Supporting Table Partitioning By Reference in Oracle

George Eadon
Ananth Raghavan[+]
Eugene Inseok Chong
Jagannathan Srinivasan
Shrikanth Shankar[+]
Souripriya Das
Oracle
1 Oracle Drive
Nashua, NH 03062
[+]500 Oracle Parkway
Redwood Shores, CA 94065

ABSTRACT
Partitioning is typically employed on large-scale data to improve manageability, availability, and performance. However, for tables connected by a referential constraint (capturing a parent-child relationship), the current approaches require individually partitioning each table thereby burdening the user with the task of maintaining the tables equi-partitioned, which not only is cumbersome but also error prone. This paper proposes a new partitioning method (*partition by reference*) that allows tables with a parent-child relationship to be logically equi-partitioned by inheriting the partition key from the parent table without duplicating the key columns. The partitioning key is resolved through an existing parent-child relationship, enforced by an active referential constraint. This logical dependency is used to automatically i) cascade partition maintenance operations performed on parent table to child tables, and ii) handle migration of child rows when partition key or parent key in parent table is updated, as a single atomic operation. This method has been introduced in Oracle Database 11gR1 with support for tables with both single level and composite partitioning methods. The paper describes the key concepts of table partitioning by reference method, discusses the design and implementation challenges, and presents an experimental study covering a usage scenario common in Information Life Cycle Management (ILM) applications.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *relational databases, query processing*, *concurrency.*

General Terms
Algorithms, Design, Experimentation, Performance.

Keywords: partitioning, partition by reference, information lifecycle management, partition pruning, partition maintenance, partition-wise join, vertical partitioning.

1. INTRODUCTION
A key challenge faced by database systems is to support ever-growing databases, which are quite common in OLTP and OLAP applications. The database sizes have grown from gigabytes to terabytes to petabytes. Such large scale data is typically handled by *horizontally* partitioning a logical table into a set of physical partitions. Users can perform operations on individual partitions, for example, exchange data of a partition with data from a staging table as well as perform operations such as query and DML operations on the logical table as a single unit.

Partitioning[1] although introduced primarily for improving manageability, has the additional benefits of improved performance and availability. Techniques such as partition pruning and partition-wise joins can significantly improve the query performance. Similarly, availability is improved as each partition can be mapped to a separate disk. Thus, unavailability of a disk impacts only the partitions residing on that disk.

However, a drawback of current partitioning support is that it works assuming a single table as a logical entity. Thus, if we have two related tables (connected by a referential integrity constraint capturing a parent-child relationship), which together represent a single logical entity, then the user has to individually manage them as two separate tables. There are several problems with this approach:

- The partitioning key columns must be replicated in the child table. This would require either modifying the existing referential constraint to include the partitioning columns, or to introduce a new referential constraint to ensure that the partitioning column values in the child table points to partitioning column values in the parent table. It also complicates loading into the child table because the partition key, which is normally present only in parent table, needs to be picked up typically via join with parent table.
- Next, user has to individually partition the two tables using the same partitioning method. Partition maintenance operations such as split of a partition or merge of partitions, need to be repeated on each of the tables.
- Similarly, DML operations that update partitioning key columns need to be repeated individually on the two tables.

Thus, lack of partitioning support for tables connected by a referential constraint burdens the user with the task of maintaining parent and child tables equi-partitioned, which is cumbersome and error-prone. Note that equi-partitioning here refers to having one to one correspondence between parent table partitions and child table partitions.

[1] By partitioning we mean *horizontal* partitioning throughout the paper. Other *vertical* partitioning of data is also possible, which when referred would be explicitly stated.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06...$5.00.

Also, data integrity expressed in terms of referential integrity constraint works at logical table level and is not aware of underlying physical partitioning. Consider, an Information Licfecyle Management (ILM) application, which manages groups of data (i.e. partitions) by distributing them across different types of storage devices. Such an application may progressively move older (inactive) data from fast devices to slow devices. If user forgets to move the child table data along with the parent table data, system will not be able to detect the problem thereby compromising the ILM application benefits.

Thus, the paper proposes a new partitioning method *partition by reference* for child tables, so that a child table is equi-partitioned with respect to the parent table and this equi-partitioning is implicitly maintained by the database system. The key benefits of this partitioning method are as follows:

- *Avoidance of duplicating partitioning key columns*: Tables with a parent-child relationship are logically equi-partitioned by inheriting the partition key from the parent table without duplicating the key columns.
- *Simplified management and atomicity of operations*: User has to *only* perform operations on the parent table. The logical (parent-child) dependency is used to automatically i) cascade partition maintenance operations performed on parent table to child tables, and ii) handle migration (deletion) of child rows when partition key or parent key in parent table is updated (deleted), as a single atomic operation.
- *Improved data integrity*: The database system enforces stricter data integrity, namely, ensures that the referenced parent key is present in the corresponding parent partition (as opposed to be present in any partition of the parent table).

This partitioning method is implemented in Oracle Database Release 11gR1 [10]. This feature can be used to create one or more *reference partitioned* child tables with respect to a parent table partitioned using a single level or composite partitioning method. Also, a reference partitioned table in turn can have reference partitioned child tables thus allowing a tree of child tables.

Three key challenges in implementing support for reference partitioned tables are: 1) support cascading PMO and row migration operations in a single atomic operation, 2) minimize the overhead incurred to enforce the stricter partition-specific referential integrity constraint during DML and load operations, and 3) support partitioning-specific query optimizations. DML operations on child tables are performed as before on the logical table. The database system ensures that the data goes to the right partition. Also, queries involving reference partitioned tables support partition-pruning and partition-wise join optimizations.

We conducted experiments using TPC-H dataset [9], specifically by considering `ORDERS` and `LINEITEM` tables, which are connected by a referential constraint. The child `LINEITEM` table was partitioned with and without reference partitioning with respect to the parent `ORDERS` table. The performance experiments demonstrate that we are able to support stricter referential integrity constraint with minimal additional overhead and process queries efficiently using partitioning-specific optimizations.

Thus, the generic and efficient support for partition by reference method extends the benefits of partitioning from individual tables to a set of related tables, which can together share the storage characteristics. Emerging ILM applications as well as traditional (OLTP and OLAP) applications can benefit from this feature.

Related Work: Horizontal partitioning [1,4,5] of tables is a well-understood topic and is supported by most database systems including Oracle [16], IBM DB2 [15], Microsoft SQL Server [12], PostgreSQL [14], and MySQL [13]. The objective of horizontal partitioning is to divide large amounts of data into a group of smaller amount of data so that the data is easily manageable and to optimize the access by application programs to relevant portions of the data by skipping non-relevant portions. A row in a partition is made up of all the columns of a table and a subset of these columns forms the partitioning key. Most popular partitioning methods are by range, list, and hash. Single-level partitioning methods are supported by almost all database systems, whereas composite partitioning methods, where each partition is further divided into subpartitions, are supported by Oracle and MySQL. For example, in Oracle, a range-partitioned table can be further divided into subpartitions based on range, list, or hash.

Vertical partitioning [1,2,3] is typically implemented by manually breaking a table into multiple tables by grouping related columns with similar access pattern, and using a referential constraint to connect the resulting tables. Its objective is to improve performance of queries by accessing only relevant columns of rows. Recently column-oriented stores [7] have been proposed for supporting read-intensive data. This can be viewed as an extreme case of single column virtual partitions.

However, as mentioned earlier, these do not address the problem of uniformly partitioning a set of related tables connected via a referential constraint.

Organization of the paper: Section 2 presents the terminology and key concepts. Section 3 discusses the design, and outlines the implementation challenges and the proposed solutions. Section 4 describes applications that can benefit from this feature. Section 5 gives an experimental study conducted using TPC-H dataset. Section 6 concludes the paper.

2. KEY CONCEPTS

2.1 Terminology

Here we primarily focus on terms related to horizontal partitioning of tables. The partitioning methods supported can be categorized as *single-level* or *composite*. Single-level partitioning methods allow distributing the data into a set of partitions based on *range* (or *list* or *hash*) of partitioning key values. Composite partitioning methods allow further breaking a table partition into a set of *sub-partitions* based on range (list or hash) of sub-partitioning key values. The *partitioning key* represents the set of columns whose value controls the placement of row in the appropriate partition or the sub-partition. Indexes on partitioned tables can be *local* or *global*. Local indexes are equi-partitioned with the table, while global indexes are non-partitioned or partitioned without being equi-partitioned with the table.

For partitioned tables, a set of *partition maintenance operations* (PMOs) are defined. They include *add partition*, *drop partition*, *truncate partition*, *move partition*, *split partition*, *merge partition*, and *exchange partition* operations. These operations are further

categorized as *PMOs with* and *without data movement*. The *PMOs without data movement* are almost zero time operation involving updates to partitioning metadata. These include add partition, drop partition, truncate partition, and exchange partition operations. The *PMOs with data movement* involve moving data. These include move partition, split partition, and merge partition operations. For composite partitioned tables, the PMOs have counterparts which create and manipulate sub-partitions (as opposed to partitions). PMOs with data movement optionally perform *index maintenance* (global, or local and global) to update indexes affected by the operation. If index maintenance is not performed, global indexes and affected local index partitions will become *unusable*.

Two key optimizations supported for partitioned tables are *partition pruning* and *partition-wise joins*. *Partition pruning* prunes out partitions based on the partitioning key predicate. The *partition-wise join* optimization, applicable on queries joining two partitioned tables on their partitioning key, translates to pair-wise join of corresponding table partitions. This is applicable when the partitioning key is same in the two tables.

Referential integrity constraints link tables in a parent-child relationship, namely *foreign key* of *child* table links to *parent key* (or *referenced key*) of *parent* table. The constraint is enforced during DML operations on parent and/or child tables. Some systems also support the option of cascading update (and delete) of parent key of a parent table to the corresponding child tables.

2.2 Partition by Reference Method

The partition by reference method equi-partitions a table with respect to another table based on a referential constraint. A reference partitioned table may be involved in multiple referential constraints. The specific constraint used as the basis for reference partitioning is known as the *partitioning constraint*.

The partitioning constraint identifies the parent table to equi-partition with, and defines the row-to-partition mapping for the reference partitioned table. Specifically, let P be the function that maps partition keys to partitions in the parent table, and M be the function that maps foreign keys (*fk*) in the child table to partition keys for the parent table. Then C, the function that maps foreign keys to partitions in the child table, is defined as:

$$C(fk) = P(M(fk))$$

The function M is defined by the partitioning constraint, and it can take either of two forms:

1. If the partitioning key includes a column that is not in the parent key, then this function's results depend on the data in the parent table. To compute the function we could, for example, find the parent row based on the parent key and get the parent's partitioning key from the row. Thus the child's row-to-partition mapping (C) is *data-driven* because its results depend on the data in the parent table. (Effectively, the location of referenced parent key controls the placement of child row.)

2. Otherwise all partitioning key columns are in the parent key, and this function can simply permute and possibly truncate the foreign key to yield the parent table partitioning key. In this case the child's row-to-partition mapping (C) is defined by the parent table's mapping (P) without depending on the data in the table. If the parent table's mapping is defined entirely by metadata, as it is for conventional partition methods, then the child's mapping is *metadata-driven* (otherwise the parent must be a reference partitioned table with a data-driven mapping, and the child's mapping is data-driven as well).

Partition by reference is a single level partition method. A reference partitioned table cannot be further subpartitioned to yield a composite partitioned table. If the parent is composite partitioned then the reference partitioned child will have one partition for each subpartition in the parent, otherwise the reference partitioned table will have one partition for each partition in the parent. Thus a reference partitioned table and its parent always have the same number of physical fragments.

2.3 Semantics of Operations

2.3.1 Data Definition Operations

To create a reference partitioned table, the user identifies the partitioning constraint by name as part of the PARTITION BY clause during CREATE TABLE. For example, let `ORDERS` table be a range partitioned table created as follows:

```
CREATE TABLE ORDERS(
  o_orderkey int, o_orderdate date, ...,
  CONSTRAINT ORDERS_pk PRIMARY KEY(o_orderkey))
PARTITION BY RANGE(o_orderdate)
(PARTITION part_94 ...);
```

Then users can create `LINEITEM` table as reference partitioned with respect to `ORDERS` table as follows:

```
CREATE TABLE LINEITEM(
  l_orderkey int NOT NULL, ...,
  CONSTRAINT LINEITEM_fk
    FOREIGN KEY(l_orderkey)
    REFERENCES ORDERS(o_orderkey))
PARTITION BY REFERENCE(LINEITEM_fk);
```

The partitioning constraint must be enabled and non-deferrable. Oracle requires a primary key or unique constraint on the parent key of a referential constraint, and we require that the primary key/unique constraint on the partitioning constraint's parent key is enabled and non-deferrable. Oracle's referential integrity mechanism does not require all-NULL or partially-NULL child keys to match a parent key; however, for reference partitioned tables the partitioning constraint's foreign key columns must be constrained NOT NULL to ensure that every child key has a matching parent key. These conditions guarantee that each foreign key always maps to a single parent row, and thus our row-to-partition mapping is well-defined. Furthermore, since the primary key/unique constraint is enabled Oracle guarantees that the key will be indexed.

Once a reference partitioned table is created it is not possible to disassociate it from its parent. In particular, it is not possible to drop a table that has reference partitioned child tables unless the child table is dropped first.

The reference partitioned tables can have their own child tables, which in turn are reference partitioned, recursively with unlimited level of depth. The recursive level of depth is constrained only by resources such as memory.

2.3.2 Partition Maintenance Operations (PMOs)

Partition maintenance operations that result in create or drop partitions (ADD, DROP, SPLIT, and MERGE PARTITION) *cannot* be run *explicitly* for a reference partitioned table. Instead,

when such PMOs are run against a table that has reference partitioned children the operation implicitly cascades to the reference partitioned child tables. At the user's discretion, index maintenance is either performed for all tables affected by the operation, or none. These PMOs always appear to be a single atomic operation, regardless of how many tables are affected.

When PMOs create new partitions for reference partitioned tables, the partitions will inherit partition name and tablespace attributes from the associated parent partition unless the user overrides these defaults using the DEPENDENT TABLES clause we have introduced for PMOs. For example, if a range-partitioned table named `parent` has two children named `child0` and `child1`, and `child0` has a child named `gchild0` (Figure 1), then the following ADD PARTITION operation will put the new partition **`ptn`** in tablespace `tbs_1` for `parent` and `child1` and in tablespace `tbs_2` for `child0` and `gchild0`.

```
ALTER TABLE parent ADD PARTITION ptn
  VALUES LESS THAN (...) TABLESPACE tbs_1
  DEPENDENT TABLES
    (child0 (PARTITION TABLESPACE tbs_2))
```

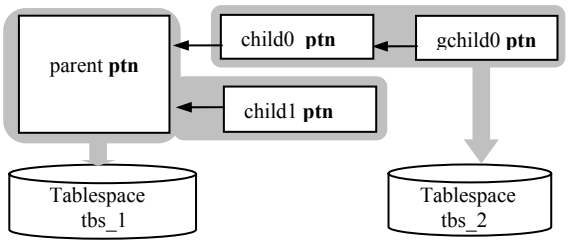

Figure 1. Tablespace Assignment after ADD PARTITION

PMOs that do not result in create or drop partitions (MOVE, TRUNCATE, and EXCHANGE PARTITION) can be run directly against reference partitioned tables and by default do not cascade when run against parent tables. A CASCADE option for these operations could be provided to allow operating only on parent table, which implicitly cascades to child tables.

The semantics for EXCHANGE PARTITION and TRUNCATE PARTITION have been enhanced for reference partitioned tables. Ordinarily, if a table is the parent table for an enabled referential constraint then it is only possible to exchange or truncate a partition in the table if the child table is empty; Oracle relies on this to efficiently maintain referential integrity. When the child table is reference partitioned then exchange and truncate partition are allowed as long as the associated partitions in the child table are empty.

All PMOs, except EXCHANGE PARTITION, can be run for a reference partitioned table when the parent key index is unusable.

2.3.3 Data Manipulation Operations

UPDATEs in the parent table may implicitly cause *cascaded row migration* in the child table to maintain the property that child rows are in the correct partition with respect to their parents. This typically occurs when a partitioning key is updated, causing a parent row to migrate to another partition, requiring migrating the associated child rows to the correct partition.

Parent key updates may cause cascaded row migration even when they do not cause row migration in the parent table; consider an example with the following data distribution:

Partition	Rows
Parent Part 0	<partition_key = 10, parent_key = 1>
Parent Part 1	<partition_key = 20, parent_key = 2>
Child Part 0	*empty (parent key 1 is not referenced)*
Child Part 1	<foreign_key = 2>

Given this data, if we perform the following update

```
UPDATE parent SET parent_key = parent_key + 1
```

then the children of key 2 must migrate from partition 1 to partition 0. This yields the following data distribution:

Partition	Rows
Parent Part 0	<partition_key = 10, parent_key = 2>
Parent Part 1	<partition_key = 20, parent_key = 3>
Child Part 0	<foreign_key = 2>
Child Part 1	*empty (parent key 3 is not referenced)*

Note that in Oracle, the primary key update in the parent table is not cascaded to child rows. That is, Oracle does not support UPDATE CASCADE option for referential constraints. Instead, only UPDATE RESTRICT semantics is supported for referential constraints. Thus, updates to primary key in parent table, such as in example above, succeed only if all child keys are resolved (that is, have a corresponding matching parent key). Also, the option DELETE SET NULL is disallowed for partitioning referential constraint because this would cause child rows with NULL foreign key, which cannot be mapped to any partition.

DELETEs in the parent table implicitly causes *cascaded row deletion* in the child table, provided the referential constraint was created with DELETE CASCADE option; otherwise DELETE RESTRICT semantics is applicable. This behavior is same as that for traditional referential constraints.

INSERTs into a reference partitioned table will automatically put each row is the appropriate partition. UPDATE of the partitioning constraint's foreign key may cause row migration between partitions so that updated child keys are in the appropriate partition.

2.3.4 Queries

Query by themselves do not have any different semantics for reference partitioned tables. However, query optimization has significant differences as outlined below.

Reference partitioned table pruning differs from the partition pruning for queries on conventional partitioned tables in that the pruning cannot be done at compile time even for equality conditions on foreign key columns because of possible row migration in the parent table or deletion followed by insertion of the same parent key row into a different partition.

Predicates on the foreign key may be used for partition pruning. However, if there is no usable index (or index not usable for the

accessed partitions, in case of a partitioned index) on the parent key partition, pruning may be disabled.

Extended Pruning: When a table has more than one partitioning column, we can usually prune only on columns that are a prefix of the partitioning columns. However, if the prefix columns of different partitions have the same boundaries, we can use predicates on non-prefix columns at runtime to do pruning. This is referred to *Extended Pruning***.** For example, if a table T1 has the partitioning column (a,b) and its boundaries are (1,4) (1,10) (1,25) (2,5) (2,30) (2,50), then the predicate 'b < 7' can be used to prune the partitions for the same boundaries of the prefix 'a'. The extended pruning is applicable to reference partitioned tables when the parent partitioning key is a subset of the parent key.

Dynamic Pruning: Also referred to as *Subquery Pruning*, is used when a table can be pruned indirectly by using a subquery involving other tables. Consider tables `T1(a, b)`, `T2(c, d)` and `T3(e, f)`, and `T1` is partitioned on `'a'`. If the WHERE clause condition is of the form `'T1.a = T2.c AND T2.d = T3.e AND T3.f < x'`, then a query of the form

```
SELECT DISTINCT find_part_num(T1,A)
   FROM (SELECT T2.c A  FROM   T2, T3
            WHERE T2.d = T3.e AND T3.f < x)
```

can be used to find partitions of T1. The operator `find_part_num(T1,A)` returns the partition number based on the values of column A of the table T1. The optimizer will decide whether or not to apply the dynamic pruning based on the cost. In addition to the join subquery pruning as shown in the example above, a single table subquery pruning is also possible where the subquery is applied to the parent table without any joins to find the partition numbers.

Unlike non-reference partitioned tables for which the extended pruning or dynamic pruning is determined based on the partition key, applicability of extended pruning and dynamic pruning on reference partitioned tables is determined based on configurations of the parent key and the parent partition key.

Partition-wise Join: For join queries involving join between the referenced key of the parent table and foreign key of the child table, partition-wise join is always possible, thereby making this class of queries efficient. In a non-reference partitioned table case, if the parent partition key is not a subset of the parent key, the partition-wise join is not possible for a join between the primary key and the foreign key.

3. DESIGN AND IMPLEMENTATION

3.1 Creation

Before creating a reference partitioned table we check that all conditions are met for reference partitioning (e.g., the partitioning constraint must be enabled and not deferrable). Partitions inherit partition name and tablespace from the associated parent partitions, unless these attributes are explicitly provided by the user as part of the CREATE TABLE command.

3.2 Loading

Oracle does not ordinarily allow direct path loads into child tables of enabled referential constraints. We have provided support for direct path loads into reference partitioned tables that have an enabled partitioning constraint as long as they are not the child table for any other enabled referential constraint.

As each row is passed into the loader we lookup the parent key in the parent key index to verify that the parent key exists, thus enforcing referential integrity. To determine which partition the child row belongs in we must determine which partition the parent row is in; if the parent key is locally indexed then the parent key columns are sufficient to determine which partition the parent row is in, otherwise we get the parent row's row identifier from the index lookup and determine which partition contains the parent row based on this row identifier.

Due to the architecture of Oracle's direct path loader, DML on the parent table will not see uncommitted rows that have been loaded into the reference partitioned table. Such DML could not detect, for example, that it was deleting a parent key that is referenced by child rows that have been loaded but not committed. Thus we take locks on the parent table to prevent DML concurrent with the load. If we are loading into a specific partition of the reference partitioned table then we take partition-level locks on the corresponding partition of the parent table, otherwise we take a table-level lock on the parent table.

3.3 PMOs without Data Movement

PMOs without data movement can further be divided into:

- PMOs that cascade to reference partitioned tables and cannot be run directly against a reference partitioned table. These PMOs are ADD for range- and list- (sub)partitions and DROP.
- PMOs that do not cascade, but can be run directly against a reference partitioned table, and have special semantics involving reference partitioned tables (see section 2.3.2 for a description of these special semantics). These PMOs are EXCHANGE and TRUNCATE.

ADD for range- and list- (sub)partitions is purely a metadata operation and cascading it simply involves inheriting attributes as appropriate for the new partition(s) in the child tables and making all metadata changes for all affected table as a single transaction.

DROP may involve global index maintenance in addition to metadata updates. When this operation cascades, if the user requested global index maintenance we perform the index maintenance for each affected table, then we make the metadata changes for all affected tables. All of this is done as a single transaction.

Since EXCHANGE and TRUNCATE do not cascade, the changes are limited, but the locking protocol is changed due to the new semantics for these operations on tables that have reference partitioned child tables. Specifically, we take partition-level locks on the associated partitions of reference partitioned child tables, rather than taking table-level locks as we do when the table is the parent for an ordinary referential constraint.

3.4 PMOs with Data Movement

MOVE does not have any special semantics or implementation for reference partitioned tables.

All other PMOs with data movement implicitly cascade to reference partitioned tables, and cannot be run directly against a reference partitioned table.

PMOs that move all rows to a single partition (such as MERGE PARTITION for tables that are not composite partitioned) are processed as follows: for each table affected by the operation we move the relevant rows to a new physical location and perform global index maintenance (if requested by the user), then we update the metadata for all affected tables. All of this is done as a single transaction. If the user requested local index maintenance then we rebuild the affected local index partitions after committing the PMO.

For other PMOs with data movement (such as SPLIT PARTITION), the data redistribution in a reference partitioned table depends on the data redistribution in its parent. These PMOs are processed top-down (that is, a parent table is processed before any of its reference partitioned children). When we process a table that has reference partitioned children we log affected parent keys and their destination to a separate logging table. When we process a reference partitioned table we join the parent's logging table with the reference partitioned table (joining the partitioning foreign key to the associated parent key in the logging table) to determine the data redistribution in the reference partitioned table. Once all tables have been processed we update the metadata for all tables. Again, all of this is done as a single atomic transaction. As with MERGE PARTITION, any required global index maintenance is done as part of the main PMO transaction, while any required local index maintenance is done in follow-up transactions.

Composite Partitioned Parent Tables
When the parent table is composite partitioned, PMOs with data movement may affect many partitions in the reference partitioned table. For example, MERGE PARTITION for a composite-partitioned table could merge a partition that has M_0 subpartitions with a partition that has M_1 subpartitions to yield a new partition that has N subpartitions. For the reference partitioned table this would require redistributing data from $M_0 + M_1$ partitions across N new partitions. Similarly, SPLIT PARTITION could require redistributing M partitions across $N_0 + N_1$ new partitions.

3.5 Row-to-Partition Mapping for DML

If the row-to-partition mapping is metadata-driven (e.g., if the parent table is conventionally partitioned and the parent key is a superset of the parent table's partitioning key) then the parent table's partition descriptor is used to map rows to partitions for the reference partitioned table. In this case the reference partitioned table acts like an ordinary partitioned table during DML.

Otherwise the row-to-partition mapping is data-driven, which implies two things:

1. INSERT (and UPDATE of the partitioning foreign key) for a reference partitioned table requires probing the parent key index to determine where the parent key resides.
2. UPDATE on the parent table may cause cascaded row migration in the child table.

Care must be taken to avoid a race condition where one session, doing INSERT into a reference partitioned table, finds where the parent key resides and inserts into the corresponding partition of the child table, while another session concurrently migrates the parent key to a new partition thus leaving the parent and child out of sync. To handle this:

- INSERT in a reference partitioned table does two probes for the parent key. The first is done before inserting the row into the table, and the second is done after processing the row. If the second probe finds that the parent key has migrated then we rollback the row and retry; during the retry we lock the parent row as part of the first probe to ensure that it will not migrate again.
- UPDATEs that induce cascaded row migration wait for uncommitted child keys. Thus, once an INSERT in the child table has performed the second probe described above any UPDATE that would cause cascaded row migration for the child row will be blocked until the INSERT commits.

Cascaded Row Migration
When a parent key migrates to a new partition, or a parent key is updated to take the value of an existing parent key, we perform cascaded row migration. This involves waiting for uncommitted child keys (as described above) then running recursive SQL to migrate the child rows. If the child table itself has reference partitioned children then this process may repeat recursively. For example, the top level SQL may trigger recursive SQL for cascaded row migration in a child table which may trigger recursive SQL for cascaded row migration in a grandchild table.

3.6 Query Optimization

Depending on the relationship between the referenced key and partitioning key, there are three cases:

1. Referenced key is superset of the partitioning key: use the parent's partitioning information for equality, range, and extended pruning.

2. Referenced key intersects the partitioning key (but is not a superset)

- if the referenced key intersects a prefix of the partitioning key, then we probe the referenced key index for equality pruning and use the parent's partitioning information for range and extended pruning.
- otherwise, we probe the referenced key index for equality pruning , do dynamic pruning to effect range pruning, and use the parent's partitioning information for extended pruning.

3. All other cases: we probe the referenced key index for equality pruning, do dynamic pruning to effect range pruning, and do not do any extended pruning.

Equality Pruning
Predicates on the foreign key can be used for partition pruning. For full pruning support, one of the following conditions must be met:

1.There is an index on the parent key that is usable, and all partitions of this index that are accessed are usable.

2. In the parent table, all partitioning and subpartitioning columns are also parent key columns. For example, the parent key is <i, j> and the table is `PARTITION BY RANGE(i)`.

If none of the above conditions are met, then (i) if the `skip_unusable_indexes` parameter is FALSE which means that unusable indexes cannot be ignored, an error will be raised if query on the reference partitioned table is attempted, or (ii) if the

skip_unusable_indexes parameter is TRUE which means that unusable indexes can be ignored, query on the reference partitioned table is allowed but partition pruning may not occur.

Range Pruning
In some cases partition pruning for range predicates is possible for reference partitioned tables. If the referenced key intersects the parent table's partitioning key, and range pruning is possible for the parent table, then range pruning is possible for the ref-partitioned table. For example, consider the following two tables:

```
PARENT: PRIMARY KEY (a, b, c)
        PARTITION BY RANGE(a, b)

CHILD: FOREIGN KEY (x, y, z) REFERENCES
                             PARENT(a, b, c)
       PARTITION BY REFERENCE
```

In this case a range on the child's foreign key can be mapped to a range on the parent's partitioning key and we can prune using the parent's partitioning information. In effect, the parent's partitioning information can serve as a surrogate for the reference partitioned table.

We do not require parent's partitioning key to be a superset of the referenced key. Consider the following example:

```
PARENT_2: PRIMARY KEY (a, b)
          PARTITION BY RANGE(a, c)

CHILD_2: FOREIGN KEY (x, y) REFERENCES
                            PARENT_2(a, b)
         PARTITION BY REFERENCE
```

A range on <x, y> in the child can be mapped to a range on <a> in the parent. Note that this applies even when the parent table is itself reference partitioned. For example, if we have another table in the prior example:

```
GRANDCHILD_2: FOREIGN KEY (i, j) REFERENCES
                                 CHILD_2(x, y)
              PARTITION BY REFERENCE
```

A range on <i, j> columns of GRANDCHILD_2 can be mapped to a range on <x, y> columns of CHILD_2, which can then be mapped to a range on <a> column of PARENT_2.

There are two cases for range pruning:

- Case 1: the referenced key intersects a prefix of the partitioning key -- In this case, the partition can be found using parent's partitioning information without the primary key index lookup, thus we always do range pruning using the partition iterator.
- Case 2: primary key index of the parent table is global -- In this case, the partition is found using find_part_num() (described earlier in Section 2.3.4) with the list of partition column values as parameters.

Group-by Pushdown
The reference partition iterator row source can be pushed up above the group-by row source so that the group-by can be performed on partition level when more than one partition is accessed. Group-by pushdown for the reference partitioned table is possible when the group by key is a (not necessarily strict) superset of the foreign key because the group-by columns become unique across partitions in this case.

Order-by Pushdown
If the parent table of a ref-partitioned table is range-partitioned or list-partitioned with a range property order-by may be pushed down to the partition level. If the parent table is reference partitioned, we keep following its parent until we hit the first non-reference partitioned table to check the range property. Order-by pushdown for the reference partitioned table is possible when the parent table is single level partitioned (i.e., is not composite partitioned) and some prefix of the parent's partitioning key intersects with the parent key, and the order-by key is a prefix of the intersection or the parent's partitioning key is a prefix of the order-by key because the partial order for each partition becomes also the total order across partitions in these cases.

Partition-wise Join
Equi-joins between the foreign key in a reference partitioned table and the associated referenced key in its parent table may be done as partition-wise joins. Partition-wise join can be performed either in serial or parallel. The reference partitioned table cannot be composite-partitioned. However, the parent table can be composite-partitioned. The partition-wise join is possible for the following cases:

1. complete join between the foreign key in a reference partitioned table and the associated referenced key in its parent table
2. when the partitioning key is a subset of referenced key and join is on those columns between the parent table and the reference partitioned table
3. when the partitioning key is a subset of referenced key and join is on those columns between reference partitioned tables with the same root parent table
4. when the partitioning key is a subset of referenced key and join is on those columns between root parent table and reference partitioned table (e.g. join between a parent table and a grandchild table)

In general, when row-to-partition mapping is metadata-driven (such as cases 2, 3, and 4 above), we could use the parent table's partition descriptor for partition-wise join determination. This would also allow partition-wise join between unrelated reference partitioned tables, for example.

4. APPLICATIONS

This section describes a few application scenarios where reference partitioned tables would be useful.

4.1 Information Lifecycle Management

Information Lifecycle Management (ILM) [6,8,11] refers to management of data throughout its lifetime. New government regulations and guidelines, such as Sarbanes-Oxley, HIPAA, DOD5015.2-STD in the US and the European Data Privacy Directive in the European Union, are a key driving force in requiring organizations to retain and control information (data) for very long periods of time.

Thus, ILM requires reconciling two objectives, namely, i) to store vast quantities of data for the lowest possible cost; and ii) to meet the new regulatory requirements for data retention and protection. Partitioning plays a key role in ILM applications as data could be classified into different categories and each category could be

mapped to a separate partition, which in turn can be stored on a different device. By using low-speed devices for infrequently accessed data, and high-speed storage devices for frequently accessed data, ILM objectives can be met. There is another proposal [8], which proposes that the importance of information should be determined by risk of not having the information available readily. High risk data should be stored in high-end device even though their access may be infrequent.

With reference partitioning support, we can set-up a configuration, where only the parent table partition needs to be assigned the storage characteristics and corresponding child table partitions implicitly gets stored in the parent partition storage device (see Figure 2).

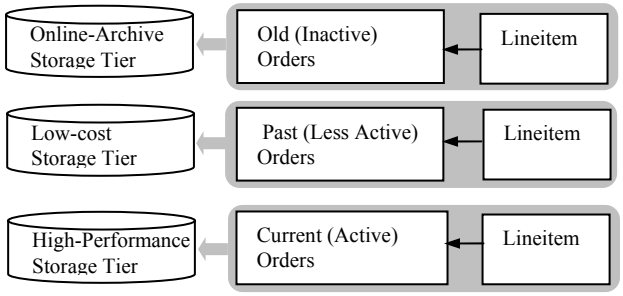

Figure 2. Managing Orders and Lineitem Table

We can leverage the cascading PMO and row migration features to support two types of ILM applications.

Coarse grain ILM: In this category, data is typically classified by a date field. For example, let the ILM requirement be to only hold recent 7 years (84 months) of orders in three storage tiers consisting of 18 months, 36 months, and 30 months respectively.

This could be implemented as follows:

- Create one tablespace for each of the three storage tiers
- Create the parent `ORDERS` table partitioned by range on `o_orderdate` field (one partition per month) and assign the partitions to one of the three tablespaces. The leading partition created with `MAXVALUE` as the partition bound.
- By creating the child table as partitioned by reference with the parent table, they inherit the parent tablespaces.
- At the beginning of each month, the following PMOs need to be performed *only* on parent `ORDERS` table, which get cascaded to the child `LINEITEM` table: i) *split* the leading (current) partition at first day of current month, to create two partitions, one for past month orders and other for new orders, ii) *drop* the trailing (oldest) partition, and iii) *move* the partition occurring at the boundary of each storage tier to the next level storage tier.

Note: The last step also requires support for cascading `MOVE PARTITION` operation to child tables, which is currently not supported in Oracle 11gR1. In absence of that two individual move operations need to be performed on the parent and child tables respectively.

Fine grain ILM: In this category, the ILM needs to be applied down to individual rows. Typically, a business related attribute categorizes a row as *active*, *less active*, and *inactive* respectively, which can be used as the partitioning key. In this case, the parent table can be partitioned by list of values possible for the business attribute. The child table is created as before using partition by reference with respect to the parent table. However, unlike the previous case, the rows migrate between partitions based on update to the partition key, which could be controlled by a business policy. Here the cascading of row migration to child table rows simplifies the management.

Note that the two techniques could also be easily combined to create a *hybrid grain* ILM application.

4.2 Vertical Partitioning

When a table has a large number of columns and a small set of the columns is accessed frequently, the table can be broken into two or more small tables that are connected by a parent-child relationship. For example, consider a voluminous sensor data maintenance application. This could be represented say as three tables, `SENSOR_DATA` (parent), and `SENSOR_DATA_DETAIL_1` (child) and `SENSOR_DATA_DETAIL_2` (grandchild) respectively. The frequently accessed columns can be stored in `SENSOR_DATA` table to improve the query performance. By use of reference partitioning, the partitioning key does not have to be duplicated in the child tables and row migrating operations and PMOs on the parent table are cascaded automatically to child tables.

Thus, reference partitioning could simplify *two-dimensional information life cycle management* where the data is horizontally and vertically partitioned and then the resulting partition is assigned to an appropriate storage tier. Figure 3 shows one such configuration.

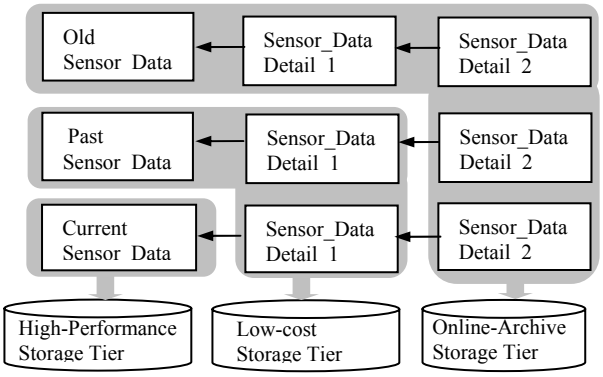

Figure 3. Managing Sensor Data and Details Table

In this case, the ability to specify tablespaces for child table partitions as part of child table creation and PMOs on parent table (using `DEPENDENT TABLES` clause) is useful as the child table representing less frequently access columns could be stored on a different storage tier as compared to corresponding parent table partition. Also, a `MOVE PARTITION` operation on a parent may need to be *selectively* performed for child tables. In the example above, if the past `SENSOR_DATA` table partition is moved to old `SENSOR_DATA` table partition, then the move needs to be cascaded to corresponding `SENSOR_DATA_DETAIL_1` partition. However, no action needs to be taken for corresponding `SENSOR_DATA_DETAIL_2` partition. For such cases, the ability to individually move the parent and child table partitions is useful.

5. EXPERIMENTAL STUDY

This section presents experimental study characterizing the performance of reference partitioned tables.

5.1 Experimental Setup and Dataset

The experimental setup uses a *desktop* machine: single Intel P4 CPU (3.0GHz with hyper threading), 2GB Memory. The operating system is Red Hat Enterprise 32-bit Linux AS release 3. The Oracle Database 11g, Release 1 with *db_cache_size*=320M and *pga_aggregate_target*=400M.

The dataset is generated from the DBGEN program, provided by the TPC-H Benchmark [9]. Two of the tables, `ORDERS` and `LINEITEM`, were used for the experiment (Figure 4).

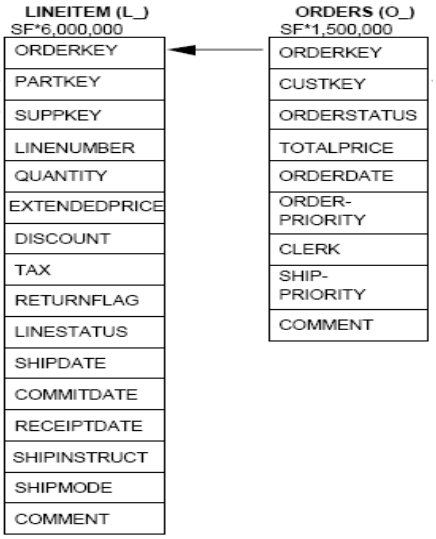

Figure 4. TPC-H Orders and Lineitem Table

The `LINEITEM` table has ~6 million rows, whereas the `ORDERS` table has 1.5 million rows. Both tables are created with compression option on. After loading the dataset, the primary key and foreign key constraints are created. There is a single index on the `ORDERS` table, specifically a global index used to enforce the primary key and unique key constraints, and no indexes on the `LINEITEM` table. Before running the experiments object-level statistics are collected. Each experiment is conducted ten times in warm cache, and the average query execution time is computed.

Three configurations are considered and compared:

- *Non-Referenced Configuration*: To allow individually partitioning `LINEITEM` table, the `o_orderdate` column of `ORDERS` is duplicated in `LINEITEM` table. Thus, referential constraint from `LINEITEM` table to `ORDERS` is on (`o_orderdate`, `o_orderkey`) columns. `ORDERS` has a unique constraint on `o_orderkey` column and primary key constraint on (`o_orderkey`, `o_orderdate`) columns. Both `ORDERS` and `LINEITEM` table are partitioned on o_orderdate, and `l_orderdate` respectively, creating 5 partitions part_94, part_95, part_96, part_98, part_mx (see Figure 5). This configuration is referred to as NRD (D indicates orderdate duplication).
- *Referenced Configuration 1*: Similar to NRD, except the child table `LINEITEM` is reference partitioned with respect to parent table `ORDERS` (Configuration referred to as RD).
- *Referenced Configuration 2*: Similar to RD, except that the orderdate field is not duplicated. This also means the referential constraint from `LINEITEM` to `ORDERS` is on `o_orderkey` column. Also, `ORDERS` table has just the primary key constraint on `o_orderkey` column. This configuration is referred to as R.

The three configurations are used for single level partitioning experiments as described below.

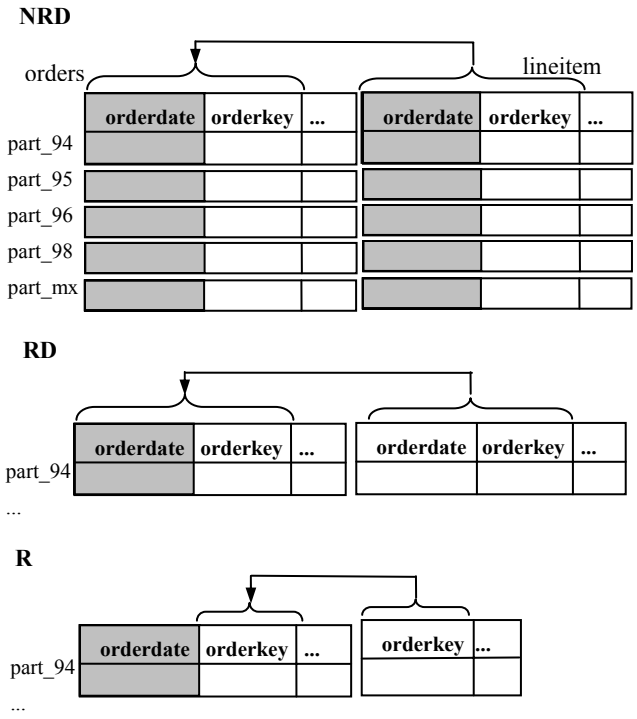

Figure 5. The Three Configurations Used for Experiments

5.2 Single Level Partitioning Experiment

The data was loaded into each of the configurations and various PMOs were performed. The initial sizes of various configurations and load times are shown in Table 1.

Table 1. Storage Costs

Config.	Orders	Orders Index	Lineitem	Load Time
NRD	168MB	48MB	603MB	234s
RD	168MB	48MB	603MB	243s
R	168MB	32MB	579MB	196s

As expected the storage costs are lowest for R configuration as orderdate column is not duplicated. Load times is also lowest for R as the loading of `LINEITEM` table does not require joining with the `ORDERS` table to get the orderdate field as in the other two configurations where orderdate field is duplicated.

Next, we measured the PMOs for the three configurations for three different sizes of the partitions. The PMOs performed global index maintenance in all cases. This could introduce variability

into the results based on internal state of the global index (such as fragmentation).

Truncate Partition: Truncate partition is performed bottom-up (i.e., truncate a partition in the `LINEITEM` table followed by truncate of the corresponding partition in the `ORDERS` table).

A truncate partition operation involves:

1. Checking for empty child tables/partitions, as required for enabled referential integrity constraints. In the NRD configuration the referential constraint must be disabled, so this check is skipped. For other configurations, the check results in a full scan of one partition in the child table, but since the child partition is empty, this operation is not expensive.
2. Performing global index maintenance to remove the rows being truncated from the index, which is proportional to the number of rows in the partition.
3. Updating metadata to remove all rows from the partition. Since this is only a metadata update, this is not expensive.

Thus, global index maintenance is the primary component of execution time in this experiment resulting in execution time proportional to the number of rows in the partition (Figure 6).

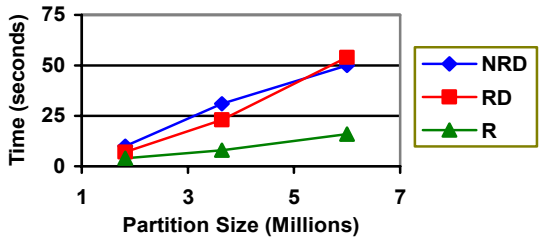

Figure 6. Truncate Partition

However, for the R configuration, the index row size (orderdate column being absent) is roughly 2/3rd of index row size for RD and NRD configurations. This translates to updates to fewer index blocks for R configuration resulting in improved performance in comparison to other two configurations (Figure 5).

Exchange Partition: In this experiment we performed an "exchange in" operation where the partitions, which are initially empty are exchanged with tables that have data. "Exchange in" is performed top-down.

The work done for exchange partition is similar to the work for truncate partition: (1) check for empty partitions in the RD and R configurations (2) perform global index maintenance to insert the new rows into the index, and (3) perform metadata updates to exchange the data.

In addition, exchange performs a fourth step: (4) iterate over all rows exchanged in and validate that they belong in the partition. For the `ORDERS` table, and the `LINEITEM` table in configurations NRD and RD, this step involves checking the row against the `ORDERS` table partition mapping descriptor; for the `LINEITEM` table in configuration R this step involves probing the parent key index for each row to find where its parent is located.

The execution time for "exchange in" with global index maintenance is proportional to the number of rows in the partition as shown in Figure 6. However, the global index maintenance cost is lowest for R configuration, whereas validation cost is highest for R configuration. Thus, for smallest partition size, R configuration performs better than the other two, whereas for the largest partition size, R configuration takes the maximum time (Figure 6). That is, validation cost dominates the lower index maintenance benefit for larger size.

The additional validation performed as step 4 (in comparison to truncate operation) makes exchange partition slower than truncate partition for the same volume of data (Figure 6 and Figure 7).

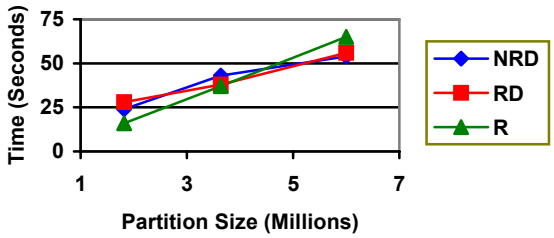

Figure 7. Exchange Partition

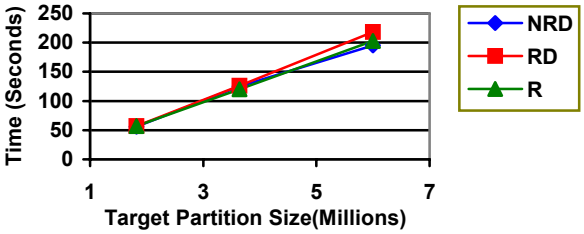

Figure 8. Merge Partition

Merge Partition:Merge partition involves (1) copying rows from the partitions being merged to a new partition, (2) performing global index maintenance, and (3) dropping the old partitions. Steps 1 and 2 are linear in the total number of rows in the partitions being merged, while step 3 only involves metadata and is low cost.

Figure 8 shows the execution time for processing both tables vs. the size of merged partition. Thus the execution time for merge partition is linear in the number of rows being merged. Further, because merge partition does not compute any row-to-partition mappings, there is no significant difference between the three configurations for this operation.

Split Partition: During split, the target partition for each row needs to be computed. This is the major difference between split and merge; otherwise the two PMOs are nearly identical.

For non-reference partitioned tables, table metadata is consulted to redistribute rows during split. This does not impose much overhead.

For reference partitioned tables, redistribution during split is data-driven (for cases, when the parent partition key is part of child foreign key, this could be optimized by using metadata driven row-to-partition mapping). As explained in section 3.4, the data driven mapping involves writing parent keys and the destination partition to a logging table when the parent table is processed and joining the logging table with the child table when the child table is processed. These logging and join costs make split more expensive for reference partitioned tables as compared to conventional partitioned tables (Figure 9).

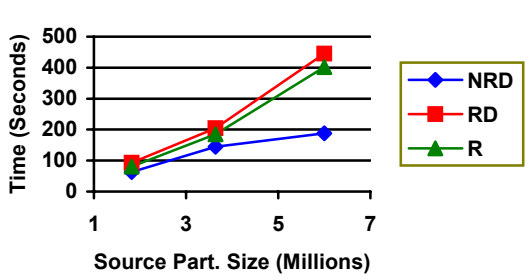

Figure 9. Split Partition

Queries: For queries, we used the NRD, RD, and R configurations consisting of five partitions. Two types of queries were considered:

- *Queries on child table with equality predicate on foreign key*: Specifically, select lineitems belonging to orders in a certain orderkey list (Q1). The goal was to see the applicability of equality predicate based partition pruning.
- *Join queries on parent and child table*: Specifically, select aggregate values, `sum(o_totalprice)` and `sum(l_extended_price)`, over pending orders (`o_orderstatus='P'`) and their lineitems information (Q2). The goal was to see the applicability of partition-wise joins.

The above queries were executed for R, RD, and NRD configurations. Also, since the foreign key for RD, and NRD includes orderdate, the query times obtained after appending orderdate predicates, namely, Q1 modified to return lineitems based on (orderkey, orderdate) , and Q2 modified to include additional join condition on orderdate predicate. These timings are reported under the RDE and NRDE categories respectively (Figure 10). The R configuration gives the best performance owing to use of partition pruning and partition-wise joins. For the RD configuration, by default, these optimizations do not get exercised as the original queries contain predicate only on orderkey (that is, does not include the complete foreign key). However, by augmenting the predicate to involve the complete foreign key, performance similar to R configuration is observed (RDE). Similarly, the NRDE configuration (which includes the complete foreign key in the predicate) shows improved performance with respect to NRD. Note that for the partition pruning experiment, performance of RDE and NRDE is not as good as that of R because a larger partition needs to be scanned after pruning.

The query experiment clearly demonstrates the use of partition-pruning and partition-wise joins even when the partition key is not duplicated in the child table (configuration R)

5.3 Composite Partitioning Experiment.

This experiment illustrates the use of reference partitioned table partitioned with respect to a composite partitioned parent table. The parent `ORDERS` table is partitioned by *range* on `o_orderkey` column and sub-partitioned by *list* on `o_orderstatus`, `'P'`, `'O'`, `'F'` (labeled spp, spo, spf in Figure 11).

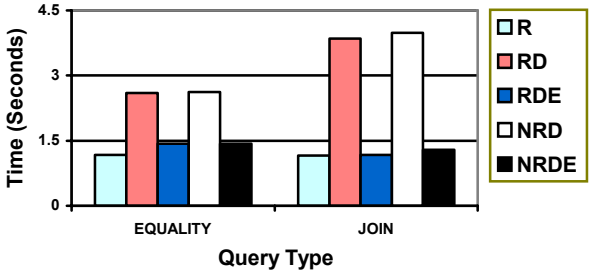

Figure 10. Partition Pruning and Partition-wise Joins

This configuration is used to examine the performance of cascaded row migration, which results from update to sub-partition key. Specifically, the o_orderstatus field of an order, which is pending (`o_orderstatus='P'`) is updated to `'F'`, causing the row to move from 'pending orders' sub-partition to 'fulfilled orders' sub-partitions and causing the cascaded row migration of corresponding `LINEITEM` records.

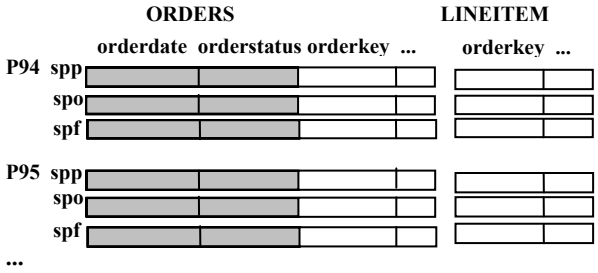

Figure 11. Comp-part. ORDERS & Ref-part. LINEITEM

The experiment was repeated with three sizes for reference partitioned (R) and non-referenced configurations (NRD). For the non-reference partitioned case, we have to additionally duplicate orderdate and orderstatus columns, to allow partitioning `LINEITEM` table independently. Figure 12 shows the update times. The x-axis gives the parent and child rows migrated. For example, updating 15 orders resulted in their migration to 'fulfilled orders' subpartition, and the migration of corresponding 75 lineitems to 'fulfilled lineitems' sub-partition.

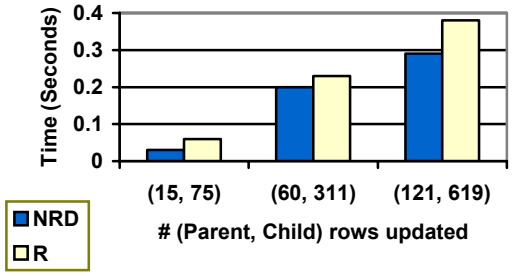

Figure 12. Cascaded Row Migration Performance

The results show a performance versus ease of use tradeoff. In the NRD configuration, cascaded row migration must be done manually by the user by issuing a separate update against the child table. While this is burdensome and potentially error prone,

in this case, it is the higher performance solution because all child rows can be processed with a single update.

In contrast, for the R configuration, Oracle implicitly cascades row migration, without effort on the application developer's part, by running a separate update for each parent key that migrates (as described in Section 3.5). This currently results in slower performance because running a series of updates is slower than running the single update that performs the same work.

However, this is not necessarily true in all cases; for example, if the manual update for the NRD configuration required a join with the parent table to find the rows to be updated, then implicit updates done in the R configuration could possibly exhibit higher performance. Also, in future, implicit cascaded row migration for R configuration can be optimized by internally batching keys to reduce the number of updates required.

5.4 Discussion

From a manageability perspective the configurations involving reference partitioned tables (R and RD) are more desirable than the conventional configurations (NRD) because PMOs cascade automatically and atomically to keep reference partitioned tables equi-partitioned, while this must be handled manually by the user in the conventional configuration.

For most PMOs, performance is about the same for all three configurations. The exceptions are SPLIT (ADD) PARTITION operation for range (hash) partitioned parent table and MERGE PARTITION for composite partitioned parent table, where reference partitioned tables suffer a performance penalty. Among these operations, the performance penalty is avoided for the SPLIT PARTITION operation when split yields an empty partition (for example, splitting a leading edge partition of a range partitioned table, which is quite common) by taking advantage of Oracle's "fast split" optimization which only requires updating the metadata.

Depending on query workload either the R or RD configurations could exhibit better performance, but there is rarely a workload where NRD is preferable over the reference partitioned configurations.

6. CONCLUSIONS

The paper describes a new partitioned method PARTITION BY REFERENCE introduced in Oracle Database 11gR1 to equi-partition tables connected by a parent child referential constraint. This method simplifies the manageability of related tables by automatically and atomically cascading PMOs. In addition, update to parent table causing row migration is also implicitly cascaded. This partitioning method can be used to create reference partitioned tables with respect to parent tables that are partitioned using single or composite partitioning methods. They are ideally suited for supporting ILM applications. The cascading PMOs feature makes it suitable for coarse grain ILM applications, whereas partition key update based cascaded row migration makes it suitable for fine grain ILM applications.

A performance study conducted using TPC-H `ORDERS` and `LINEITEM` tables show that for all PMOs, except the SPLIT operation, the performance is similar to the configuration of individually partitioning the child and parent tables. The study also demonstrated the storage benefits of reference partitioned tables, and illustrated the applicability of partition-pruning and partition-wise joins to reference partitioned tables for better query performance.

In future, we plan to extend cascading PMO support to MOVE, EXCHANGE, and TRUNCATE operations, and optimize cascaded row migration for cases when a large number of rows are updated.

7. ACKNOWLEDGMENTS

We thank Jay Banerjee for his encouragement and support.

8. REFERENCES

[1] Agrawal, S., Narasayya, V.R., and Yang, B. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. *SIGMOD* 2004, 359-370.

[2] Navathe, S.B. and Ra, M. Vertical Partitioning for Database Design: A Graphical Algorithm. *SIGMOD* 1989, 440-450.

[3] Abadi, D.J., Marcus, A., Madden, S., and Hollenbach, K.J. Scalable Semantic Web Data Management Using Vertical Partitioning. *VLDB 2007*, 411-422.

[4] Ceri, S., Negri, M., and Pelagatti, G. Horizontal Data Partitioning in Database Design. *SIGMOD* 1982, 128-136.

[5] Shin, D.G. and Irani, K.B. Partitioning a Relational Database Horizontally Using a Knowledge-Based Approach. *SIGMOD* 1985, 95-105.

[6] Reiner, D., Press, G., Lenaghan, M., Barta, D., and Urmston, R. Information Lifecycle Management: The EMC Perspective. *ICDE 2004*, 804-807.

[7] Stonebraker, M. et al. C-Store: A Column-oriented DBMS. *VLDB 2005*, 553-564.

[8] Tallon, P.P and Scannell, R. Information Life Cycle Management. *Communications of the ACM*, Nov. 2007, 65-69.

[9] TPC-H Benchmark, http://www.tpc.org/tpch/.

[10] Oracle 11gR1, Oracle Corporation, http://www.oracle.com/technology/products/database/oracle11g/index.html.

[11] Hobbs, L. Information Lifecycle Management for Business Data, *Oracle White Paper*, June. 2007, http://www.oracle.com/technology/deploy/ilm/pdf/ILM_for_Business_11g.pdf.

[12] http://msdn2.microsoft.com/en-us/library/ms345146.aspx.

[13] http://dev.mysql.com/doc/refman/5.1/en/partitioning-types.html.

[14] http://www.postgresql.org/docs/8.1/interactive/ddl-partitioning.html.

[15] http://www.ibm.com/developerworks/db2/library/techarticle/dm-0608mcinerney/.

[16] http://www.oracle.com/technology/products/bi/db/11g/pdf/partitioning-11g-datasheet.pdf.