Oracle 11g Reference Partitioning – Benefits, Hazards & Other Considerations

Andrew V. Zitelli
Thales-Raytheon Systems
zitelli@raytheon.com

NoCOUG Spring Conference 2010
May 20, 2010 – Redwood Shores, California

Regarding Software Complexity

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. *C.A.R. Hoare*

Increasingly, people seem to misinterpret complexity as sophistication, which is baffling -- the incomprehensible should cause suspicion rather than admiration. *Niklaus Wirth*

Who Am I

- Bay Area native & graduate of UC Berkeley.
- Software developer for 35 years.
- Member of the OakTable Network.
- ACM member since 1974.
- Working with Oracle database products since 1992.
  - 10 other relational database products since 1982.
- With Thales-Raytheon Systems, Fullerton, CA since 2001.
  - Joint venture of aerospace firms Thales (France) and Raytheon (USA).
- Primarily working on new development of large multi-tier applications with complex Oracle databases.

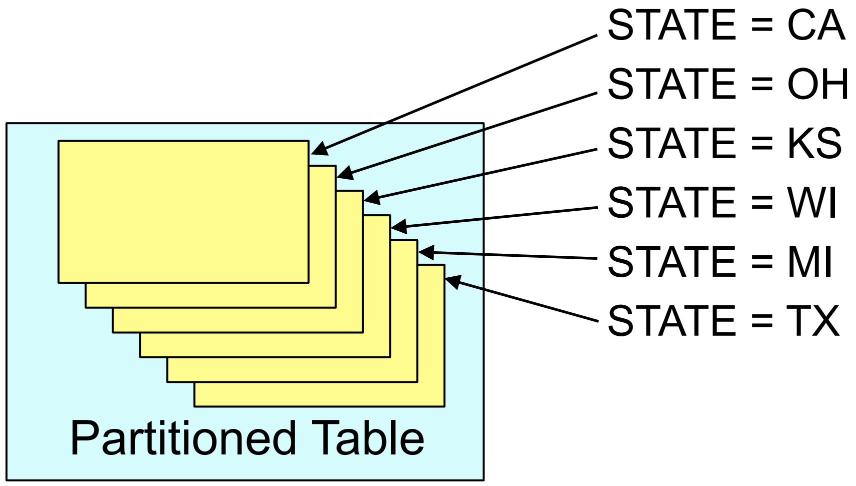
Contents

Introduction

- Oracle 11g introduced a new table partitioning method known as “Reference Partitioning.”
- Reference Partitioning is Oracle’s first partitioning method to support the *partitioning of multiple tables together.*
- In normalized data models, a single logical entity is often represented by multiple tables.
  - These are typically connected within a database using referential (foreign key) constraints.
- The distribution of a logical entity's data across multiple tables raises several problems for single table partitioning methods.
- Reference partitioning alleviates many problems associated with single table partitioning, but adds some new restrictions.

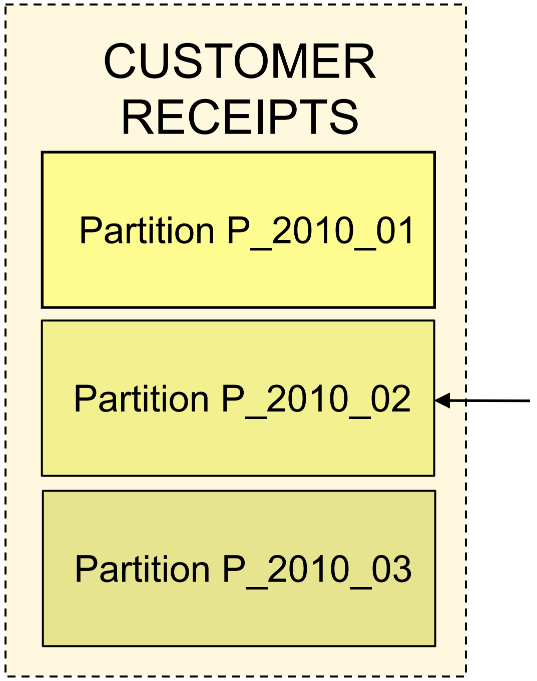
What Is Table Partitioning?

- Table partitioning physically collocates rows of data which share a common characteristic.
  - The common characteristic is know as the *partition key* and is derived from the values of one or more columns in a table.
  - Within a partitioned table, all rows in a given database block will belong to the same partition.



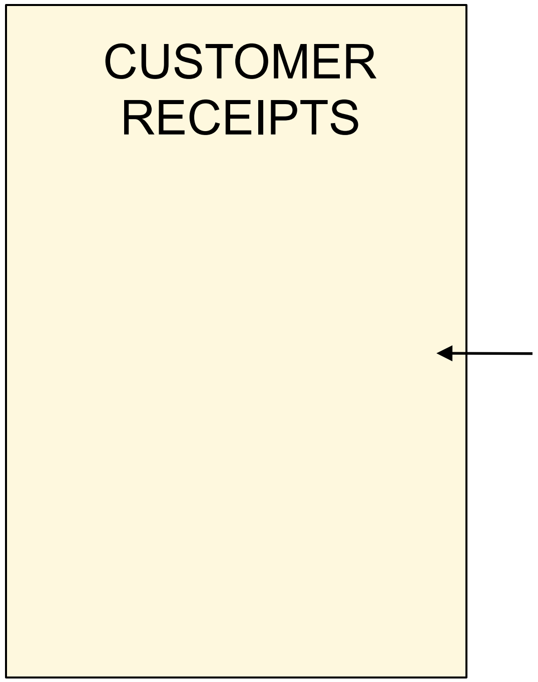
- Partitioning was originally introduced, primarily to simplify the management of large data sets.
- Other commonly cited benefits of partitioning include improved database performance and improved data availability.

Example of Partitioned vs. Non-Partitioned Table



All rows whose transaction_dates fall within the same month, will reside in the same partition.

```
CREATE TABLE CUSTOMER_RECEIPTS (
  RECEIPT_NO           NUMBER NOT NULL,
  TRANSACTION_DATE     DATE   NOT NULL,
  TOTAL_RECEIVED       NUMBER NOT NULL )
  PARTITION BY RANGE (TRANSACTION_DATE) (
    PARTITION P_2010_01 VALUES LESS THAN
      (TO_DATE('2009-02-01', 'YYYY-MM-DD')),
    PARTITION P_2010_02 VALUES LESS THAN
      (TO_DATE('2009-03-01', 'YYYY-MM-DD')),
    PARTITION P_2010_03 VALUES LESS THAN
      (TO_DATE('2009-04-01', 'YYYY-MM-DD')));
```



Rows will be intermingled regardless of their transaction_date values.

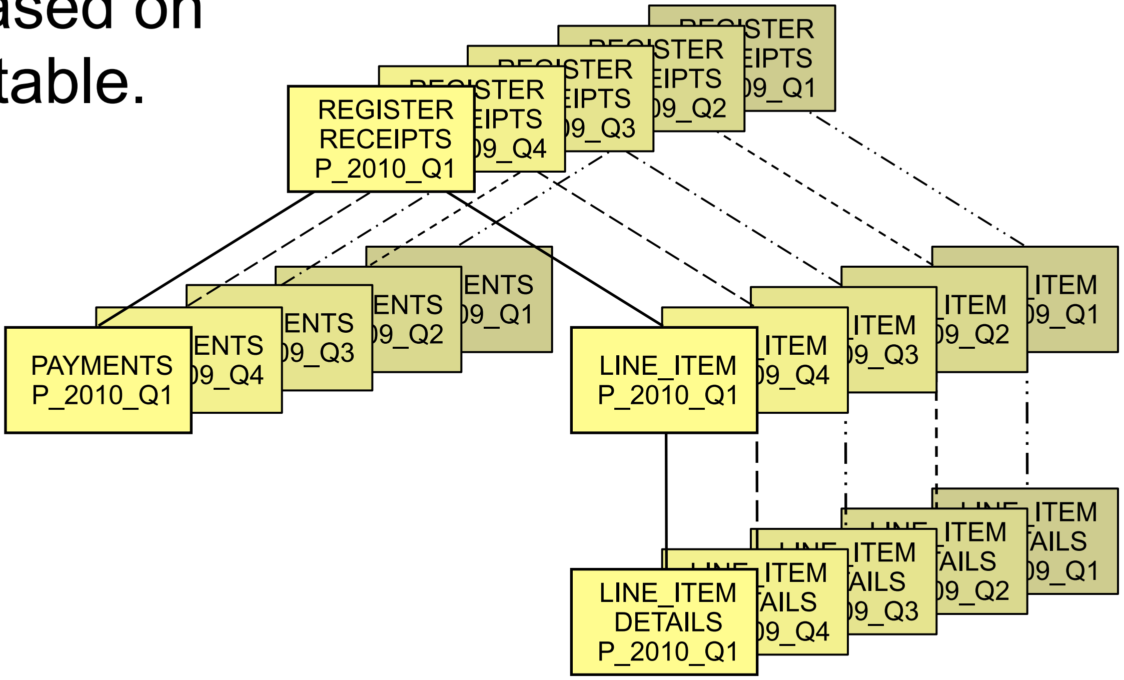
```
CREATE TABLE CUSTOMER_RECEIPTS (
  RECEIPT_NO           NUMBER NOT NULL,
  TRANSACTION_DATE     DATE   NOT NULL,
  TOTAL_RECEIVED       NUMBER NOT NULL );
```

Oracle 11g Partitioning Methods

- Oracle 11g's partitioning methods are Range, List, Hash, System, Interval and Reference partitioning.
  - *Range partitioning* is based on the partition key falling within a specified range of values.
  - *List partitioning* is based on each partition being associated with an explicit list of values.
  - *Hash partitioning* uses a hash function applied to one or more columns, to uniquely determine the partition for each row in a table.
  - *System partitioning* relies on the corresponding SQL INSERT or UPDATE statement to specify the partition to be used for each row.
  - *Interval partitioning* is based on the Range partitioning method, with Oracle automatically creating new partitions using a predefined interval.
  - Oracle also supports *subpartitions* which allow composite partitions based on range, list and hash methods (e.g., range-hash partitions).

What Is Reference Partitioning?

- Reference Partitioning "equi-partitions" two tables, based on a foreign key constraint.
  - Child table rows will be partitioned to match the partitioning of the corresponding parent table rows.
- Reference Partitioning supports hierarchies of tables all partitioned based on a single root table.



Some of Reference Partitioning's Advantages

- It removes the need to individually partition related tables.
  - Developers and administrators are no longer responsible to assure related tables are partitioned the same, with all rows synchronized.
- Partition key columns no longer need to be replicated across multiple tables, solely to support matching partitions.
  - This improves data integrity by eliminating situations in which replicated columns can become out of sync, between related tables.
  - It guarantees that every child table row's corresponding parent row, must reside in the partition matching the child row's partition.
  - Without reference partitioning, it is possible for a child row to have a different partition key value than its parent.
- It simplifies certain data manipulation operations.
  - ADD, DROP, SPLIT and MERGE partition operations are only carried out on the root table, automatically cascading to descendants.

Reference Partitioning Details (1)

- The root for a reference partitioning hierarchy must use List, Range, or Hash partitioning or a sub-partitioned composite.

```
CREATE TABLE PARENT_TAB
  PARENT_ID                NUMBER              NOT NULL,
  PARENT_NAME              VARCHAR2 (30)       NOT NULL,
  LOCAL_TRANSACTION_TIME   TIMESTAMP           NOT NULL,
  CONSTRAINT PK_PARENT PRIMARY KEY (PID))
  PARTITION BY RANGE (LOCAL_TRANSACTION_TIME) (
     PARTITION P_20090206 VALUES LESS THAN (TIMESTAMP' 2009-02-07 00:00:00'),
     PARTITION P_20090207 VALUES LESS THAN (TIMESTAMP' 2009-02-08 00:00:00'),
     PARTITION P_20090209 VALUES LESS THAN (TIMESTAMP' 2009-02-09 00:00:00'))
  ENABLE ROW MOVEMENT;
```

- The DDL for each remaining table in a reference partitioned hierarchy identifies only the table's *partitioning constraint.*

```
CREATE TABLE CHILD_A (
   CHILD_A_ID       NUMBER
   PID              NUMBER          NOT NULL,
   CLASSIFICATION   NUMBER,
   CONSTRAINT PK_CHILD_A PRIMARY KEY (CHILD_ID),
   CONSTRAINT FK_CHILD_A_TO_PARENT FOREIGN KEY (PID) REFERENCES PARENT_TAB (PARENT_ID))
   PARTITION BY REFERENCE (FK_CHILD_A_TO_PARENT)
   ENABLE ROW MOVEMENT;
```

Reference Partitioning Details (2)

- Reference partitioning is specified during table creation.
  - Existing tables cannot be altered later, to add reference partitioning.
- The contents of reference partitioned tables are not intermingled with their parents.
  - Each partition, for each table, uses a separate database segment.
  - Reference partitioned tables will contain a separate partition for each partition and sub-partition in the root table.
- Most reference partition characteristics are inherited from corresponding partitions in the parent table.
  - Partition names, physical and storage characteristics may optionally differ between child and parent table partitions.
  - Oracle 11g’s SQL syntax has been expanded to support these differences between parent and child table partitions.

Reference Partitioning Details (3)

- Reference partitioned tables may reference or be referenced by tables outside the partitioning hierarchy.
  - Data Pump bugs in Oracle 11.1 and 11.2 may interfere with table creation and importing of data when reference partitioned child tables have foreign key constraints other than their partitioning constraints.
- Oracle indexes may also be partitioned but there is no “reference partitioning” for indexes.
- Omission of indexes on partitioning constraints can result in full partition scans during DML operations.
  - This can occur for tables several generations removed from the table being modified.

Reference Partitioning Restrictions (1)

- Parent table Primary Key and Unique constraints used for reference partitioning must be enabled and non-deferrable.

```
ALTER TABLE PARENT_TAB DISABLE CONSTRAINT UK_PARENT
ORA-02297: cannot disable constraint (RHT.UK_PARENT) - dependencies exist
```

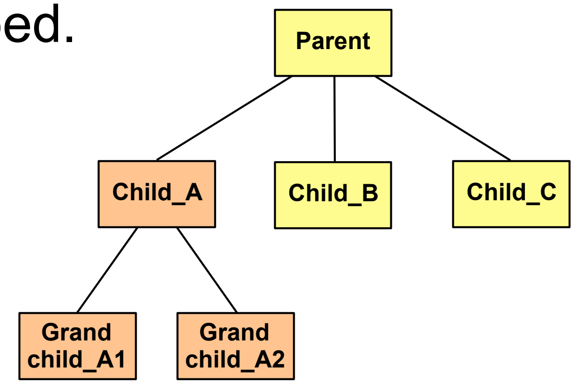
- Child table “partitioning constraints” must be enabled and non-deferrable.

```
ALTER TABLE CHILD_A DISABLE CONSTRAINT FK_CHILD_A_TO_PARENT
ORA-14650: operation not supported for reference-partitioned tables
```

- All child table columns used in “partitioning constraints” must be defined as NOT NULL.
  - This is required to assure that every child row maps to exactly one parent row.

Reference Partitioning Restrictions (2)

- Child tables created using reference partitioning can never be disassociated from their parents without being dropped.
  - There is no ALTER TABLE command to convert a reference partitioned table into a non-reference partitioned table.
- As with other tables referenced via foreign key constraints, parent tables cannot be dropped until all foreign key constraints referencing them are removed.
  - In the case of reference partitioning, this means all descendent tables must first be dropped, since the partitioning constraints on the child tables cannot be disabled or dropped.
  - For example, table CHILD_A cannot be dropped until GRANDCHILD_A1 and GRANDCHILD_A2 are first dropped.

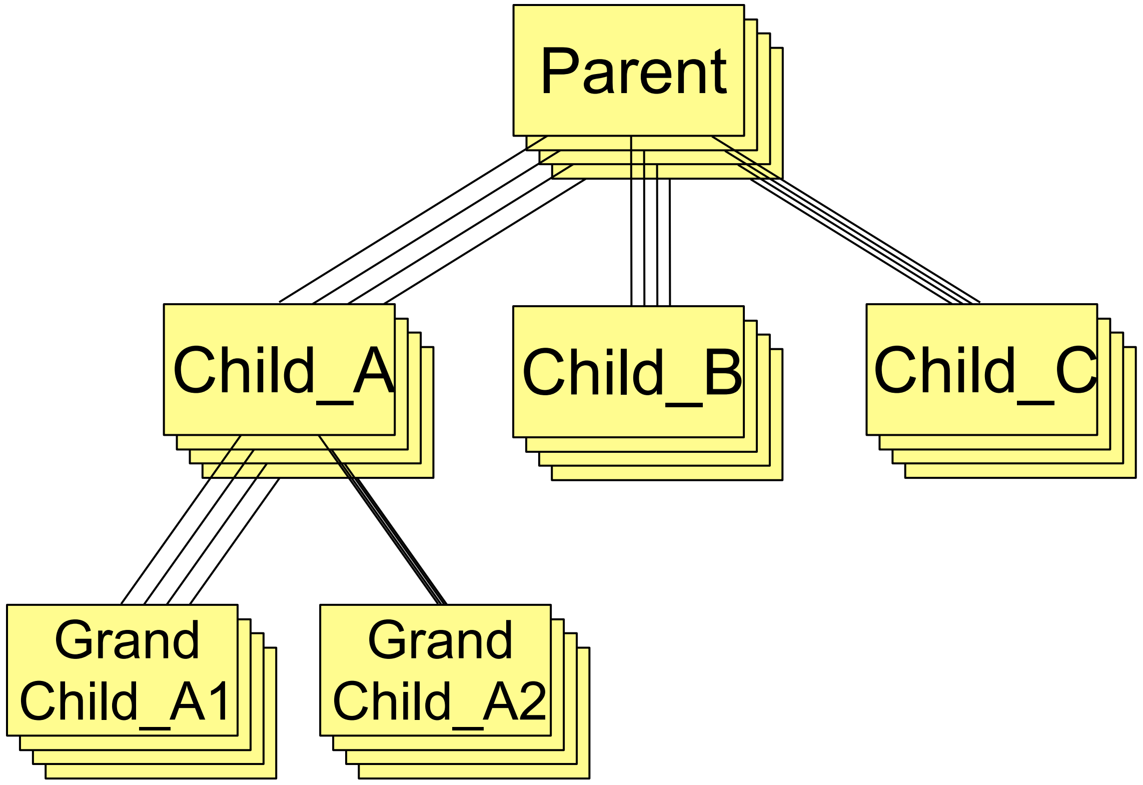


Reference Partitioning Restrictions (3)

- Oracle 11.1 and 11.2 Reference Partitioning does not support use of Oracle 11's Interval partitioning method.
  - That is to say, automatic generation of new partitions is not currently supported.
- You cannot create reference partitioned tables using a CREATE TABLE ... AS SELECT statement.
- These pages highlight several key restrictions but the list is not comprehensive.

Comparison of Data Removal Using Partitioned and non-Partitioned Methods

- In the following examples, partitioned data is range partitioned using the column "LOCAL_TRANSACTION_TIME."
- Under Reference Partitioning, this column is only required in Parent table.



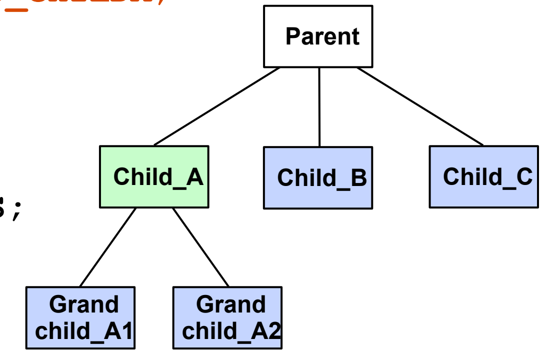
Test Case 1: SQL Needed to Drop the Oldest Partition Using Single Table Partitioning

- SQL used to drop the oldest partition using range partitioning.
  - Foreign key constraints must be disabled for each child table while the corresponding parent table's partition is dropped.
  - The partition key column(s) must be replicated in each table.

```
ALTER TABLE GRANDCHILD_A1 DROP PARTITION P_20090206 UPDATE INDEXES;
ALTER TABLE GRANDCHILD_A2 DROP PARTITION P_20090206 UPDATE INDEXES;
ALTER TABLE CHILD_B       DROP PARTITION P_20090206 UPDATE INDEXES;
ALTER TABLE CHILD_C       DROP PARTITION P_20090206 UPDATE INDEXES;

ALTER TABLE GRANDCHILD_A1 DISABLE CONSTRAINT FK_GCHILDA1_TO_CHILDA;
ALTER TABLE GRANDCHILD_A2 DISABLE CONSTRAINT FK_GCHILDA2_TO_CHILDA;
ALTER TABLE CHILD_A  DROP PARTITION P_20090206 UPDATE INDEXES;
ALTER TABLE GRANDCHILD_A1 ENABLE  CONSTRAINT FK_GCHILDA1_TO_CHILDA;
ALTER TABLE GRANDCHILD_A2 ENABLE  CONSTRAINT FK_GCHILDA2_TO_CHILDA;

ALTER TABLE CHILD_A DISABLE CONSTRAINT FK_CHILDA_TO_PARENT;
ALTER TABLE CHILD_B DISABLE CONSTRAINT FK_CHILDA_TO_PARENT;
ALTER TABLE CHILD_C DISABLE CONSTRAINT FK_CHILDA_TO_PARENT;
ALTER TABLE PARENT DROP PARTITION P_20090206 UPDATE INDEXES;
ALTER TABLE CHILD_A ENABLE  CONSTRAINT FK_CHILDA_TO_PARENT;
ALTER TABLE CHILD_B ENABLE  CONSTRAINT FK_CHILDB_TO_PARENT;
ALTER TABLE CHILD_C ENABLE  CONSTRAINT FK_CHILDC_TO_PARENT;
```



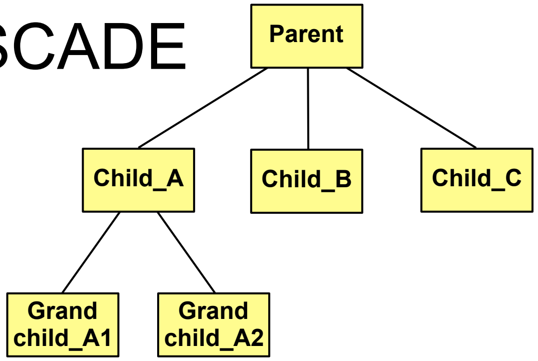
Test Cases 2 & 3: SQL Needed to Delete the Oldest Equivalent Data without any Partitioning

- SQL to delete same data when partitioning is not used and CASCADE DELETE not defined on foreign key constraints.
  - Column(s) matching the partition key must be added to each table.
  - DELETE statements must be issued in the proper order, to assure all child rows are deleted before any corresponding parent rows.

```
DELETE GRANDCHILD_A1 WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
DELETE GRANDCHILD_A2 WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
DELETE CHILD_A WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
DELETE CHILD_B WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
DELETE CHILD_C WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
DELETE PARENT  WHERE LOCAL_TRANSACTION_TIME < TO_DATE ('2009-02-07');
COMMIT;
```

- SQL to delete data when ON DELETE CASCADE is defined for all foreign key constraints.

```
DELETE PARENT WHERE LOCAL_TRANSACTION_TIME
   < TO_DATE ('2009-02-07');
COMMIT;
```



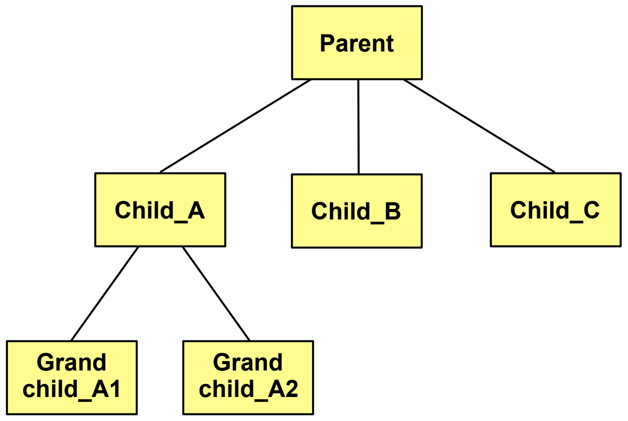
Test Case 4: SQL Needed to Drop the Oldest Partition Using Reference Partitioning

- SQL used to drop the oldest partition using reference partitioning.

```
ALTER TABLE PARENT DROP PARTITION P_20090206 UPDATE INDEXES;
```

- This statement will cause the partition P_20090206 to be dropped from all tables in the reference partitioned hierarchy, as a single operation.

- The UPDATE INDEXES clause will assure that no global indexes are invalidated on any tables. Other database sessions may use all corresponding indexes during the DROP PARTITION operation.



Details regarding Tests as Conducted

Table Name	Num Rows	Avg Row Length*	8K Table Blocks	Indexes	Index Leaf Blocks	Rows Removed
PARENT	15,000,000	377	1,099,302	7	269,664	1,000,000
CHILD_A	13,949,801	61	184,268	5	194,916	930,094
CHILD_B	1,050,199	14	5,293	2	6,571	69,906
CHILD_C	89,999,908	23	555,489	1	391,500	6,000,000
GRAND CHILD_A1	13,949,801	354	976,511	2	106,587	930,094
GRAND CHILD_A2	83,698,714	22	490,473	1	351,238	5,580,564

* Average Row Length, # Table Blocks and # Leaf Blocks reflect details for the Reference Partitioning test case.

Observed Performance Metrics from Tests

Data Removal Method	LIO's†	DB File Reads‡	PIO Blocks Read†	EXEC Calls	Total Read Time‡ (sec)	Total CPU† (sec)	Elapsed Time† (sec)	~ Redo
Reference Partition DROP	1,025,244	79,568	467,210	919	139	223	384	1.2 Gb
Single Table Partition DROP's*	2,189,679	91,437	1,672,248	2,417	220	643	1039	1.0 Gb
Single Table DROP's excluding FK's	1,060,244	78,687	302,379	1906	128	226	386	1.0 Gb
Non-Partitioned DELETE's	100,770,818	398,181	443,780	881	1430	830	2465	20 Gb
Non-Partitioned with Cascaded DELETE's	177,929,197	337,109	337,109	3,860,478	1903	1948	4110	29 Gb

* Single Table Partition Drops included overhead associated with the re-enabling of disabled foreign key constraints.

† LIO's, PIO's, CPU and Elapsed time were measured using Method R Corporation's *mrls* tool (see Appendix B).

‡ DB File Reads and Total Read Time were measured using Method R Corporation's *mrskew* tool (see Appendix B).

Rationale for Excessive Overhead Associated with Cascaded Deletes

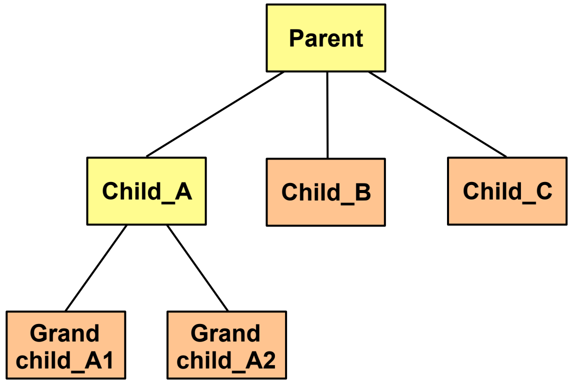
- Oracle uses recursive SQL to execute an explicit DELETE against each child table, where one or more rows exist corresponding to each parent table row being removed.
- DELETE statements are only issued when corresponding child rows exist for a given parent row.

```
PARSING IN CURSOR #11 len=63 dep=1 uid=0 oct=7 lid=0 tim=6898566654873 hv=853327986 ad='6aef09db0' sqlid='18 . . .
 delete from "DATA_REPL"."GRANDCHILD_A1" where "CHILD_A_ID" = :1
END OF STMT
EXEC #11:c=0,e=392,p=0,cr=4,cu=54,mis=0,r=6,dep=1,og=4,plh=712978407,tim=6898566654873
CLOSE #11:c=0,e=2,dep=1,type=3,tim=6898566654978
=====================
PARSING IN CURSOR #2 len=83 dep=3 uid=0 oct=7 lid=0 tim=6898566655727 hv=4202949588 ad='68e3f8570' sqlid='ca . . .
 delete from "DATA_REPL"."GRANDCHILD_A2" where "CHILD_A_ID" = :1
END OF STMT
EXEC #2:c=0,e=353,p=0,cr=4,cu=54,mis=0,r=6,dep=3,og=4,plh=3674908788,tim=6898566655727
CLOSE #2:c=0,e=3,dep=3,type=3,tim=6898566655898
=====================
PARSING IN CURSOR #6 len=78 dep=2 uid=0 oct=7 lid=0 tim=6898566655925 hv=3961620125 ad='67e134920' sqlid='g3 . . .
 delete from "DATA_REPL"."CHILD_A" where "PID" = :1
END OF STMT
EXEC #6:c=0,e=700,p=0,cr=7,cu=68,mis=0,r=1,dep=2,og=4,plh=1029372813,tim=6898566655925
CLOSE #6:c=0,e=2,dep=2,type=3,tim=6898566656023
=====================
PARSING IN CURSOR #7 len=65 dep=1 uid=0 oct=7 lid=0 tim=6898566656050 hv=3569125061 ad='67e137500' sqlid='b7 . . .
 delete from "DATA_REPL"."CHILD_C" where "PID" = :1
END OF STMT
EXEC #7:c=0,e=1030,p=0,cr=10,cu=89,mis=0,r=1,dep=1,og=4,plh=902635858,tim=6898566656049
```

Table and Partition Truncation under Reference Partitioning (1)

- Leaf tables in a reference partitioned hierarchy may be truncated using TRUNCATE TABLE.

```
TRUNCATE TABLE GRANDCHILD_A1;
TRUNCATE TABLE GRANDCHILD_A2;
TRUNCATE TABLE CHILD_B;
TRUNCATE TABLE CHILD_C;
```



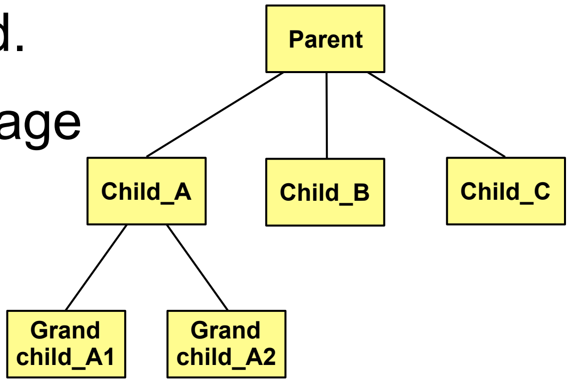
- No parent tables may be truncated using TRUNCATE TABLE.
  - TRUNCATE TABLE cannot be used when enabled foreign key constraints reference a table.
  - Partitioning constraints can never be disabled.
- Parent tables can be truncated by individually truncating each partition using ALTER TABLE .. TRUNCATE PARTITION.
- All descendent partitions of a given partition must be empty before the partition can be truncated.

Table and Partition Truncation under Reference Partitioning (2)

- An individual partition can be truncated throughout the hierarchy by individually truncating it in each table.
  - All descendants of a given partition must be empty before the partition can be truncated so ordering of the TRUNCATE commands is significant.
- Example of SQL to truncate a partition throughout a hierarchy.

```
ALTER TABLE GRANDCHILD_A1 TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
ALTER TABLE GRANDCHILD_A2 TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
ALTER TABLE CHILD_A TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
ALTER TABLE CHILD_B TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
ALTER TABLE CHILD_C TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
ALTER TABLE PARENT TRUNCATE PARTITION P_20090208 DROP STORAGE UPDATE INDEXES;
```

- No cascaded TRUNCATE PARTITION is supported.
- DROP STORAGE clause must be specified or storage will be retained for future reuse by the partition.
- UPDATE INDEXES clause must be used or all global indexes will be invalidated.



Test Case 2: Comparison of Drop Partition vs. Truncate Partition when Using Reference Partitioning

Data Removal Method	LIO's	DB File Reads	PIO Blocks Read	EXEC Calls	Total Read Time (sec)	Total CPU (sec)	Elapsed Time (sec)	~ Redo
Reference Partition DROP	1,025,244	79,568	467,210	919	139	223	384	1.2 Gb
Reference Partition TRUNCATE	1,679,726	109,705	687,358	1579	126	251	421	1.2 Gb

- Partition truncate tests were based on the same table definitions and data as the earlier DROP PARTITION tests.
- DROP STORAGE and UPDATE INDEXES clauses were used.

Further Partition Truncation Details (1)

- Truncation of a given partition across all tables in a reference partition hierarchy can be automated using PL/SQL.
  - Child table partitions must always be truncated prior to corresponding parent table partitions.
  - Code must simply iterate through the partition hierarchy one generation at a time.
- Referential constraints, other than partitioning constraints, which reference tables in the partitioning hierarchy, must be disabled before parent partitions can be truncated.
  - Since referential (foreign key) constraints are disabled for an entire table, the subsequent re-enabling of referential constraints for large tables can incur a large amount of overhead.

Further Partition Truncation Details (2)

- The column REF_PTN_CONSTRAINT_NAME has been added to DBA_PART_TABLES, listing the name of the partitioning constraint for reference partitioned (child) tables.
  - This allows the Oracle data dictionary to be used to navigate through a reference partitioned hierarchy.
  - The following query provides an example for retrieving the table names and parent table names for all reference partitioned tables owned by the current user. This query excludes the root table(s).

```
SELECT PTAB.TABLE_NAME "PARENT_TABLE", DPT.TABLE_NAME "CHILD_TABLE"
   FROM DBA_CONSTRAINTS PTAB, DBA_CONSTRAINTS DC, DBA_PART_TABLES DPT
   WHERE DC.OWNER = USER AND DPT.OWNER = DC.OWNER AND
      PTAB.OWNER = DC.OWNER AND
      DC.CONSTRAINT_NAME = DPT.REF_PTN_CONSTRAINT_NAME AND
      PTAB.CONSTRAINT_NAME = DC.R_CONSTRAINT_NAME
      ORDER BY PARENT_TABLE, CHILD_TABLE;
```

Row Migration Considerations (1)

- Partitioned tables can be configured to enable or disable the migration of rows between partitions.
  - Rows may migrate when partition keys change.
  - For child tables using reference partitioning, child rows will normally migrate to a new partition when the parent rows migrate.
  - Child rows may also migrate if their partitioning constraint columns are modified to reference new parent rows.
- When reference partitions are used, all child tables (and subsequent generations) of a parent table with migration enabled, must also have migration enabled.
  - CREATE TABLE … ENABLE ROW MOVEMENT is used for this.
  - If row movement is not enabled as required, ORA-14661 will be raised during the child table's creation:

```
ORA-14661: row movement must be enabled
```

Row Migration Considerations (2)

- Migration of large numbers of rows within a reference partitioning hierarchy can incur a large amount of overhead.
- Oracle uses recursive SQL to execute an explicit UPDATE statement against each descendent row corresponding to every parent table row which migrates.
- Based on the same data as earlier tests, row migration of 265,000 root table rows caused the movement of 1.27 million descendant rows, motivated the execution of an individual UPDATE command for each row:

```
update "DATA_REPL"."CHILD_A" partition (dataobj_to_partition
    ("DATA_REPL"."PARENT", :1))
    move to partition (dataobj_to_partition("DATA_REPL"."PARENT", :1))
    set "PID"="PID" where "PID" =:1
```

	LIO's	DB File Reads	PIO Blocks Read	EXEC Calls	Read Time (sec)	Total CPU (sec)	Elapsed Time (sec)
Row Migration Metrics	82,603,529	17,389	50,202	1,270,408	129	883	1134

Row Migration Considerations (3)

- Reference partitioned children may have row migration enabled even if the parent has row migration disabled.
  - This is permitted to allow child rows to move from one parent row to another, if their partitioning constraint's value changes.
- With row migration disabled, if a child row's partitioning constraint value is changed, the new parent row must reside in same partition as the original parent row.

```
ORA-14402: updating partition key column would cause a partition change
```

- Values changed in parent table columns referenced by partitioning constraints, must never orphan any child rows.

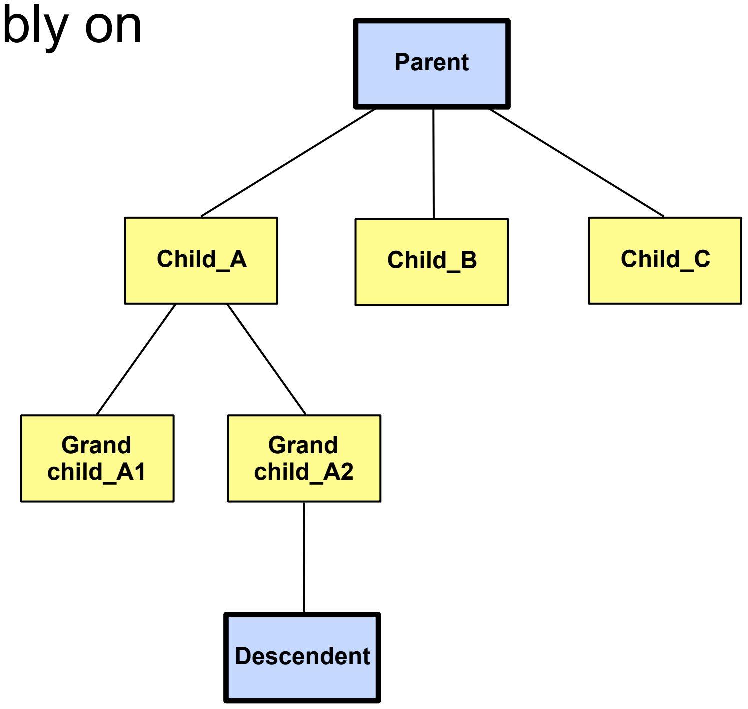
```
ORA-02292: integrity constraint (PCOAD.FK_LNI) violated - child record found
```

Row Migration Considerations (4)

- Changes to parent rows may cause child rows to migrate without causing migration in the parent table.
  - This can occur when values are modified in the columns of the parent constraint which are referenced by a child table's partitioning constraint.
  - Again, child table rows are never allowed to be orphaned.
- Row movement is disallowed when a partitioning constraint references a parent table constraint which is enforced by a non-unique index.
  - See ORA-14657 description for details.
- Row movement can be enabled/disabled after table creation.
  - Parent table row movement must be disabled before children.
  - Likewise, child table row movement must be enabled before parents.

Row Migration – Hidden Deadlock Risks (1)

- Deadlocks occur in Oracle when two (or more) sessions simultaneously hold locks which other deadlocked sessions are waiting to acquire.
- Every deadlocked session blocks another deadlocked session, while itself waiting to acquire a lock held by another deadlocked session.
- Row migration within reference partitioning trees incurs a risk of deadlock from seemingly unrelated actions, possibly on tables several generations apart.
- Partition migration of a parent row must acquire locks on all descendent rows being moved, throughout the partitioning tree.
- The example on the following page assumes PARENT table id = 1 corresponds to DESCENDENT id = 1 and PARENT id = 2 corresponds to DESCENDENT id = 2.



Row Migration – Hidden Deadlock Risks (2)

Session #1

```
-- Due to row movement, this UPDATE locks
-- the PARENT row with PARENT_ID = 1 and all
-- corresponding child rows.
```

```
UPDATE PARENT SET TRANSACTION_TIME =
TRANSACTION_TIME + 30 WHERE PARENT_ID = 1;
```

```
-- Due to row movement, this UPDATE attempts
-- to lock rows related to PARENT_ID = 2. It
-- is blocked by Session #2 from locking the
-- child row with DESC_ID=2.
```

```
UPDATE PARENT SET TRANSACTION_TIME =
TRANSACTION_TIME - 25 WHERE PARENT_ID = 2;
```

```
-- Upon deadlock detection, Oracle responds
-- by raising error ORA-00060 on Session #1.
```

```
UPDATE PARENT SET TRANSACTION_TIME =
   TRANSACTION_TIME - 25 WHERE PARENT_ID = 2;
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting
   for resource
```

Session #2

```
-- This UPDATE command locks the DESCENDENT
-- table row with DESC_ID = 2.
```

```
UPDATE DESCENDENT SET POSTAL_CODE = 92626 WHERE
DESC_ID = 2;
```

```
-- This UPDATE command attempts to lock the
-- DESCENDENT row with DESC_ID = 1, thereby
-- resulting in a deadlock.
```

```
UPDATE DESCENDENT SET NAME = 'THOMPSON' WHERE
DESC_ID = 1;
```

Row Migration – Hidden Deadlock Risks (3)

- ORA-00060 exceptions only roll back the listed SQL statement, not an entire transaction. Transactions continue to block until one transaction is rolled back.
- If row movement is enabled while using reference partitions, applications must include necessary logic to rollback transactions which receive an ORA-00060 error.

Session #1

Time

Session #2

```
UPDATE PARENT SET TRANSACTION_TIME =
   TRANSACTION_TIME - 25 WHERE PARENT_ID = 2;
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting
   for resource

-- If Session #1 attempts to re-execute
-- the failed query, a deadlock reoccurs.

UPDATE PARENT SET TRANSACTION_TIME =
TRANSACTION_TIME - 25 WHERE PARENT_ID = 2;
```

```
UPDATE DESCENDENT SET NAME = 'THOMPSON' WHERE
DESC_ID = 1;
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting
   for resource
```

Local Index Considerations (1)

- **When using reference partitioning, most child table indexes should be defined as *global,* unless there is a compelling reason for a given index to be defined as *local.***
- *Local* indexes are partitioned indexes whose partitions match their corresponding table partitions.
  - Local index entries always reside in the index partition matching the corresponding row's table partition.
  - If the root table's partition key columns are not replicated in a child table, defining an index on the child table as *local* means the index will be partitioned on columns excluded from the child table.
- Local indexes can only be created as UNIQUE, if the partitioning columns form a subset of the index columns.
  - This guarantees that rows with identical index keys always map to the same partition.

Local Index Considerations (2)

- **If a parent's partition key columns are replicated in a child table, but are not contained in the child's partitioning constraint, Oracle cannot infer the "matching columns" represent the same information.**
- Consider the tables and indexes on the following page, used for subsequent examples.
  - Function based indexes using constants are used to create both local and global indexes on the DATE_OF_BIRTH column.

    ```
    CREATE INDEX I#PARENT#DOB_GLOBAL ON PARENT (DATE_OF_BIRTH, 'G');
    ```

  - As an aside, this technique can be used to force "single column" indexes to include rows with NULL values since the supplied constant is always non-null.
    - It has been reported that the numeric value 0 may not always work properly when used as a constant value in an index.
- Test data includes exactly one parent and one child row for July 31, 1977.
- PARENT and CHILD tables and local indexes each contain 50 partitions.

Local Index Considerations (3)

```
CREATE TABLE PARENT (
   PARENT_ID                NUMBER         NOT NULL,
   PARENT_NAME              VARCHAR2 (40)  NOT NULL,
   DATE_OF_BIRTH            TIMESTAMP      NOT NULL,
   CONSTRAINT PK_PARENT PRIMARY KEY (PARENT_ID))
   PARTITION BY RANGE (DATE_OF_BIRTH) (
      PARTITION P_1951 VALUES LESS THAN (TIMESTAMP' 1952-01-01 00:00:00'),
         . . .
      PARTITION P_2000 VALUES LESS THAN (TIMESTAMP' 2001-01-01 00:00:00'));

CREATE INDEX I#PARENT#DOB_LOCAL  ON PARENT (DATE_OF_BIRTH, 'L') LOCAL;
CREATE INDEX I#PARENT#DOB_GLOBAL ON PARENT (DATE_OF_BIRTH, 'G');

CREATE TABLE CHILD (
   CHILD_ID                NUMBER         NOT NULL,
   PID                     NUMBER         NOT NULL,
   CHILD_NAME              VARCHAR2 (40) NOT NULL,
   DATE_OF_BIRTH           TIMESTAMP      NOT NULL,
   CONSTRAINT PK_CHILD PRIMARY KEY (CHILD_ID),
   CONSTRAINT FK_CHILD_TO_PAR FOREIGN KEY (PID) REFERENCES PARENT (PARENT_ID) )
      PARTITION BY REFERENCE (FK_CHILD_TO_PAR);

CREATE INDEX I#CHILD#FK ON CHILD (PID);
CREATE INDEX I#CHILD#DOB_LOCAL  ON CHILD (DATE_OF_BIRTH, 'L') LOCAL;
CREATE INDEX I#CHILD#DOB_GLOBAL ON CHILD (DATE_OF_BIRTH, 'G');
```

Partitioning constraint column

Local Index Considerations (4)

Parent Table Index Details

- In the first example, Oracle uses the local index on the PARENT table, performing 4 “consistent gets” (db block accesses consistent with a given point in time or SCN).

```
SQL> select * from parent where date_of_birth = '31-JUL-77';
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name                |Rows|Bytes|Cost (%CPU)| Time      | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                     |  1 |  21 |    3   (0)| 00:00:01 |       |       |
|   1 |  PARTITION RANGE SINGLE            |                     |  1 |  21 |    3   (0)| 00:00:01 |   KEY |   KEY |
|   2 |   TABLE ACCESS BY LOCAL INDEX ROWID| PARENT              |  1 |  21 |    3   (0)| 00:00:01 |   KEY |   KEY |
|*  3 |    INDEX RANGE SCAN                | I#PARENT#DOB_LOCAL  |  1 |     |    2   (0)| 00:00:01 |   KEY |   KEY |
-----------------------------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          4  consistent gets
```

- When an INDEX hint is used forcing Oracle to use the global index, the query execution performs 5 consistent gets.

```
SQL> select /*+ index (parent I#PARENT#DOB_GLOBAL) */  * from parent where date_of_birth = '31-JUL-77';
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name                 |Rows|Bytes|Cost (%CPU)| Time      | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                      |  1 |  21 |    4   (0)| 00:00:01 |       |       |
|   1 |  TABLE ACCESS BY GLOBAL INDEX ROWID| PARENT               |  1 |  21 |    4   (0)| 00:00:01 | ROWID | ROWID |
|*  2 |   INDEX RANGE SCAN                 | I#PARENT#DOB_GLOBAL  |  1 |     |    3   (0)| 00:00:01 |       |       |
-------------------------------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          5  consistent gets
```

Local Index Considerations (5)
Child Table Index Details

- When a similar query is executed on the CHILD table, Oracle chooses to use the global index, performing 5 consistent gets.

```
SQL> select * from child where date_of_birth = '31-JUL-77';
---------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name               |Rows|Bytes|Cost (%CPU)| Time     | Pstart| Pstop |
---------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                    |  1 |  30 |    4   (0)| 00:00:01 |       |       |
|   1 |  TABLE ACCESS BY GLOBAL INDEX ROWID| CHILD              |  1 |  30 |    4   (0)| 00:00:01 | ROWID | ROWID |
|*  2 |   INDEX RANGE SCAN                 | I#CHILD#DOB_GLOBAL |  1 |     |    3   (0)| 00:00:01 |       |       |
---------------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          5  consistent gets
```

- When a hint is used forcing Oracle to use the local index, the execution plan shows all 50 index partitions must be probed, performing *102* consistent gets.

```
SQL> select /*+ index (child I#CHILD#DOB_GLOBAL) */  * from child where date_of_birth = '31-JUL-77';
--------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name              |Rows|Bytes|Cost (%CPU)| Time     | Pstart| Pstop |
--------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |  1 |  30 |   52   (0)| 00:00:01 |       |       |
|   1 |  PARTITION REFERENCE ALL           |                   |  1 |  30 |   52   (0)| 00:00:01 |     1 |    50 |
|   2 |   TABLE ACCESS BY LOCAL INDEX ROWID| CHILD             |  1 |  30 |   52   (0)| 00:00:01 |     1 |    50 |
|*  3 |    INDEX RANGE SCAN                | I#CHILD#DOB_LOCAL |  1 |     |   51   (0)| 00:00:01 |     1 |    50 |
--------------------------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
        102  consistent gets
```

Local Index Considerations (6)

- A local index on a reference partitioned child table may be used if the query explicitly specifies the partition, in this example performing 5 consistent gets.

```
SQL> select * from child partition (P_1977) where date_of_birth = '31-JUL-77';
----------------------------------------------------------------------------------------------
| Id  | Operation                          | Name              |Rows|Bytes|Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                   |  2 |  60 |    5   (0)| 00:00:01 |       |       |
|   1 |  PARTITION REFERENCE SINGLE        |                   |  2 |  60 |    5   (0)| 00:00:01 |    27 |    27 |
|   2 |   TABLE ACCESS BY LOCAL INDEX ROWID| CHILD             |  2 |  60 |    5   (0)| 00:00:01 |    27 |    27 |
|*  3 |    INDEX RANGE SCAN                | I#CHILD#DOB_LOCAL |  1 |     |    3   (0)| 00:00:01 |    27 |    27 |
----------------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
        5  consistent gets
```

- Oracle's nomenclature regarding "prefixed" local indexes is misleading with regard to reference partitioned child tables.
- Local indexes using partitioning constraint column(s) as leading columns display as prefixed, even though partitioning is based on column(s) excluded from the table.

```
CREATE INDEX I#CHILD#FK_LOCAL ON CHILD (PID) LOCAL;
SQL> select table_name, index_name, partitioning_type, alignment from user_part_indexes;
TABLE_NAME                 INDEX_NAME                 PARTITIONING_TYPE ALIGNMENT
-------------------------- -------------------------- ----------------- ------------
CHILD                      I#CHILD#DOB_LOCAL          UNKNOWN           NON_PREFIXED
CHILD                      I#CHILD#FK                 UNKNOWN           PREFIXED
PARENT                     I#PARENT#DOB_LOCAL         RANGE             PREFIXED
```

Partition-Wise Joins (1)

- Where applicable, Oracle attempts to use reference partitioned tables to improve query performance.
- For example, if a child table is joined to its parent, using the columns in its partitioning constraint, Oracle can always perform *partition-wise joins.*
  - In these cases, Oracle will only attempt joins between rows in matching partitions.
  - This optimization can extend through multiple generations, if each reference partitioned table in a query, is joined to its parent using the partitioning constraint's columns.
  - In these cases, partitions pruned in the parent table will also be pruned in the child tables.

Partition-Wise Joins (2)

- Consider the following reference partitioned tables:

```
CREATE TABLE PARENT
   PARENT_ID                    NUMBER        NOT NULL,
   PARENT_NAME                  VARCHAR2 (30) NOT NULL,
   DATE_OF_BIRTH                TIMESTAMP     NOT NULL,
   CONSTRAINT PK_PARENT PRIMARY KEY (PID))
   PARTITION BY RANGE (DATE_OF_BIRTH) (
      PARTITION P_1901 VALUES LESS THAN (TIMESTAMP' 1902-01-01 00:00:00'),
        . . .
      PARTITION P_2000 VALUES LESS THAN (TIMESTAMP' 2001-01-01 00:00:00'));

CREATE TABLE CHILD (
   CHILD_ID                    NUMBER        NOT NULL,
   PID                         NUMBER        NOT NULL,
   CHILD_NAME                  VARCHAR2 (30) NOT NULL,
   CONSTRAINT FK_CHILD_TO_PAR FOREIGN KEY (PID) REFERENCES PARENT (PARENT_ID))
   PARTITION BY REFERENCE (FK_CHILD_TO_PAR);

CREATE TABLE GRANDCHILD (
   GRANDCHILD_ID               NUMBER        NOT NULL,
   CID                         NUMBER        NOT NULL,
   GCHILD_NAME                 VARCHAR2 (30) NOT NULL,
   CONSTRAINT FK_GCHILD_TO_CHILD FOREIGN KEY (CID) REFERENCES CHILD (CHILD_ID))
   PARTITION BY REFERENCE (FK_GCHILD_TO_CHILD);
```

Partition-Wise Joins (3)

```
SELECT PARENT_NAME, CHILD_NAME, GCHILD_NAME
   FROM PARENT P, CHILD C, GRANDCHILD G
   WHERE
      P.DATE_OF_BIRTH = TO_DATE ('1955-01-01','YYYY-MM-DD') AND
      P.PARENT_ID = C.PID AND
      C.CHILD_ID = GC.CID;
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                 | Name        | Rows  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |             |     5 |   630 |     2   (0)| 00:00:01 |       |       |
|   1 |  PARTITION RANGE SINGLE|             |     5 |   630 |     2   (0)| 00:00:01 |     5 |     5 |
|   2 |   NESTED LOOPS            |             |     5 |   630 |     2   (0)| 00:00:01 |       |       |
|   3 |    NESTED LOOPS           |             |     3 |   288 |     2   (0)| 00:00:01 |       |       |
|*  4 |     TABLE ACCESS FULL     | PARENT      |     1 |    48 |     2   (0)| 00:00:01 |     5 |     5 |
|*  5 |     TABLE ACCESS FULL     | CHILD       |     3 |   144 |     0   (0)| 00:00:01 |     5 |     5 |
|*  6 |    TABLE ACCESS FULL      | GRANDCHILD  |     2 |    60 |     0   (0)| 00:00:01 |     5 |     5 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("P"."DATE_OF_BIRTH"=TIMESTAMP' 1955-07-01 00:00:00')
   5 - filter("P"."PARENT_ID"="C"."PID")
   6 - filter("C"."CHILD_ID"="GC"."CID")
```

- Without supporting indexes, Oracle was still able to use a single partition in each of the three tables, to satisfy this query.

Partition Merges, Splits and Exchanges

- Reference partitioning can significantly reduce performance during partition merges, splits and exchanges.
  - The cascading of row migration across descendent tables is one of the key factors.
  - Eadon [2008] cites tests showing that range and hash partitioned *splits* as well as composite partitioned *merges* can incur a particularly high performance penalty.
  - Merges and splits which do not require row migration do not incur a performance penalty.
- During partition exchanges, every row in the partition being exchanged into a reference partitioned table must already have matching parent rows.
- Testing is needed, to see how each of these operations performs in a given environment.

Summary (1)

- Reference Partitioning's primary advantages appear to be:
  - Simplification of data management when dealing with large volumes of data having a shared life-cycle.
  - Simplification of application code when partitioning multiple tables together, all sharing the same partitioning key.
    - Partition keys no longer need to be replicated across multiple tables.
    - Application code no longer needs to keep partition keys synchronized.
  - New query optimizations are supported when large tables are joined, but are not joined on the each table's partition key.
    - This specifically applies to joins between parent and child tables, based on the child's partitioning constraint.
    - Eadon [2008] provides details regarding many of these optimizations.

Summary (2)

- Reference Partitioning adds new restrictions and limitations which may make it unsuitable in many environments.
  - Omission of indexes supporting partitioning constraints can severely degrade UPDATE performance across multiple tables in a hierarchy.
    - It can also increase the risk of deadlocks involving table level locks.
  - The inability to defer partitioning constraints and their parent table constraints may interfere with application design.
  - Row migration of large numbers of rows between partitions may incur excessive overhead.
  - Risk of row migration induced deadlocks will increase the complexity of application development and maintenance.
  - The limited availability to use local indexes may adversely affect some query’s performance when compared to single table partitioning.
  - The inability to drop tables without first dropping all descendent tables limits the reorganization of partitioned hierarchies to repair mistakes.

Conclusions

- Reference Partitioning has strong advantages and strong limitations, so its use requires careful planning.
  - Reference Partitioning should not simply be used wherever possible.
- Use of Reference Partitioning appears best suited for very large tables and related tables, whose contents share the same life cycle.
- If applications can tolerate it, the disabling of row movement on reference partitioned tables should be considered, to reduce the risk of deadlocks and the substantial overhead incurred by row movement.
  - All code accessing tables susceptible to reference partition row movement must properly handle ORA-00060 deadlock exceptions, rolling back the corresponding transactions.
- All child table partitioning constraints should be supported by indexes.
- All applicable Oracle patches should be applied. Where available, Oracle 11.2.0.1.0 or later should be used. See Appendix B.
- Reference Partitioning should be carefully tested before use.

Q & A

References

Eadon, G., et al. 2008. *Supporting Table Partitioning By Reference in Oracle.* Vancouver, BC. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pages 1111–1122. ISBN 978-1-60558-103-3.

Millsap, C.; Holt, J. 2003. *Optimizing Oracle Performance.* Sebastopol CA: O'Reilly. ISBN 059600527X.

Millsap, C. 2009. *MR Tools Documentation.* Method-R Corporation.
http://method-r.com/component/content/article/61-documentation/124-mrlsdoc
http://method-r.com/component/content/article/61-documentation/125-mrnldoc
http://method-r.com/component/content/article/61-documentation/126-mrskewdoc

Oracle Corporation. 2010. *Interpreting Raw SQL_TRACE and DBMS_SUPPORT.START_TRACE Output - Note 39817.1.*

Oracle Corporation. 2010. *Oracle Database VLDB and Partitioning Guide 11g Release 2 (11.2).* Part number E10837-04.

Poder, T., 2010. *Session Snapper Documentation.* E2SN Company.
http://tech.e2sn.com/oracle-scripts-and-tools/session-snapper

Appendix A:
Oracle 11g
Recommended Patches

Appendix A: Recommended Patches for Reference Partitioning

- If possible Oracle 11.2.0.1.0 or later should be used whenever reference partitioning is used.
  – Patch #8477142 (described below) is the only patch related to reference partitioning which I know about, required for 11.2.0.1.0.
- If 11.1.x is used, the following patches are recommended:
  – 7722575 Excessively slow Data Pump export "Estimate" step for partitioned tables. Only patched on Linux.
    - Bug #8845859 for Solaris has no patch and is not fixed until 12.1. A potential workaround is to specify VERSION=10.2.0.3 during export.

  – 7654925 Fixes 11.1 problems with very slow inserts to reference partitioned child tables. Fixed in 11.2.

  – 8477142 Fixes widespread 11.1 and 11.2 Data Pump import errors related to reference partitioned tables. Fixed in 12.1.

  – 9364608 Solaris only; Other patch numbers apply on other platforms. Very slow statistics collection for partitioned tables. Fixed in 11.2.

Appendix A: Recommended Patches for 10046 SQL Tracing

- Oracle 11.1.0.7.0 introduced 2 bugs severely corrupting the tim=, ela= and e= timing values inside Oracle trace files.
  - Patch 7522002 fixes a generic bug on all platforms for 11.1.0.7.0 & 7.1. Later versions of 11.1.0.7.x may remove this patch and resume erroneous behavior.
  - Patch 8342329 fixes a second bug on HP/UX, Solaris and AIX. This patch is only available on 11.1.0.7.0. On Solaris only, patch 9415425 has been released for 11.1.0.7.1, merging patches 7522002 and 8342329.
- Oracle 11.2 fixes both bugs.
  - 11.2 now uses microseconds for timer increments on all platforms. Prior releases from 9.0 through 11.1 used nanoseconds/1024 on Solaris, AIX, HP/UX and some Linux versions.

Appendix B:
Tools used for 10046 Trace File Analysis

Appendix B:
Tools Used for Trace File Analysis

- All tests in this presentation were conducted using Oracle 11.1.0.7.0, while collecting Oracle's 10046 SQL Trace Data.
- The primary tools I use for 10046 trace file analysis are:
  - MR Tools (*mrls*, *mrskew* and *mrnl*) from Method-R Corporation
    For details please see: *http://www.method-r.com/software/mrtools*
  - Method-R Profiler (also sold as the Hotsos Profiler).
    For details please see: *http://www.method-r.com/software/profiler*
  - Personal tools I have written in PL/SQL and Perl.
  - A text editor capable of quickly opening and manipulating very large files. I prefer TextPad under MS Windows & BBEdit on the Macintosh.
  - I also strongly recommend Tanel Poder's *Session Snapper,*
    not for 10046 analysis but as a powerful PL/SQL tool supporting fast, flexible first round performance troubleshooting.
    http://tech.e2sn.com/oracle-scripts-and-tools/session-snapper

Appendix B: MR Tools – *mrskew* Example

- This command line tool allows one to extract a wide variety of summarized details regarding the contents and skew within Oracle 10046 trace files.
  - Task: Determine the average retrieval time from disk, which an Oracle session is experiencing, during the time period covered by its trace file.
  - Details are found in "db file scattered read" & "db file sequential read" events.

```
WAIT #3: nam='db file scattered read' ela=9946 file#=80 block#=72659 blocks=2 obj#=25939 tim=672349123
WAIT #3: nam='db file sequential read' ela=267 file#=80 block#=72624 blocks=1 obj#=25939 tim=672929350
```

```
$ mrskew --name='db.*read' CRIEFF_ora_17077.trc

Matched event names:
   db file scattered read
   db file sequential read
RANGE {min <= e < max}          DURATION       CALLS         MEAN          MIN          MAX
  0.000000     0.000001     0.000000    0.0%          0
  0.000001     0.000010     0.000000    0.0%          0
  0.000010     0.000100     0.000000    0.0%          0
  0.000100     0.001000     0.356379    8.1%       1014     0.000351     0.000103     0.000973
  0.001000     0.010000     2.700134   61.7%        469     0.005757     0.001000     0.009983
  0.010000     0.100000     1.189528   27.2%         79     0.015057     0.010004     0.068535
  0.100000     1.000000     0.130391    3.0%          1     0.130391     0.130391     0.130391
  1.000000    10.000000     0.000000    0.0%          0
 10.000000   100.000000     0.000000    0.0%          0
100.000000  1000.000000     0.000000    0.0%          0
1000.000000     Infinity    0.000000    0.0%          0
            TOTAL (4)       4.376432  100.0%       1563     0.002800     0.000103     0.130391
```

Answer = 2.8 milliseconds