pruning que es menos eficiente que el static pruning execution plan, porque el formato indicado en el where no se corresponde con el nls del cliente. Para evitar esto, es más recomendable explícitamente indicar el formato del tipo de fecha en la sentencia:

```
explain plan for select sum(amount_sold)
from sales
where time_id between to_date('01-JAN-2000','dd-MON-yyyy')
  and to_date('31-DEC-2000','dd-MON-yyyy') ;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |   127   (4)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |            |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |   230K|   127   (4)|    13 |    16 |
|*  3 |    TABLE ACCESS FULL       | SALES |   230K|   127   (4)|    13 |    16 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("TIME_ID"<=TO_DATE(' 2000-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

- Se recomienda evitar el uso explícito o implícito de funciones en las columnas de la partición. Si las consultas usan de forma común llamadas a funciones, se podría considerar en versión 11g el uso de una columna virtual y particionamiento en esta columna virtual para beneficiarse del partition pruning en estos casos.

  Por ejemplo, si se considerara la siguiente forma de construir la sentencia para obtener los mismos datos del año 2000 que en el ejemplo anterior, pero usando una función como to_char en la columna clave de particionamiento. Esto deshabilita el partition pruning y el plan de ejecución muestra un full table scan sin partition pruning:

```
EXPLAIN PLAN FOR
SELECT SUM(amount_sold)
FROM sales
WHERE TO_CHAR(time_id,'yyyy') = '2000';
```

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |     1 |   527   (9)|       |       |
|   1 |  SORT AGGREGATE      |       |     1 |            |       |       |
|   2 |   PARTITION RANGE ALL|       |  9188 |   527   (9)|     1 |    28 |
|*  3 |    TABLE ACCESS FULL | SALES |  9188 |   527   (9)|     1 |    28 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(TO_CHAR(INTERNAL_FUNCTION("TIME_ID"),'yyyy')='2000')
```

Nota: En los ejemplos anteriores para hacer más legible los planes de ejecución se han eliminado algunas columnas del sql plan.

Partition-wise Joins

El particionamiento también mejorará el rendimiento en uniones (joins) entre varias tablas, usando una técnica conocida como partition-wise join. Partition-wise joins mejorará el tiempo de respuesta especialmente en ejecuciones con parallel ya que minimizará la cantidad de datos intercambiados entre los procesos de ejecución paralela.

Los Partition-wise joins pueden ser full o partial. Oracle decidirá qué tipo de join usará:

Full Partition-wise Join

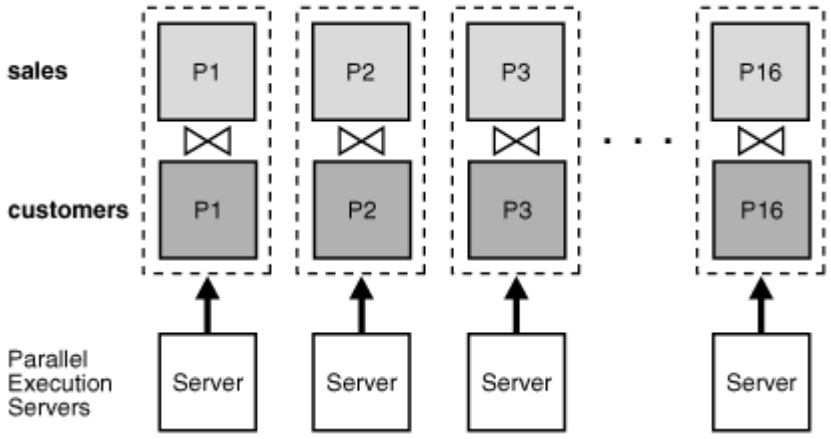
Un full partition-wise join divide una unión muy grande en pequeñas uniones entre parejas de particiones desde las tablas que se hace la unión. Para usar esta técnica, ambos objetos (podrán ser tablas, índices o particiones) deberán estar equiparticionados (tendrán ambos las mismas particiones y mismas definiciones) en las claves de unión, o usar el método Reference Partitioning de 11g.

Reference partitioning es una fácil forma de coparticionar dos tablas de forma que el optimizador pueda considerar un full partition-wise join si las tablas son unidas en una consulta.

Cuando la ejecución de la unión es serial (no parallel), el join se realiza entre parejas de particiones poco a poco. Cuando una pareja de particiones han sido unidas, la unión de otra pareja comienza, la unión completa finaliza cuando todas las parejas han sido procesadas.

Se podrá mejorar el tiempo de respuesta ejecutando un full partition-wise join en paralelo. El grado de paralelismo estará limitado al número de particiones.

En la imagen se muestra un ejemplo de la ejecución paralela de un full partition-wise join:



Ejemplo:

Considerar por un lado la tabla sales particionada por fecha usando la columna sale_date y subparticionada por hash usando la columna customer_id, y por otro, la tabla customers particionada por hash usando la columna customer_id. En este caso, el número de particiones hash es el mismo para la tabla customers como para cada partición por rango de la tabla sales. Si consideramos la siguiente query:

```
select *
  from sales, customers
 where sale_date between to_date(1-jan-2008)
   and to_date(31-jan-2008)
   and sales.customer_id = customers.customer_id
```

Las tablas sales y customers serán equiparticionadas. Por consiguiente un full partition-wise join podrá ser usado.

Partial Partition-wise Join

Al contrario de un full partition-wise joins, partial partition-wise joins sólo requiere el particionamiento de una tabla en la clave del join, no ambas tablas. La tabla particionada será referenciada como la tabla de referencia. La otra tabla podrá o no estar particionada. Para ejecutar un partial partition-wise join, Oracle dinámicamente reparticionará la otra tabla basándose en el particionamiento de la tabla de referencia. Una vez la otra tabla es reparticionada, la ejecución es similar a un full partition-wise join. Oracle sólo realiza partial partition-wise joins en paralelo.

Para más información y ejemplos sobre el Partial partition-wise join revisar:
Database VLDB and Partitioning Guide 12cR2. Partitioning for Availability, Manageability, and Performance. Partial Partition-Wise Joins

Recomendaciones en el uso de Partition-wise joins:

- El grado de paralelismo no necesita ser igual al número de particiones, pero se recomienda que el número de particiones sea un múltiplo del grado de paralelismo usado.

- En entornos RAC en plataformas MPP (Massive parallel processing), los nodos donde se localicen las particiones será crítico para conseguir una buena escalabilidad. Para evitar I/O remoto entre nodos, ambas particiones que se unan deberían tener afinidad al mismo nodo. Todas las parejas de particiones del join deberían repartirse entre todos los nodos para evitar cuellos de botella y usar todos los recursos de CPU disponibles en el sistema.