ORACLE®

Get the best out of Oracle Partitioning

A practical guide and reference

Version 4.0

ORACLE

Table of Contents

ORACLE

Partitioning Summary

What is Oracle Partitioning?

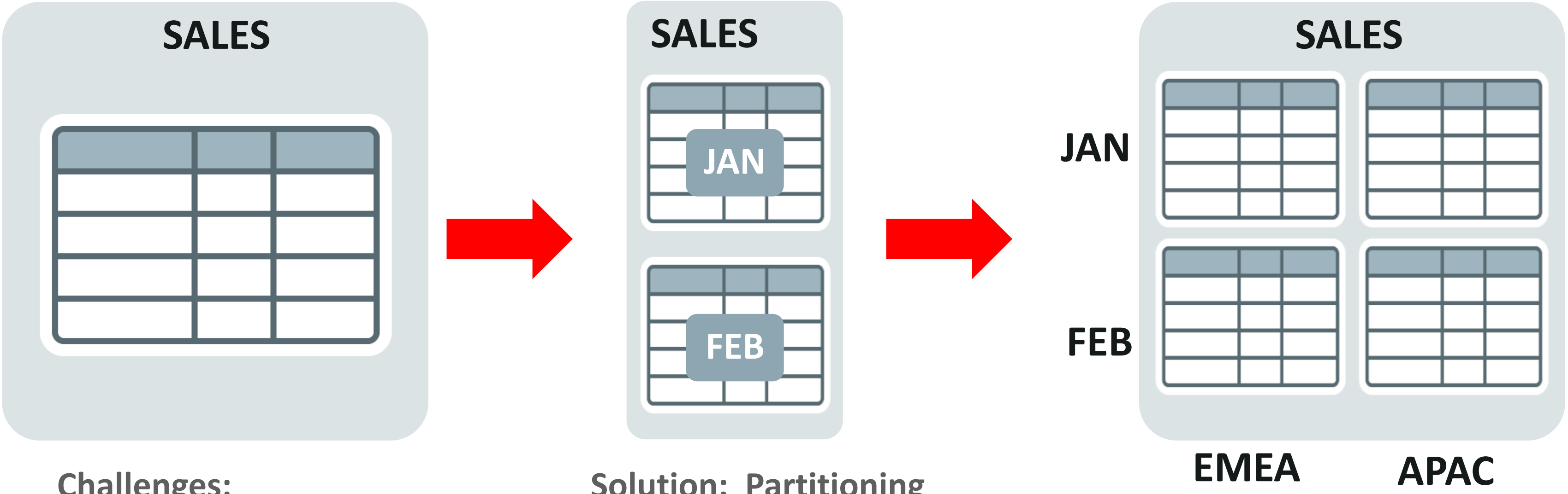
- Powerful functionality to logically divide objects into smaller pieces
- Key requirement for large databases needing high performance and high availability
- Driven by business requirements

Why use Oracle Partitioning?

- Performance – lowers data access times
- Availability – improves access to critical information
- Costs – leverages multiple storage tiers
- Easy Implementation – requires no changes to applications and queries
- Mature Feature – supports a wide array of partitioning methods
- Well Proven – used by thousands of Oracle customers

How does Partitioning work?

Enables large databases and indexes to be split into smaller, more manageable pieces



Challenges:
Large tables are difficult to manage

Solution: Partitioning
- Divide and conquer
- Easier data management
- Improve performance

Partitioning Benefits

Increased Performance

Only work on the data that is relevant

Partitioning enables data management operations such as…

- Data loads, joins and pruning,
- Index creation and rebuilding,
- Backup and recovery

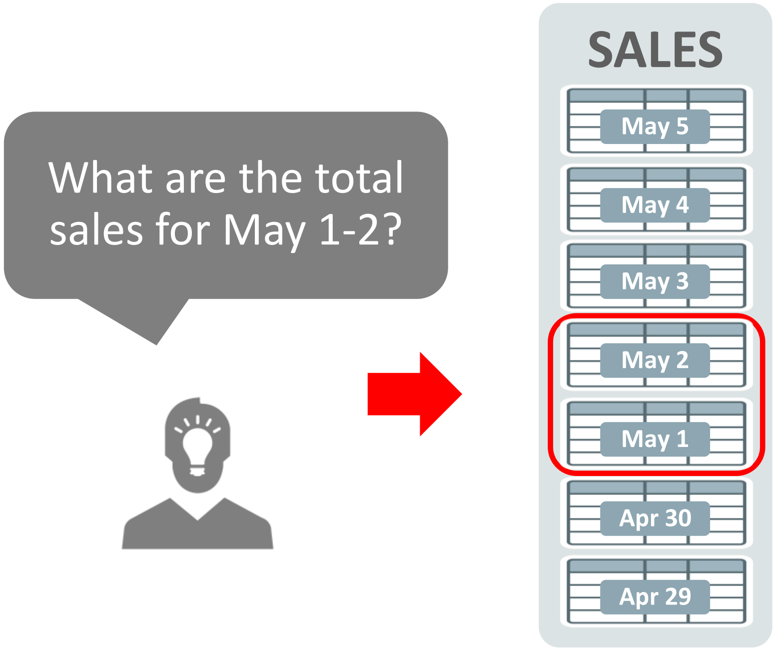
At partition level instead of on the entire table

Result: Order of magnitude gains on performance

ORACLE

Increased Performance - Example

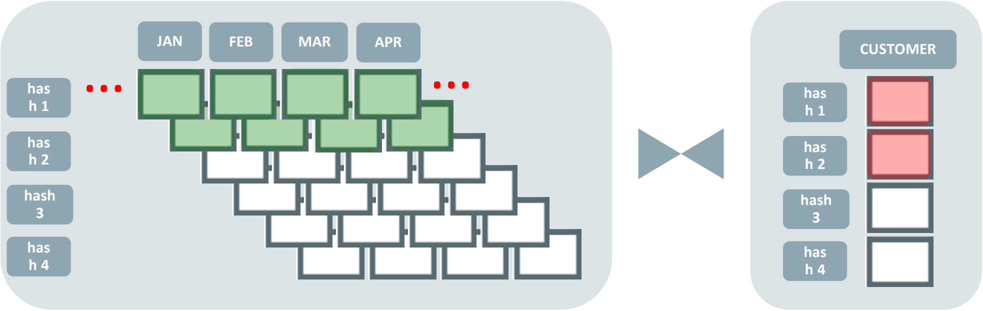
Partition Pruning

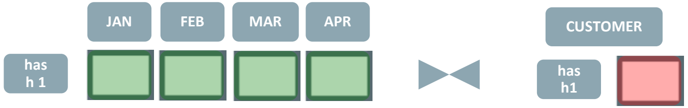


- Partition elimination
  - Dramatically reduces amount of data retrieved from storage
  - Performs operations only on relevant partitions
  - Transparently improves query performance and optimizes resource utilization

Increased Performance - Example

Partition-wise joins





- A large join is divided into multiple smaller joins, executed in parallel
  - # of partitions to join must be a multiple of DOP
  - Both tables must be partitioned the same way on the join column

Decreased Costs

Store data in the most appropriate manner

Partitioning finds the balance between…

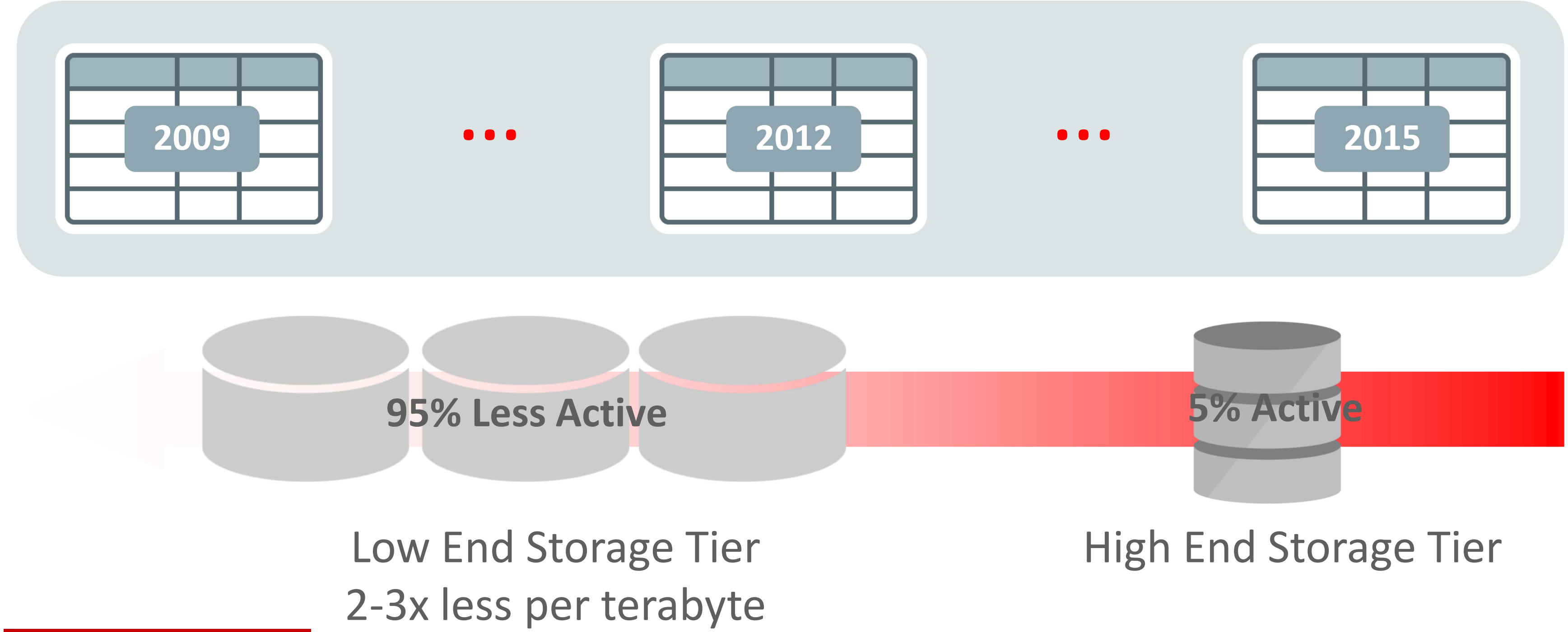
- data importance,
- storage performance,
- storage reliability,
- storage form

… allowing you to leverage multiple storage tiers

Result: Reduce storage costs by 2x or more

Decreased Costs - Example

Partition for Tiered Storage



Increased Availability

Individual partition manageability

Partitioning reduces…

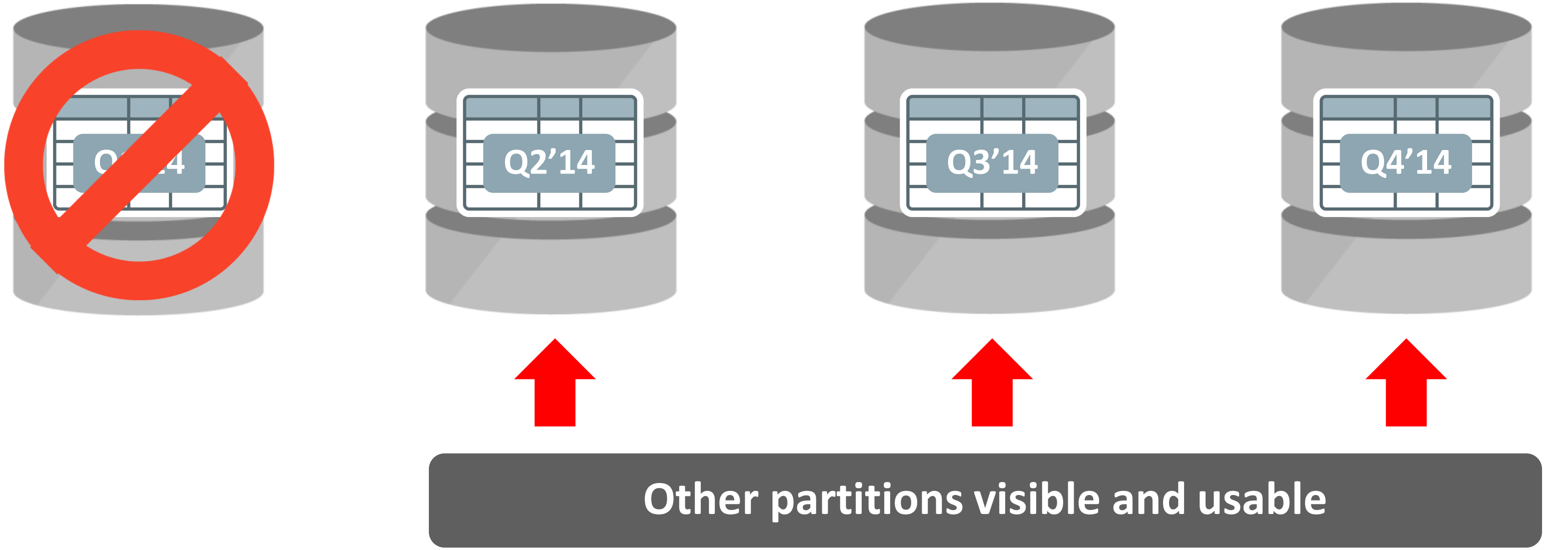
- Maintenance windows
- Impact of scheduled downtime and failures,
- Recovery times

… if critical tables and indexes are partitioned

Result: Improves access to critical information

Increased Availability - Example

Partition for Manageability/Availability



Easy Implementation

Transparent to applications

- Partitioning requires NO changes to applications and queries

Mature, Well Proven Functionality

Over a decade of development

- Used by tens of thousands of Oracle customers
- Supports a wide array of partitioning methods

Oracle Partitioning

Over a decade of development

	Core functionality	Performance	Manageability
Oracle 8.0	Range partitioning Local and global Range indexing	Static partition pruning	Basic maintenance: ADD, DROP, EXCHANGE
Oracle 8*i*	Hash partitioning Range-Hash partitioning	Partition-wise joins Dynamic partition pruning	Expanded maintenance: MERGE
Oracle 9*i*	List partitioning		Global index maintenance
Oracle 9*i* R2	Range-List partitioning	Fast partition SPLIT	
Oracle 10*g*	Global Hash indexing		Local Index maintenance
Oracle 10*g* R2	1M partitions per table	Multi-dimensional pruning	Fast DROP TABLE
Oracle 11g	Virtual column based partitioning More composite choices Reference partitioning		Interval partitioning Partition Advisor Incremental stats mgmt
Oracle 11g R2	Hash-* partitioning Expanded Reference partitioning	"AND" pruning	Multi-branch execution (aka table or-expansion)
Oracle 12c R1	Interval-Reference partitioning	Partition Maintenance on multiple partitions Asynchronous global index maintenance	Online partition MOVE Cascading TRUNCATE Partial indexing

ORACLE

Partitioning Concepts

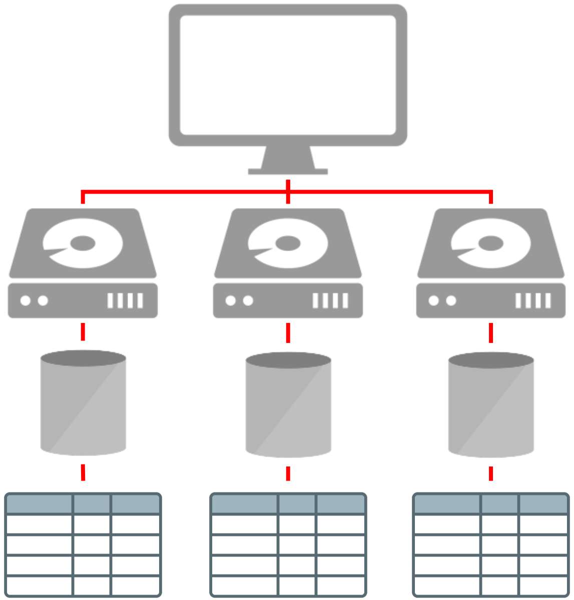
def **Par•ti•tion**

To divide (something) into parts

– "Miriam Webster Dictionary"

Physical Partitioning

Shared Nothing Architecture

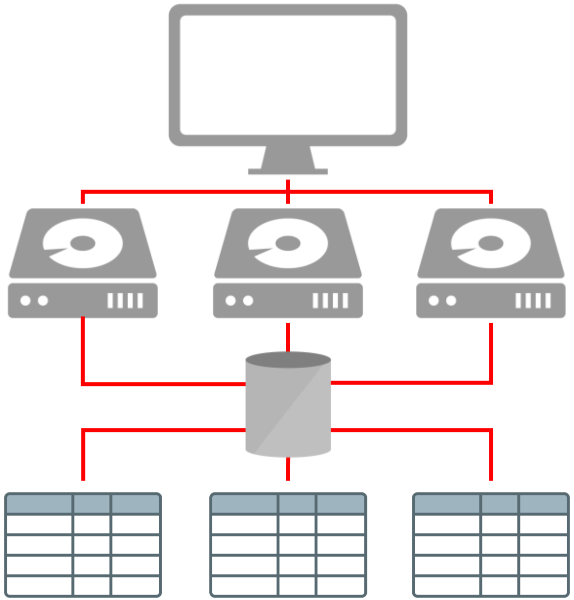


- Fundamental system setup requirement
- Node owns piece of DB
- Enables parallelism
- Number of partitions is equivalent to min. parallelism
- Always needs HASH distribution
- Equally sized partitions per node required for proper load balancing

ORACLE

Logical Partitioning

Shared Everything Architecture - Oracle

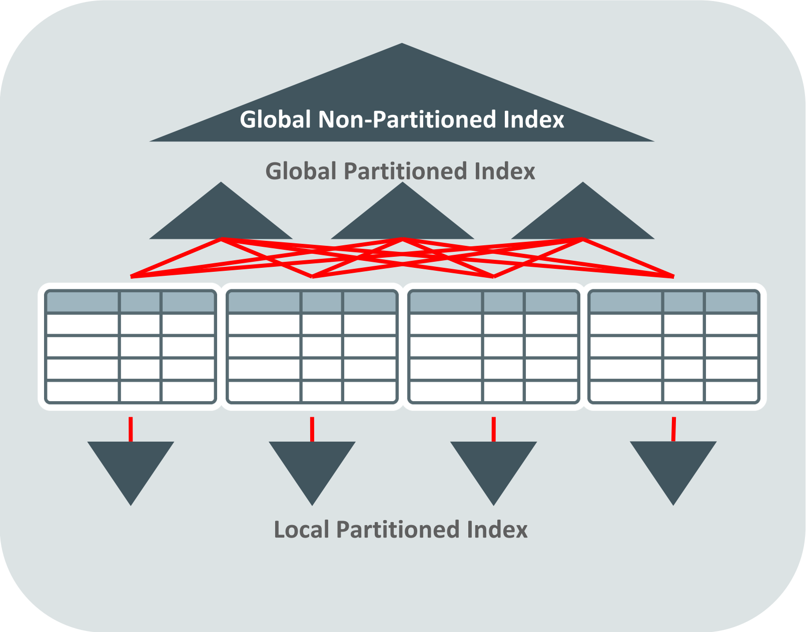


- Does not underlie any constraints
  - SMP, MPP, Cluster, Grid does not matter
- Purely based on the business requirement
  - Availability, Manageability, Performance
- Beneficial for every environment
  - Provides the most comprehensive functionality

Partitioning Methods

What can be partitioned?

- Tables
  - Heap tables
  - Index-organized tables
- Indexes
  - Global Indexes
  - Local Indexes
- Materialized Views
- Hash Clusters



Partitioning Methods

Single-level partitioning

- Range
- List
- Hash

Composite-level partitioning

- [Range | List | Hash | Interval] – [Range | List | Hash]

Partitioning extensions

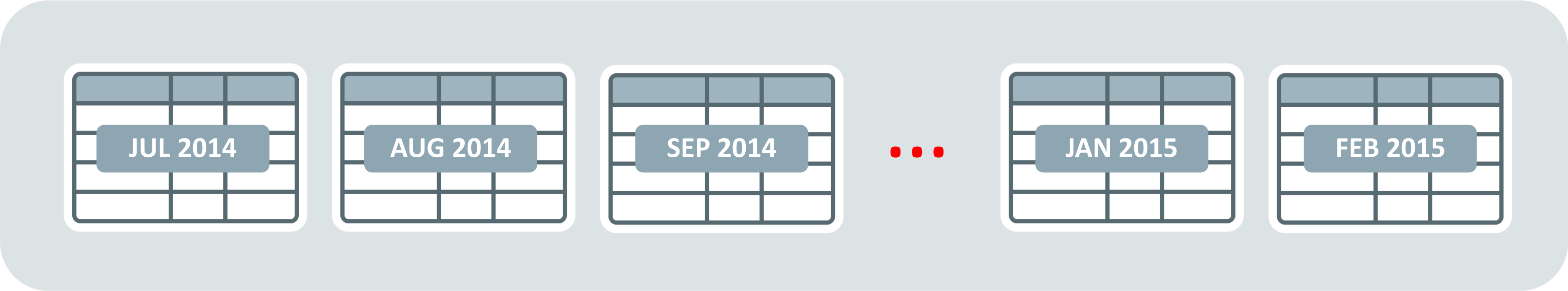
- Interval
- Reference
- Interval Reference
- Virtual Column Based

ORACLE

Range Partitioning

Introduced in Oracle 8.0

Range Partitioning

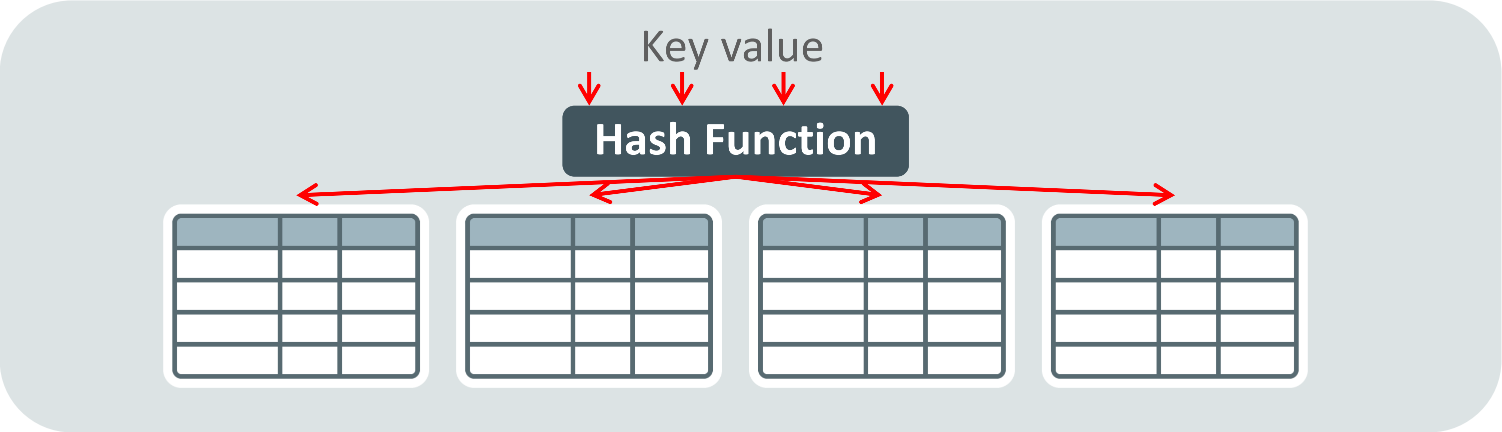


- Data is organized in ranges
  – Lower boundary derived by upper boundary of preceding partition
  – No gaps
- Ideal for chronological data

Hash Partitioning

Introduced in Oracle 8i (8.1)

Hash Partitioning

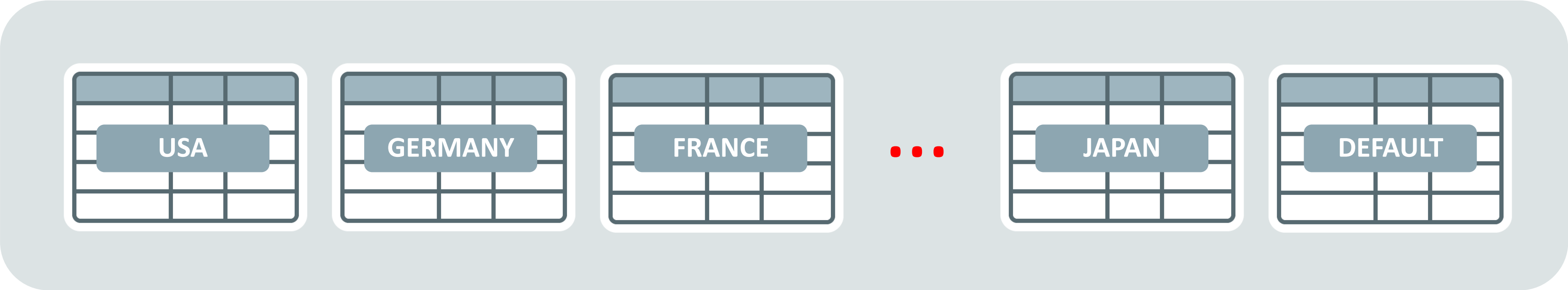


- Data is placed based on hash value of partition key
  – Number of hash buckets equals number of partitions
- Ideal for equal data distribution
  – Number of partitions should be a power of 2 for equal data distribution

List Partitioning

Introduced in Oracle 9i (9.0)

List Partitioning



- Data is organized in lists of values
  - One or more unordered distinct values per list
  - Functionality of DEFAULT partition (Catch-it-all for all unspecified values)
- Ideal for segmentation of distinct values, e.g. region

Interval Partitioning

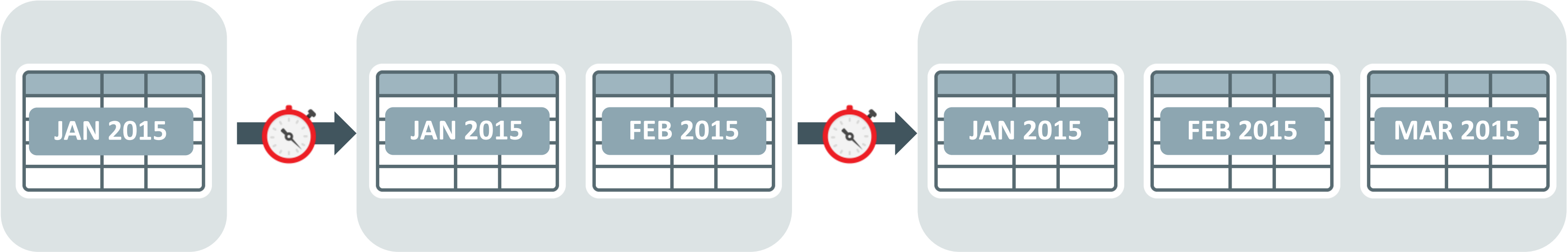
Introduced in Oracle 11g Release 1 (11.1)

Interval Partitioning

- Extension to Range Partitioning
- Full automation for equi-sized range partitions
- Partitions are created as metadata information only
  - Start Partition is made persistent
- Segments are allocated as soon as new data arrives
  - No need to create new partitions
  - Local indexes are created and maintained as well

No need for any partition management

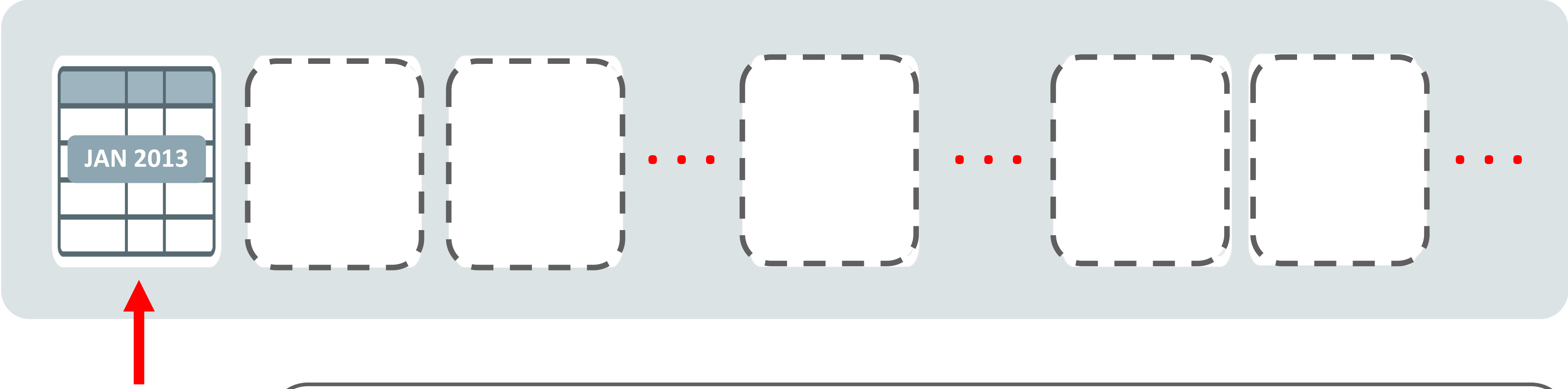
Interval Partitioning



- Partitions are created automatically as data arrives
  - Extension to RANGE partitioning

Interval Partitioning

As easy as One, Two, Three...

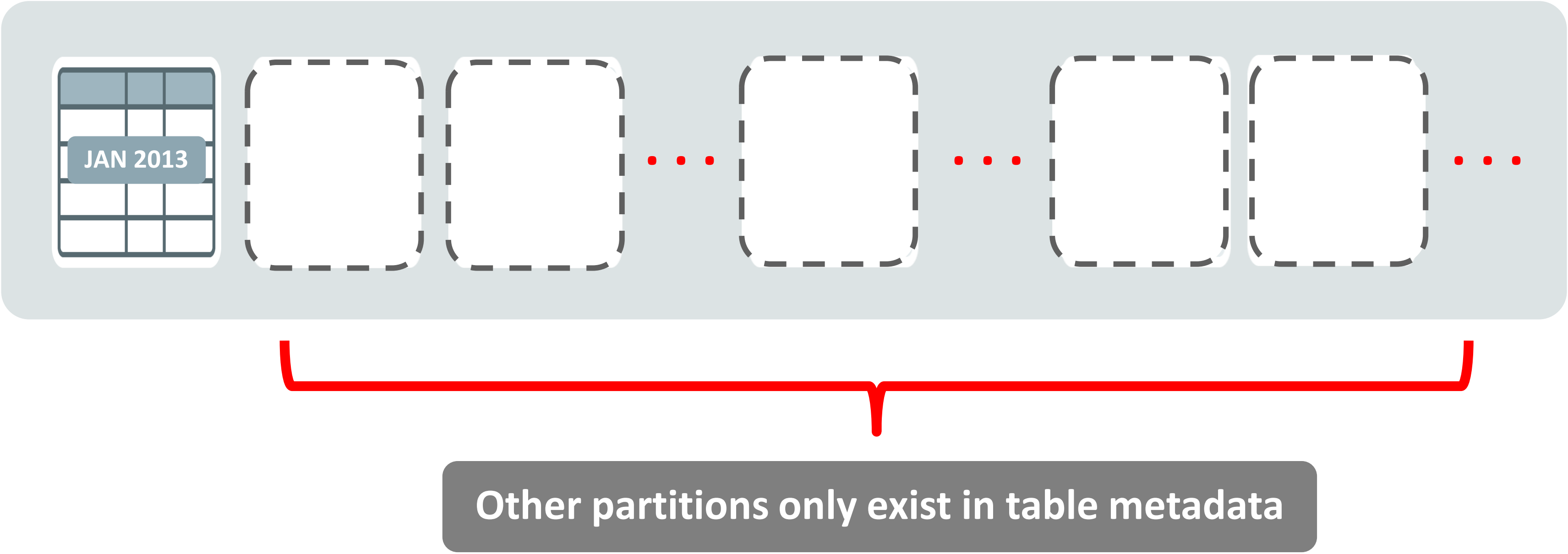


First partition is created

```
CREATE TABLE sales (order_date DATE, ...)
PARTITON BY RANGE (order_date)
INTERVAL(NUMTOYMINTERVAL(1,'month')
(PARTITION p_first VALUES LESS THAN ('01-JAN-2013');
```

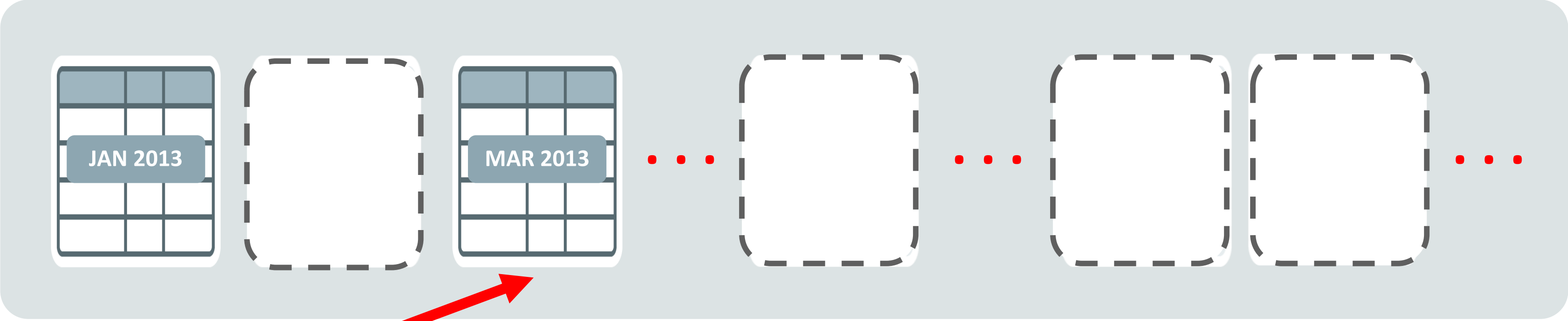
Interval Partitioning

As easy as One, Two, Three…



Interval Partitioning

As easy as One, Two, Three…

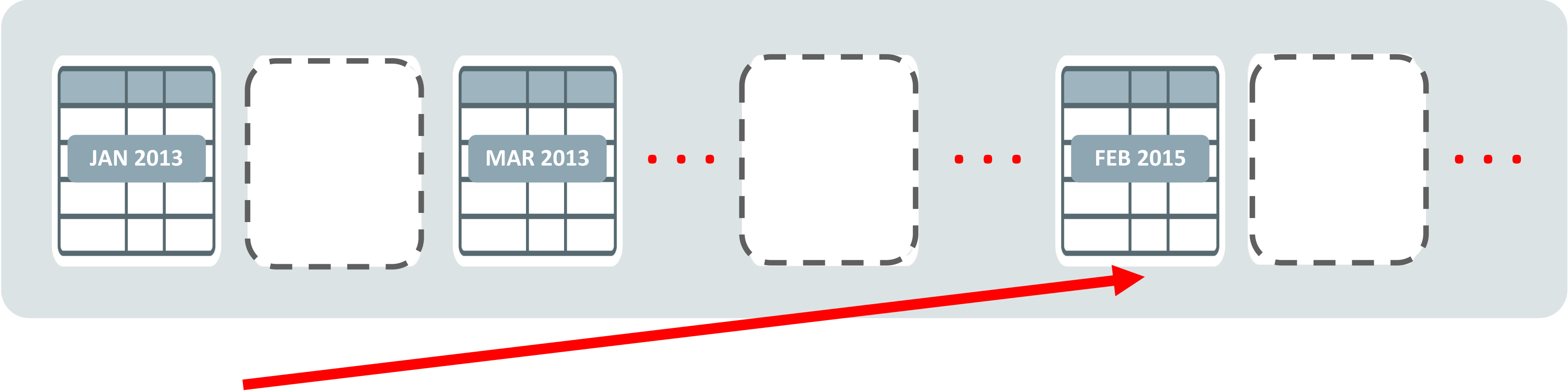


New partition is automatically instantiated

```
INSERT INTO sales (order_date DATE, ...)
VALUES ('30-MAR-2013',...);
```

Interval Partitioning

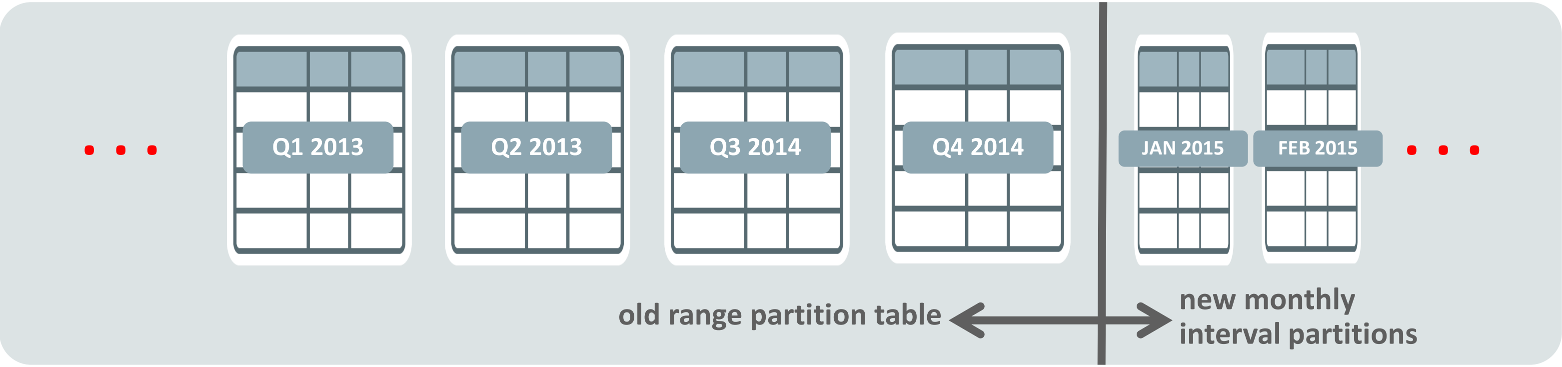
As easy as One, Two, Three...



Whenever data for a new partition arrives

```
INSERT INTO sales (order_date DATE, ...)
VALUES (’04-FEB-2015',...);
```

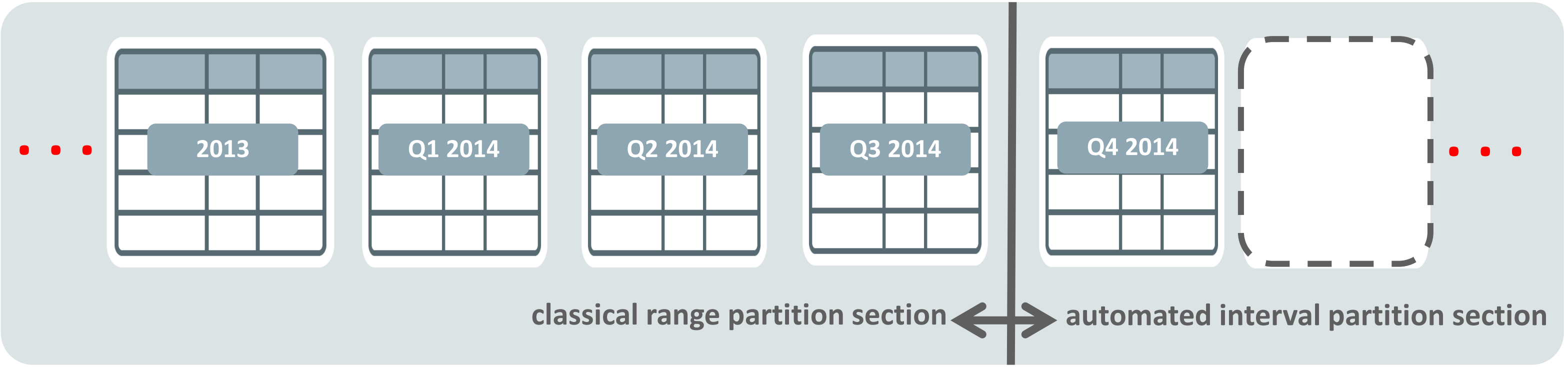
Interval Partitioning



- Range partitioned tables can be extended into interval partitioned tables
  - Simple metadata command
  - Investment protection

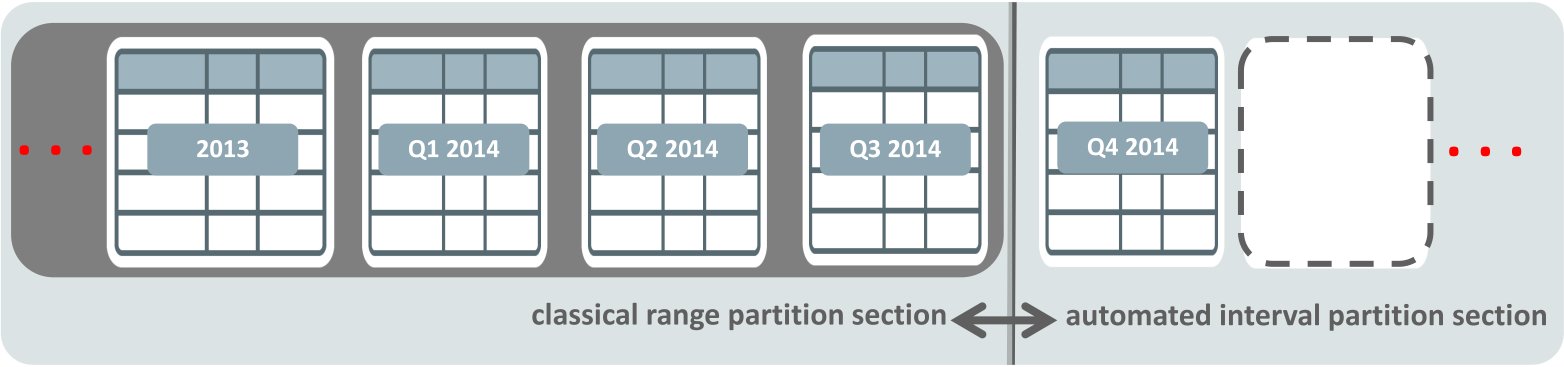
```
ALTER TABLE sales
SET INTERVAL(NUMTOYMINTERVAL(1,'month');
```

Interval Partitioning



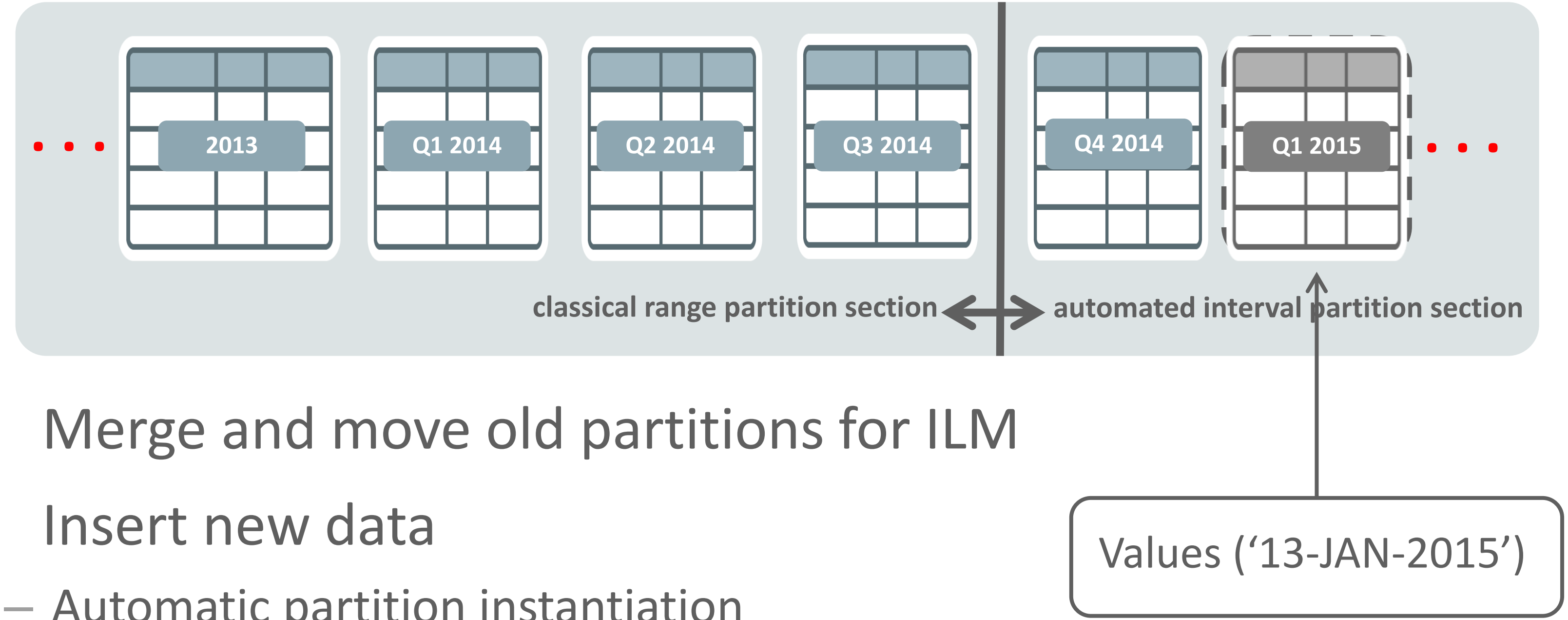
- Interval partitioned table has classical range and automated interval section
  – Automated new partition management plus full partition maintenance capabilities: ***"Best of both worlds"***

Interval Partitioning



1. Merge and move old partitions for ILM

Interval Partitioning



1. Merge and move old partitions for ILM
2. Insert new data
   – Automatic partition instantiation

Deferred Segment Creation vs Interval Partitioning

Interval Partitioning

- Maximum number of one million partitions are pre-defined
  - Explicitly defined plus interval-based partitions
- No segments are allocated for partitions without data
  - New record insertion triggers segment creation
- Ideal for “ever-growing” tables

“Standard” Partitioning with deferred segment creation

- Only explicitly defined partitions are existent
  - New partitions added via DDL
- No segments are allocated for partitions without data
  - New record insertion triggers segment creation when data matches pre-defined partitions
- Ideal for sparsely populated pre-defined tables

ORACLE

Difference Between Range and Interval

Interval Partitioning

- Full automation for equi-sized range partitions
- Partitions are created as metadata information only
  - Start Partition is made persistent
- Segments are allocated as soon as new data arrives
  - No need to create new partitions
  - Local indexes are created and maintained as well
- Interval Partitioning is almost a transparent extension to range partitioning
  - .. But interval implementation introduces some subtle differences

ORACLE

Interval versus Range Partitioning

Partition bounds

- Interval partitions have lower and upper bound
- Range partitions only have upper bounds
  - Lower bound derived by previous partition

Partition naming

- Interval partitions cannot be named in advance
  - Use the PARTITION FOR (<value>) clause
- Range partitions must be named

ORACLE

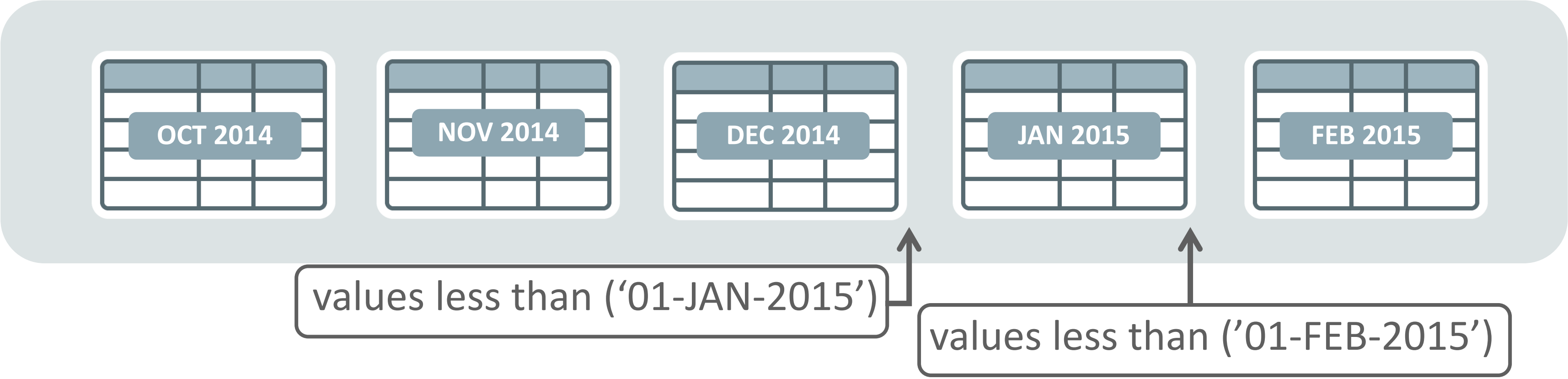
Interval versus Range Partitioning, cont.

- Partition merge
  - Multiple non-existent interval partitions are silently merged
  - Only two adjacent range partitions can be merged at any point in time
- Number of partitions
  - Interval partitioned tables have always one million partitions
    - Non-existent partitions “exist” through INTERVAL clause
    - No MAXVALUE clause for interval partitioning
      - Maximum value defined through number of partitions and INTERVAL clause
  - Range partitioning can have up to one million partitions
    - MAXVALUE clause defines most upper partition

ORACLE

Interval Versus Range Partitioning

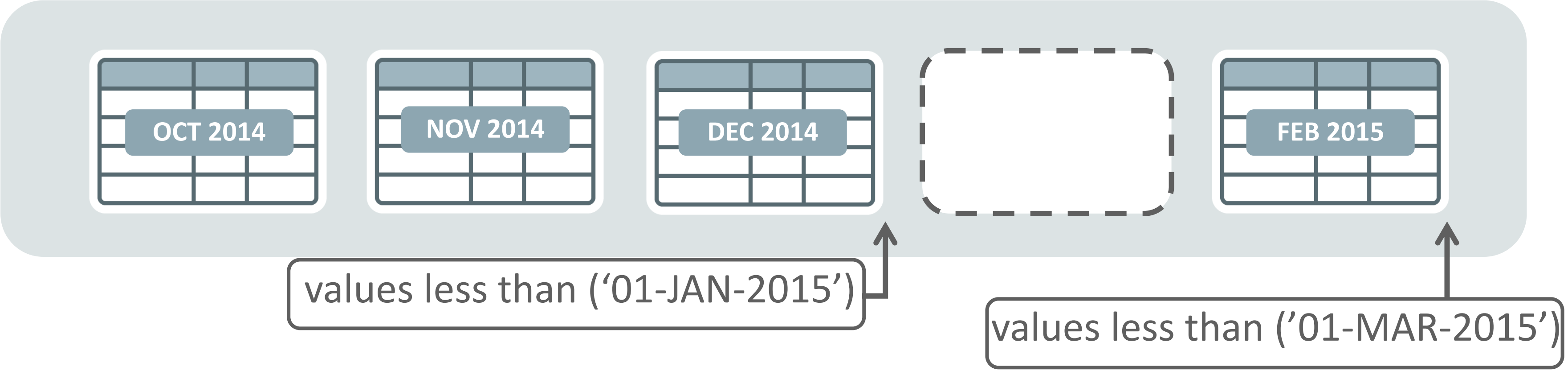
Partition Bounds for Range Partitioning



- Partitions only have upper bounds
  - Lower bound derived through upper bound of previous partition

Interval Versus Range Partitioning

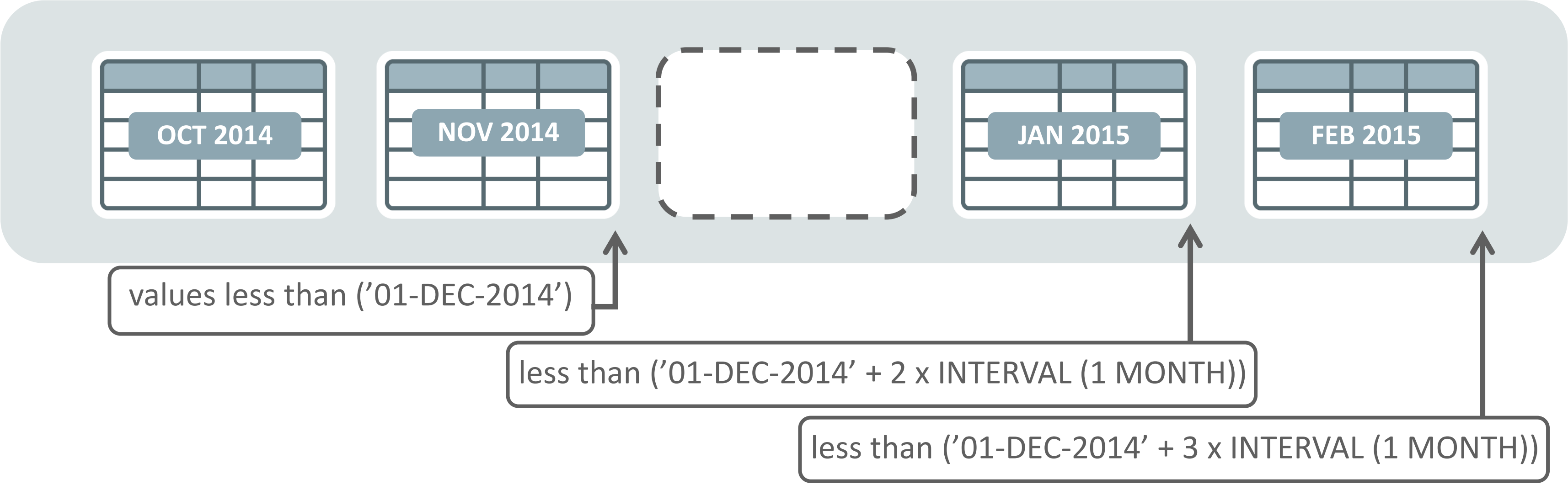
Partition Bounds for Range Partitioning



- Drop of previous partition moves lower boundary
  - "Feb 2015" now spawns 01-JAN-2015 to 30-FEB-2015

Interval Versus Range Partitioning

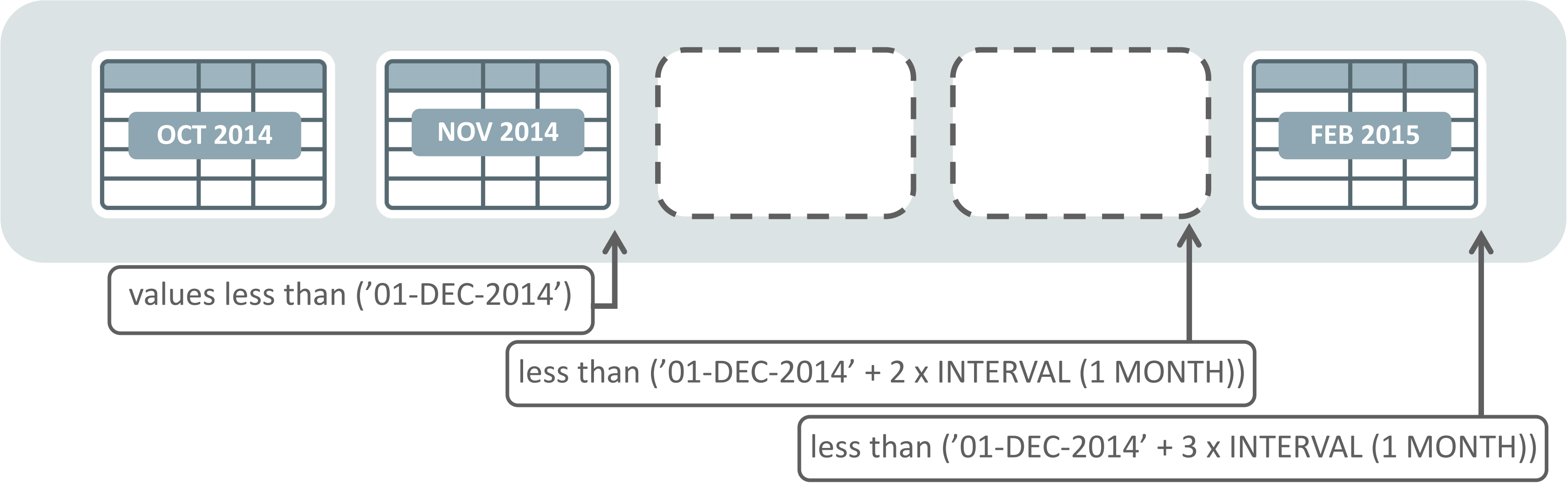
Partition Bounds for Interval Partitioning



- Partitions have upper and lower bounds
  - Derived by INTERVAL function and last range partition

Interval Versus Range Partitioning

Partition Bounds for Interval Partitioning



- Drop does not impact partition boundaries
  - "Feb 2015" still spawns 01-FEB-2015 to 30-FEB-2015

Interval versus Range Partitioning

Partition Naming

- Range partitions **can** be named
  - System generated name if not specified

```
SQL> alter table t add partition values less than(20);
Table altered.
SQL> alter table t add partition P30 values less than(30);
Table altered.
```

- Interval partitions **cannot** be named
  - Always system generated name

```
SQL> alter table t add partition values less than(20);
                  *
ERROR at line 1: ORA-14760: ADD PARTITION is not permitted
on Interval partitioned objects
```

- Use new deterministic PARTITION FOR () extension

```
SQL> alter table t1 rename partition for (9) to p_10;
Table altered.
```

Interval Versus Range Partitioning

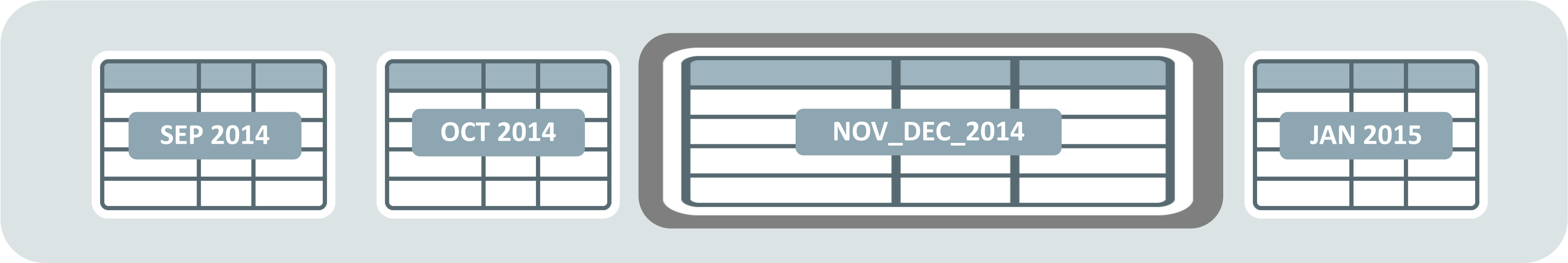
Partition Merge – Range Partitioning


```
MERGE PARTITIONS NOV_2014, DEC_2014 INTO PARTITION NOV_DEC_2014
```

- Merge two adjacent partitions for range partitioning
  - Upper bound of higher partition is new upper bound
  - Lower bound derived through upper bound of previous partition

Interval Versus Range Partitioning

Partition Merge – Range Partitioning




```
MERGE PARTITIONS NOV_2014, DEC_2014 INTO PARTITION NOV_DEC_2014
```

- New segment for merged partition is created
  - Rest of the table is unaffected

Interval Versus Range Partitioning

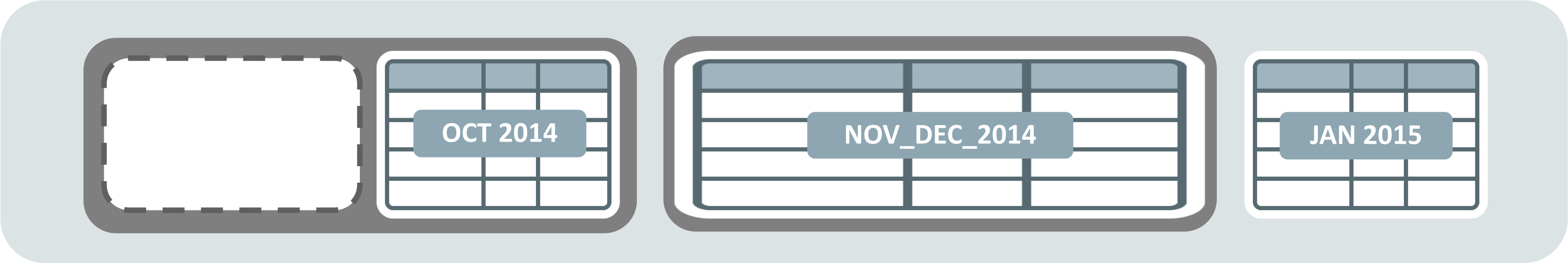
Partition Merge – Interval Partitioning


```
MERGE PARTITIONS NOV_2014, DEC_2014 INTO PARTITION NOV_DEC_2014
```

- Merge two adjacent partitions for interval partitioning
  - Upper bound of higher partition is new upper bound
  - Lower bound derived through lower bound of first partition

Interval Versus Range Partitioning

Partition Merge – Interval Partitioning



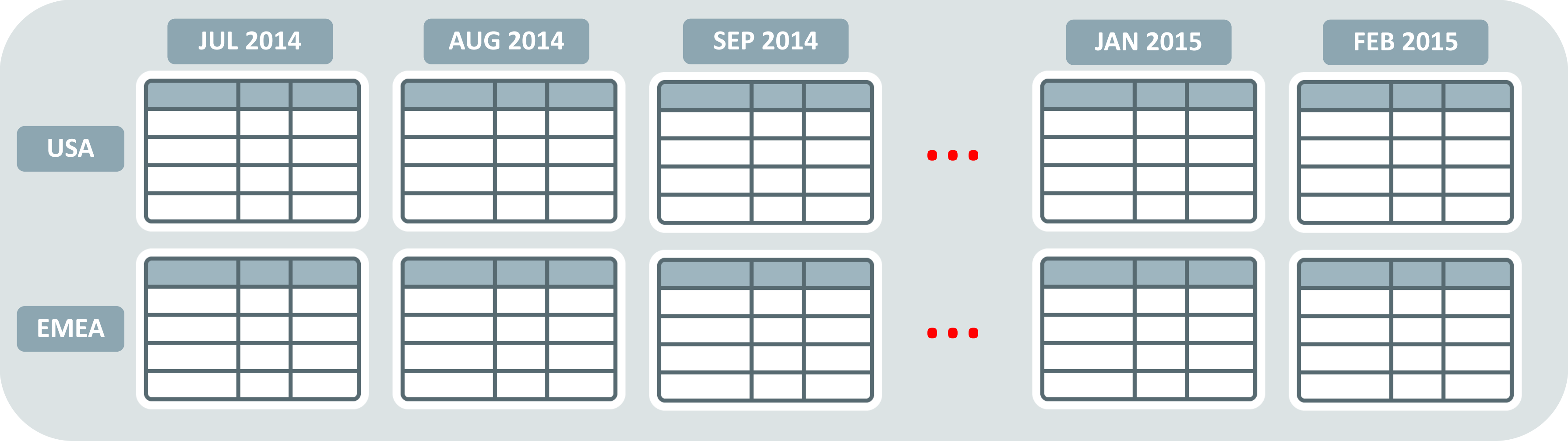

```
MERGE PARTITIONS NOV_2014, DEC_2014 INTO PARTITION NOV_DEC_2014
```

- New segment for merged partition is created
  - Holes before highest non-interval partition will be silently “merged” as well
    - Interval only valid beyond the highest non-interval partition

Composite Partitioning

Range-Hash introduced in Oracle 8i
Range-List introduced in Oracle 9i Release 2
[Range|List|Hash]-[Range|List|Hash] introduced in Oracle 11g Release 1|2
***Hash-Hash in 11.2**

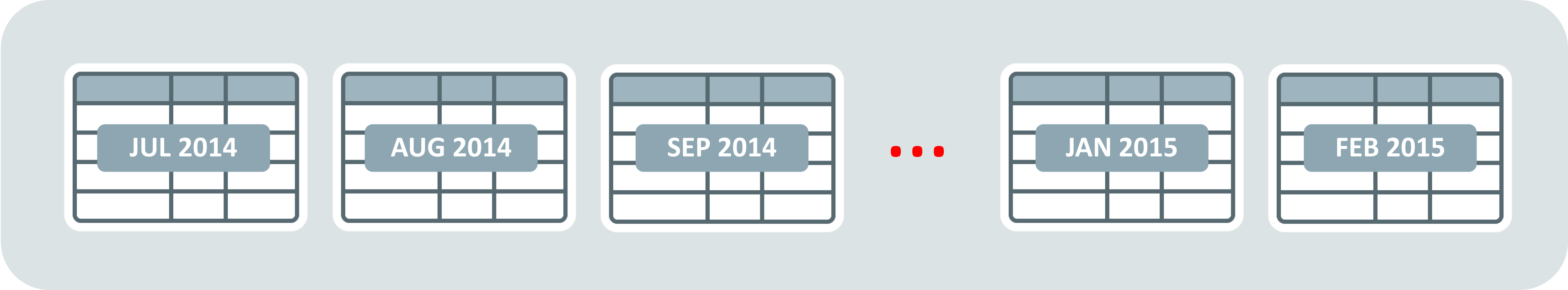
Composite Partitioning



- Data is organized along two dimensions
  - Record placement is deterministically identified by dimensions
    - Example RANGE-LIST

Composite Partitioning

Concept

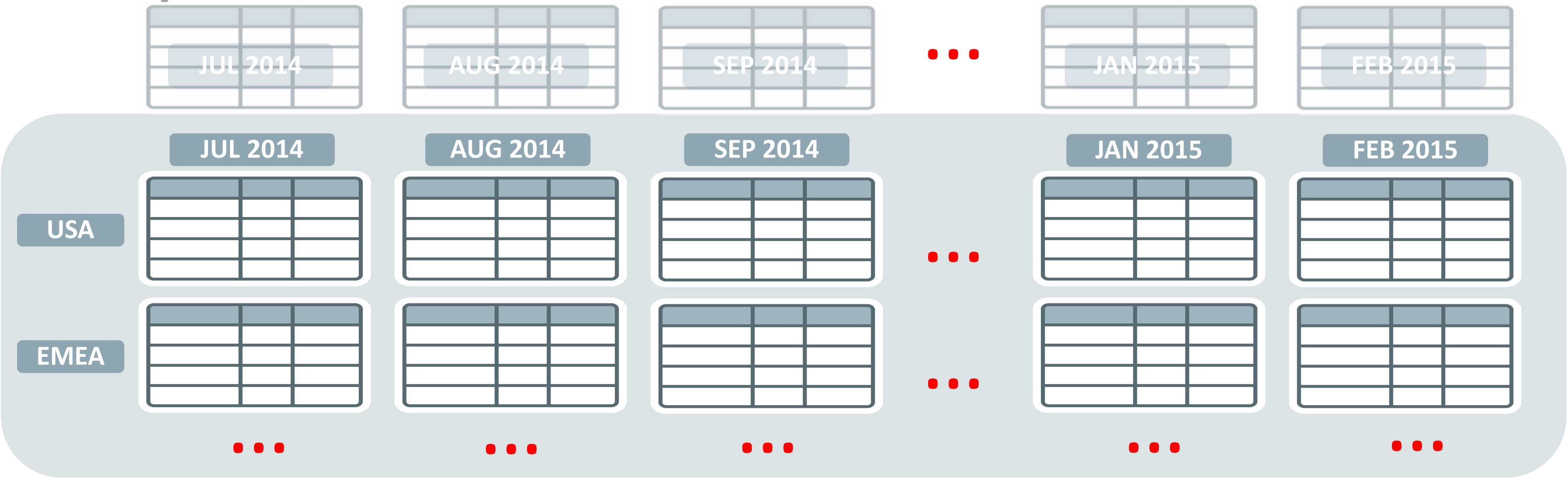



```
CREATE TABLE SALES ..PARTITION BY RANGE (time_id)
```

ORACLE

Composite Partitioning

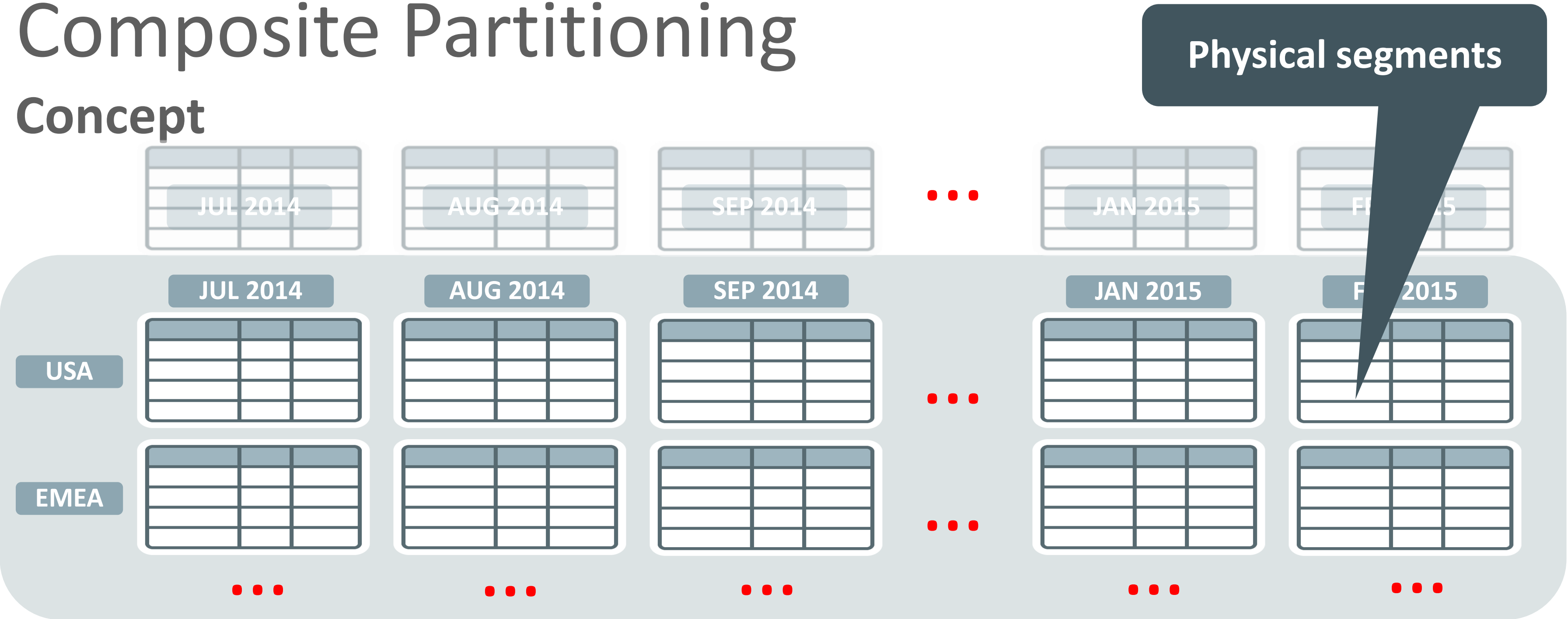
Concept




```
CREATE TABLE SALES ..PARTITION BY RANGE (time_id)
                     SUPARTITION BY LIST (region)
```

Composite Partitioning

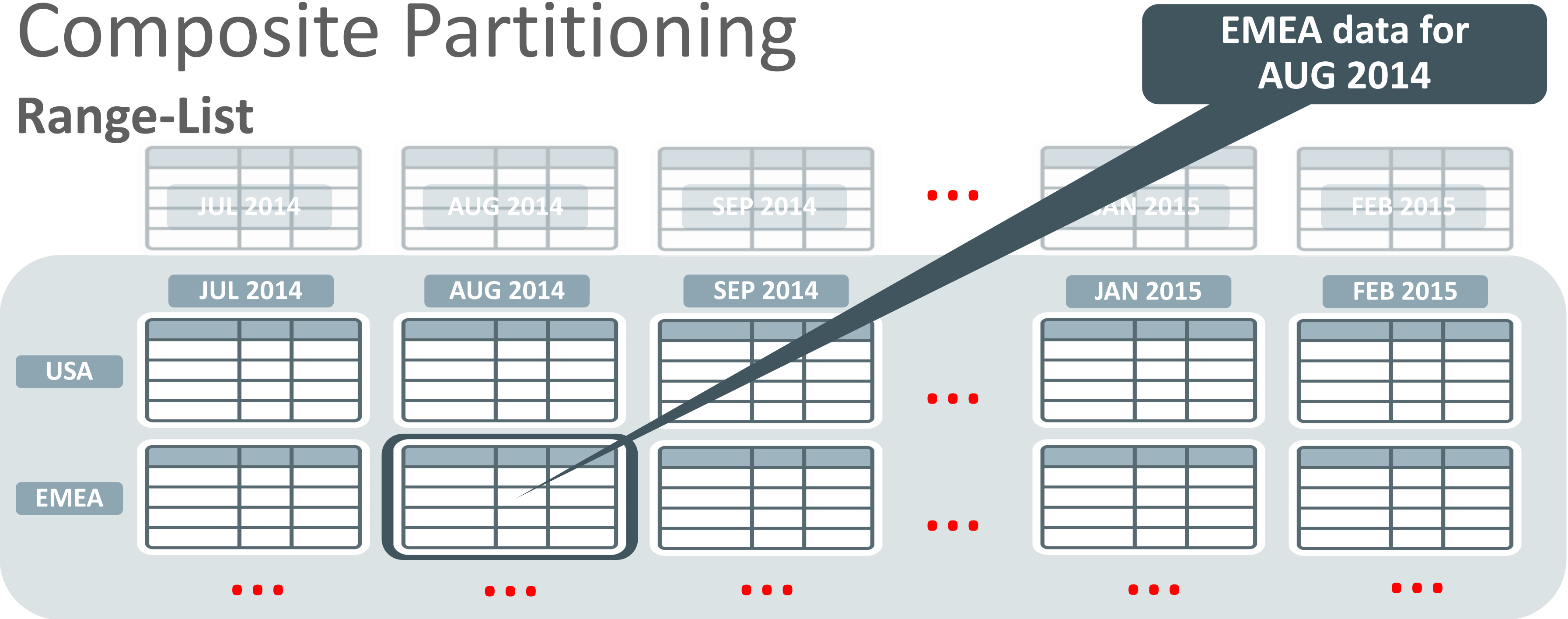
Concept




```
CREATE TABLE SALES ..PARTITION BY RANGE (time_id)
                    SUPARTITION BY LIST (region)
```

Composite Partitioning

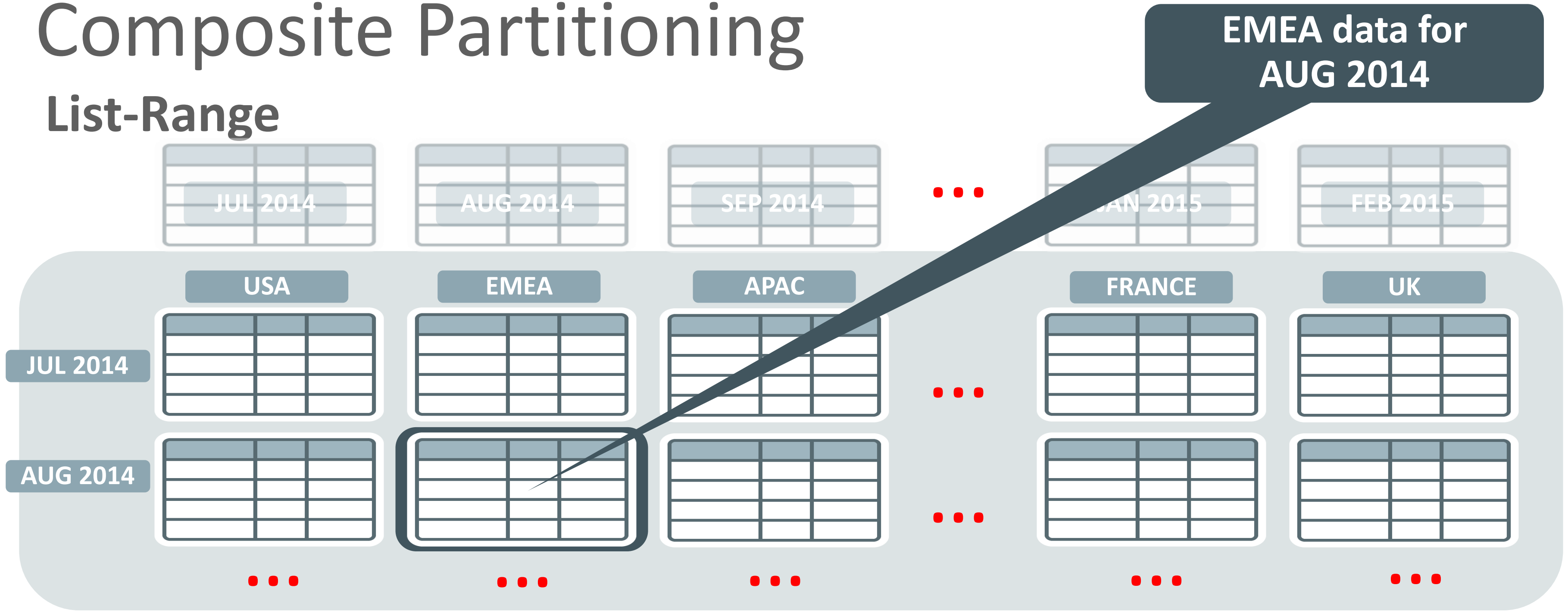
Range-List




```
CREATE TABLE SALES ..PARTITION BY RANGE (time_id)
                     SUPARTITION BY LIST (region)
```

Composite Partitioning

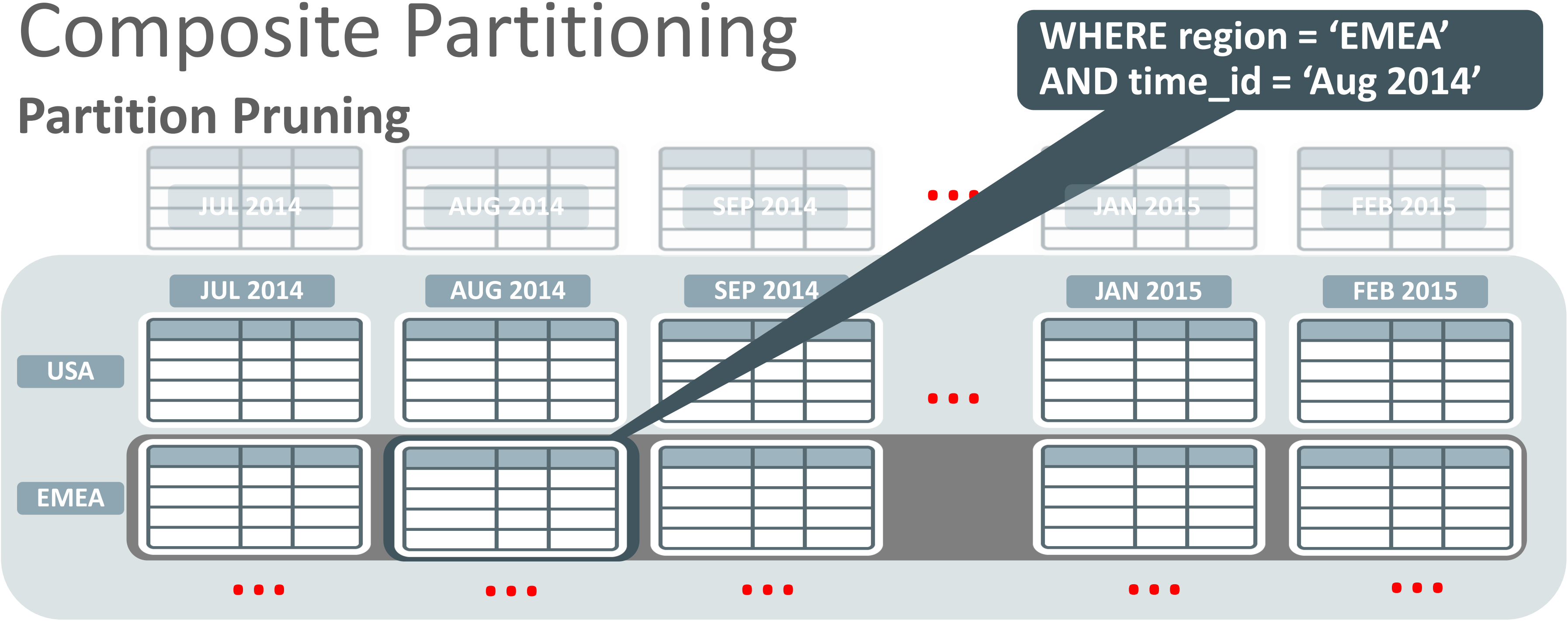
List-Range




```
CREATE TABLE SALES ..PARTITION BY LIST (region)
                     SUPARTITION BY RANGE (time_id)
```

Composite Partitioning

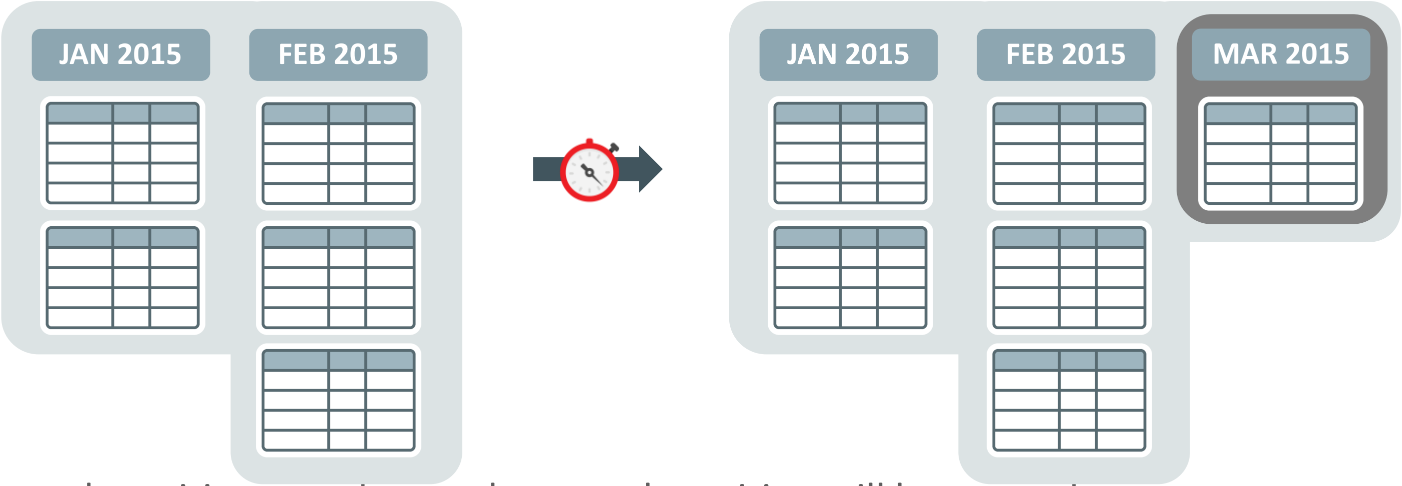
Partition Pruning



- Partition pruning is independent of composite order
  - Pruning along one or both dimensions
  - Same pruning for RANGE-LIST and LIST_RANGE

Composite Interval Partitioning

Add Partition



- Without subpartition template, only **one** subpartition will be created
  - Range: MAXVALUE
  - List: DEFAULT
  - Hash: one hash bucket

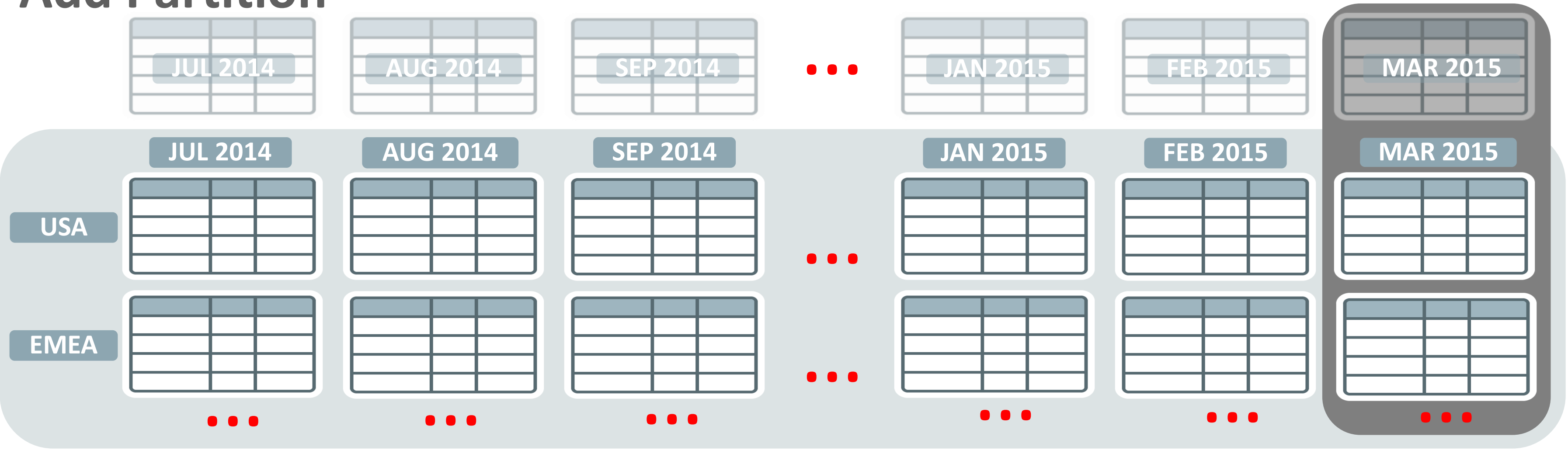
Composite Interval Partitioning

Subpartition template

- Subpartition template defines shape of **future** subpartitions
  - Can be added and/or modified at any point in time
  - No impact on existing [sub]partitions
- Controls physical attributes for subpartitions as well
  - Just like the default settings for a partitioned table does for partitions
- Difference Interval and Range Partitioning
  - Naming template only for Range
  - System-generated names for Interval

Composite Partitioning

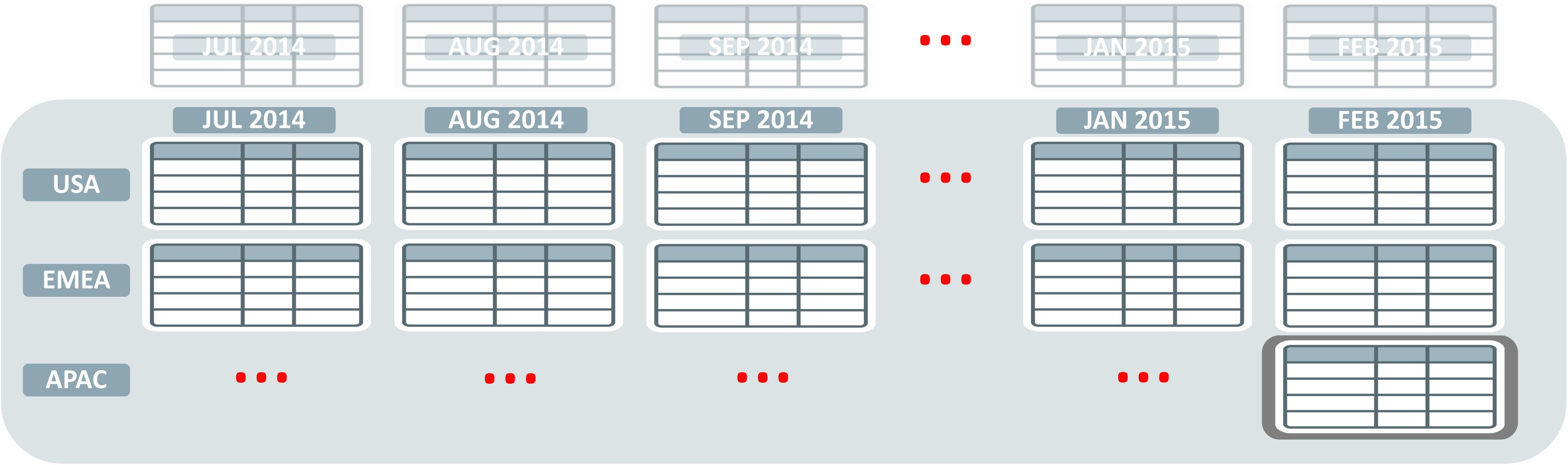
Add Partition



- ADD PARTITION always on top-level dimension
  - Identical for all newly added subpartitions
    - RANGE-LIST: new time_id range
    - LIST-RANGE: new list of region values

Composite Partitioning

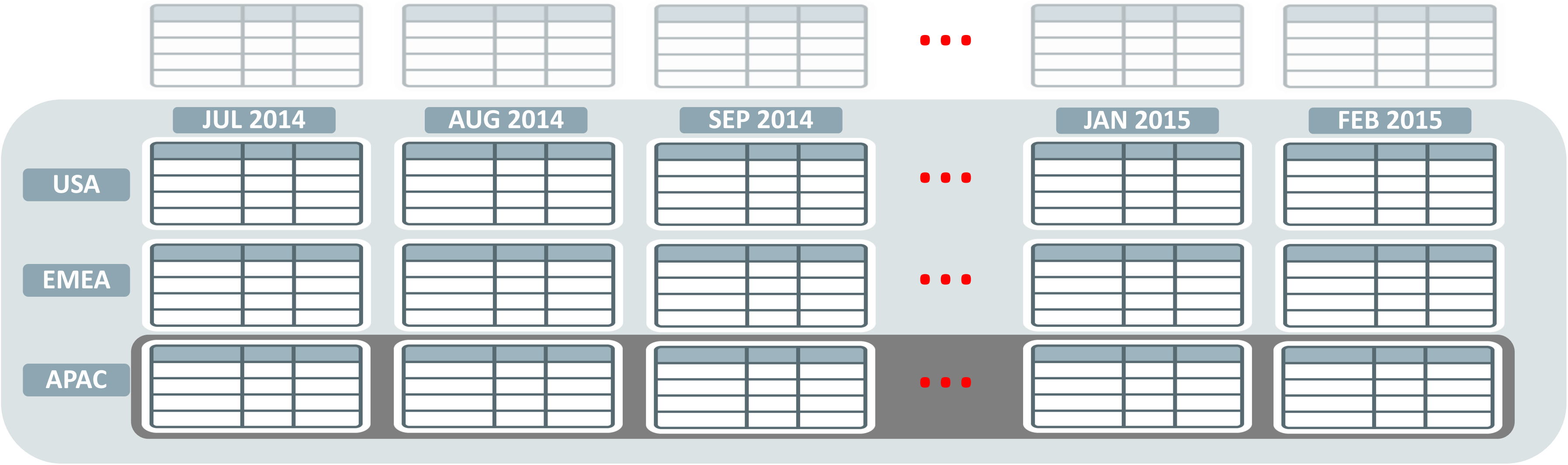
Add Subpartition



- ADD SUBPARTITION only for one partition
  - Asymmetric, only possible on subpartition level
  - Impact on partition-wise joins

Composite Partitioning

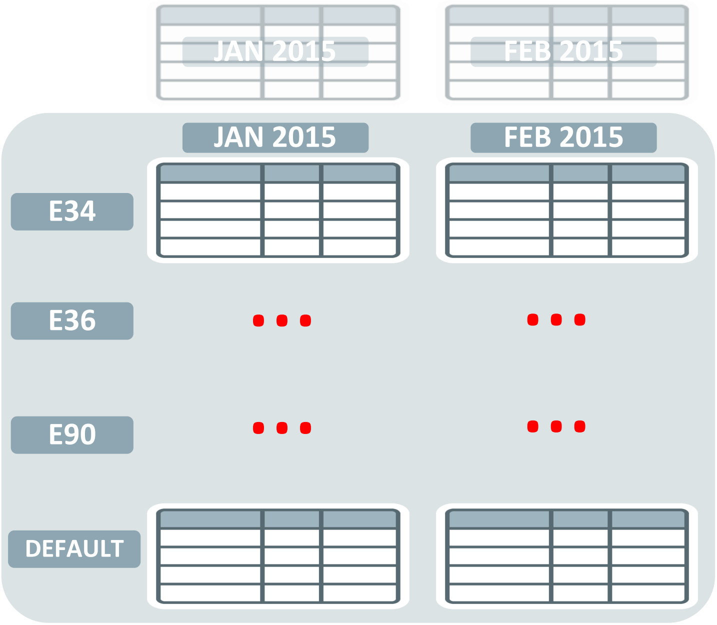
Add Subpartition



- ADD SUBPARTITION for all partitions
  - N operations necessary (for each existing partition)
  - Adjust subpartition template for future partitions

Composite Partitioning

Asymmetric subpartitions



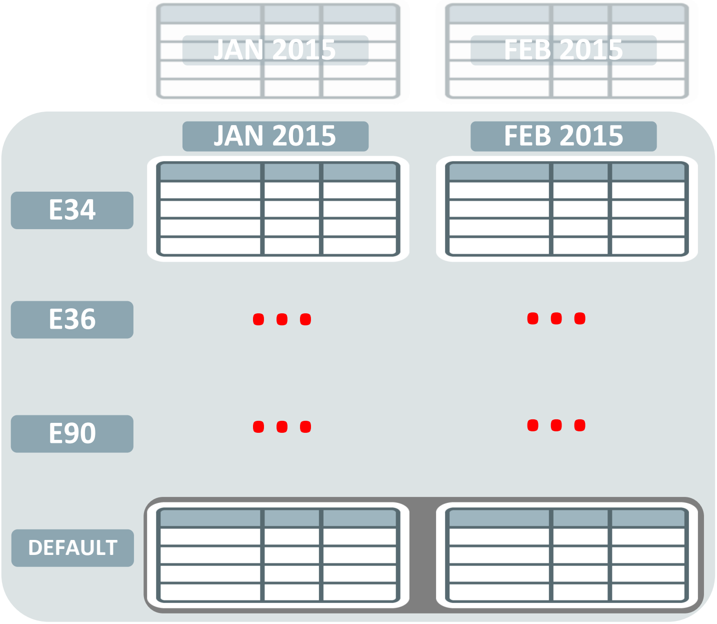
- Number of subpartitions varies for individual partitions
  - Most common for LIST subpartition strategies

```
CREATE TABLE CARS..
PARTITION BY RANGE (time_id)
SUPARTITION BY LIST (model)
```

ORACLE

Composite Partitioning

Asymmetric subpartitions



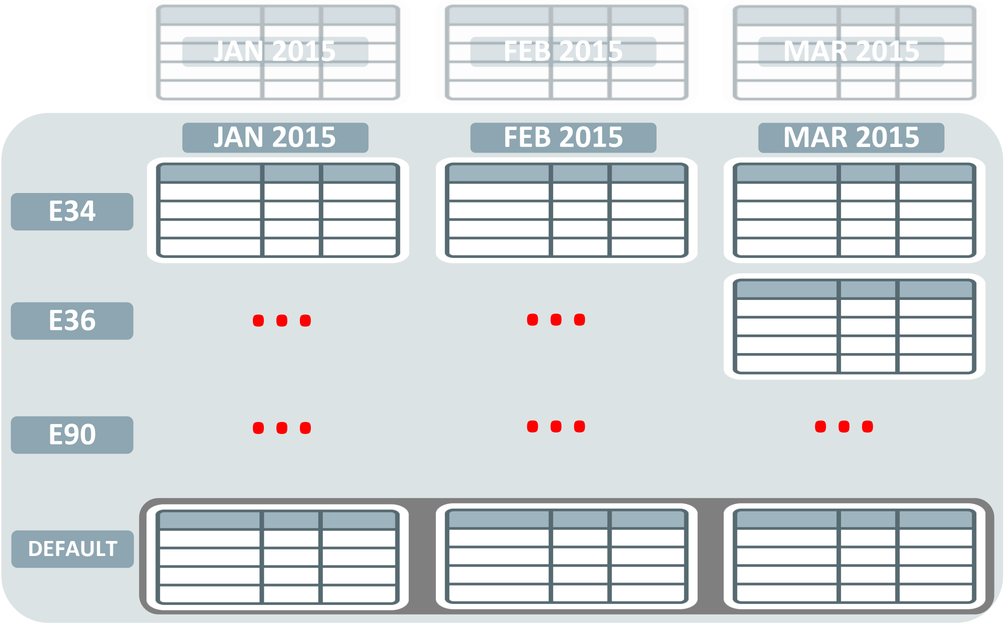
- Number of subpartitions varies for individual partitions
  - Most common for LIST subpartition strategies
- Zero impact on partition pruning capabilities

```
SELECT .. FROM cars
WHERE model = 'E90';
```

ORACLE

Composite Partitioning

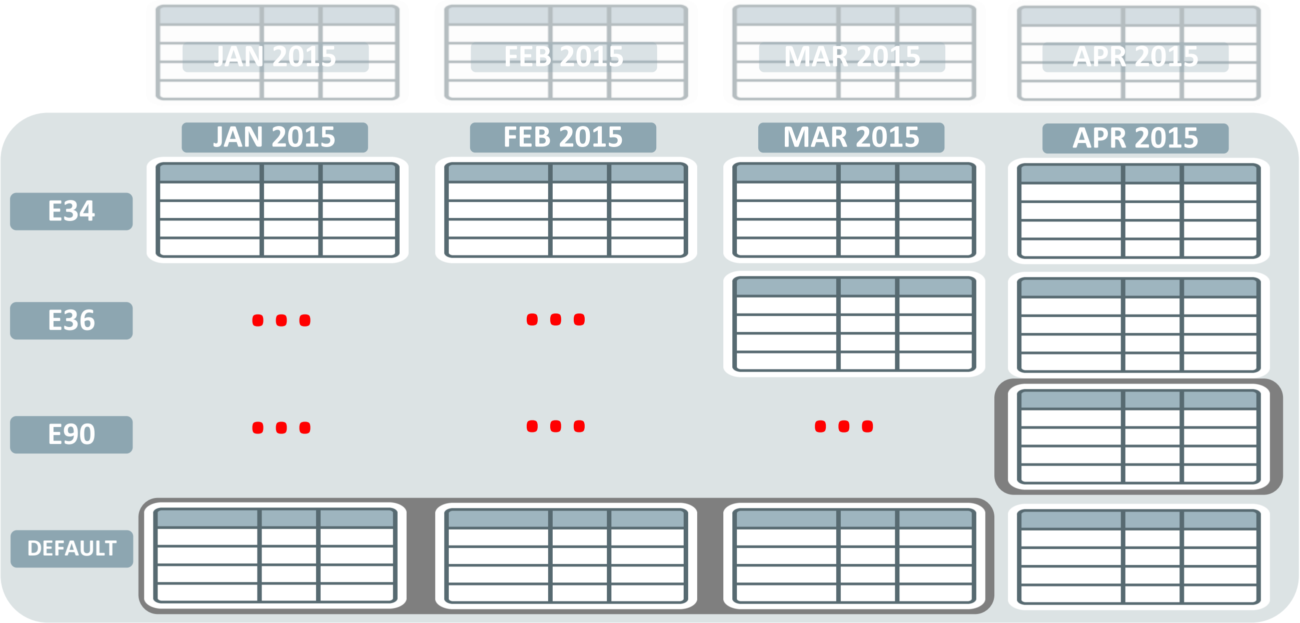
Asymmetric subpartitions




```
SELECT .. FROM cars
WHERE model = 'E90';
```

Composite Partitioning

Asymmetric subpartitions




```
SELECT .. FROM cars
WHERE model = 'E90';
```

Composite Partitioning

- Always use appropriate composite strategy
  - Top-level dimension mainly chosen for Manageability
    - E.g. add and drop time ranges
  - Sub-level dimension chosen for performance or manageability
    - E.g. load_id, customer_id
  - Asymmetry has advantages but should be thought through
    - E.g. different time granularity for different regions
    - Remember the impact of asymmetric composite partitioning

ORACLE

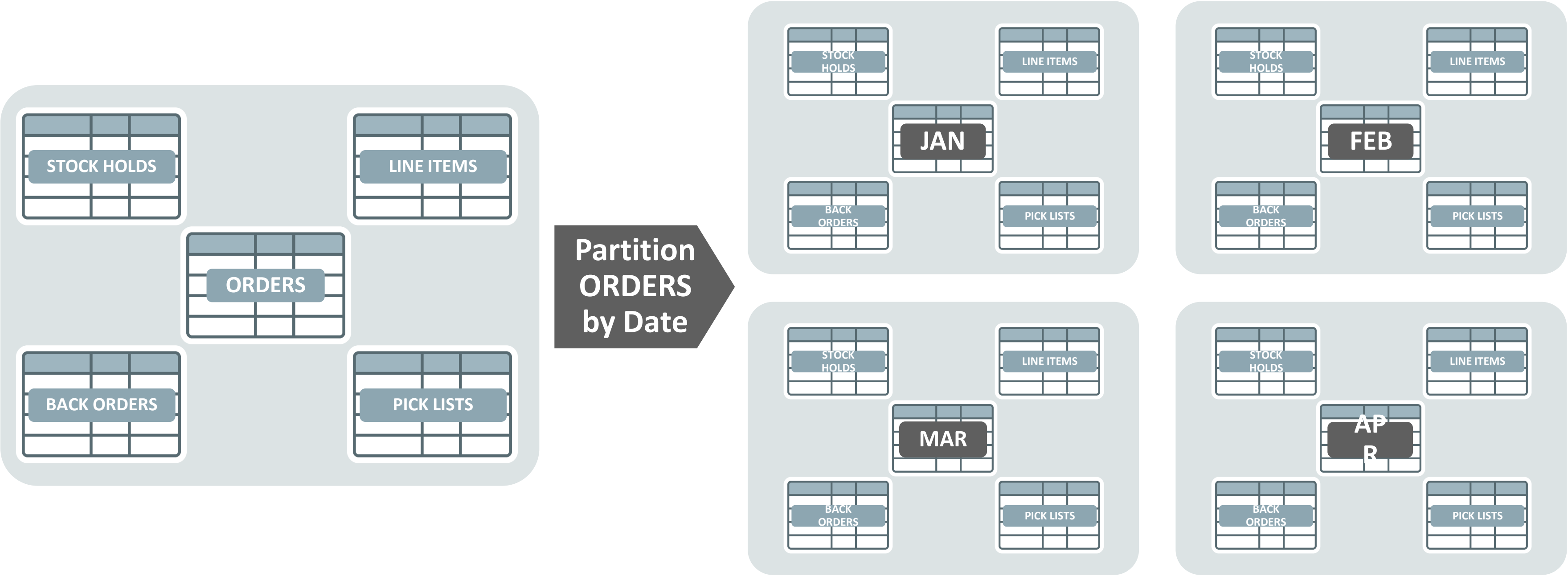
Reference Partitioning

Introduced in Oracle 11g Release 1 (11.1)

ORACLE

Reference Partitioning

Inherit partitioning strategy



Reference Partitioning

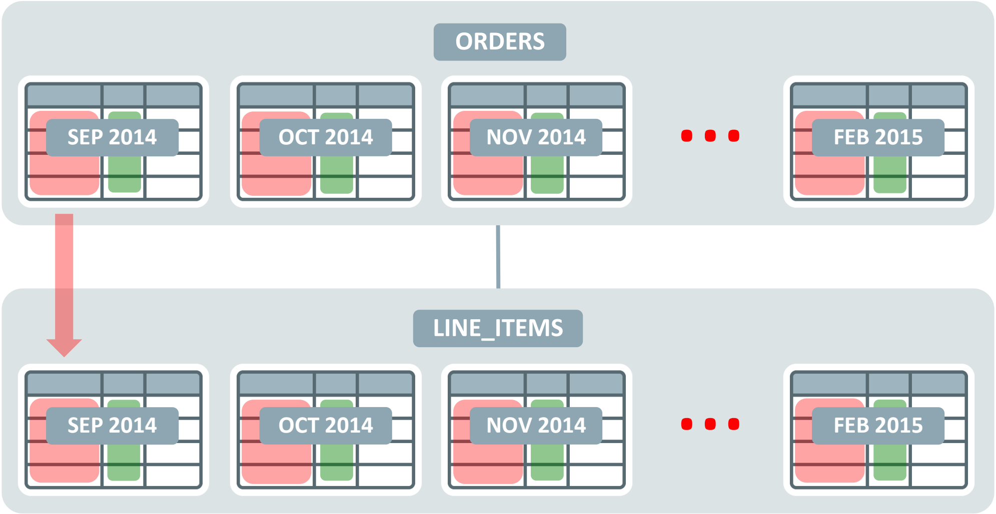
Business Problem

- Related tables benefit from same partitioning strategy
  - Sample 3NF order entry data model
- Redundant storage of same information solves problem
  - Data and maintenance overhead

Solution

- Oracle Database 11g introduces Reference Partitioning
  - Child table inherits the partitioning strategy of parent table through PK-FK
  - Intuitive modelling
- Enhanced Performance and Manageability

Without Reference Partitioning

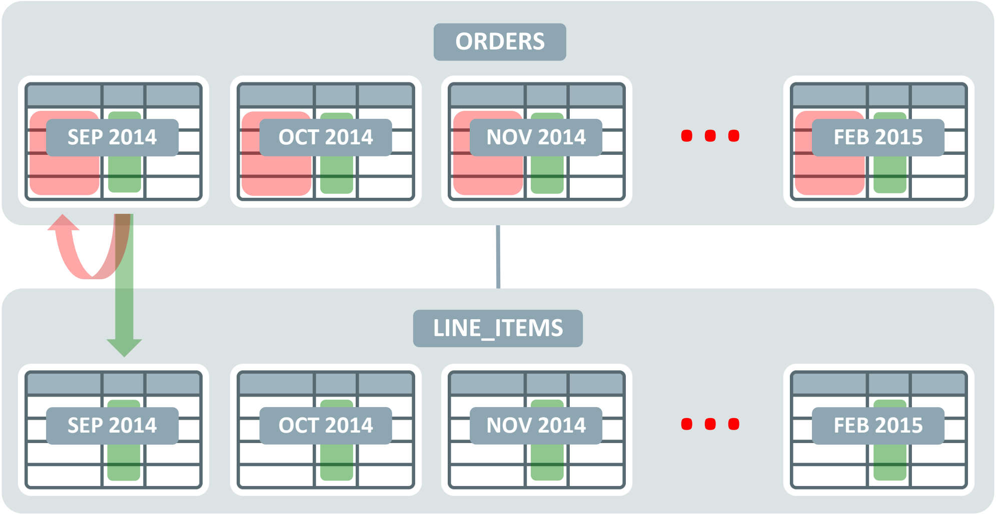



```
RANGE (order_date)
Primary key order_id
```

- Redundant storage
- Redundant maintenance

```
RANGE (order_date)
Foreign key order_id
```

With Reference Partitioning




```
RANGE (order_date)
Primary key order_id
```

- Partitioning key inherited through PK-FK relationship

```
RANGE (order_date)
Foreign key order_id
```

Reference Partitioning

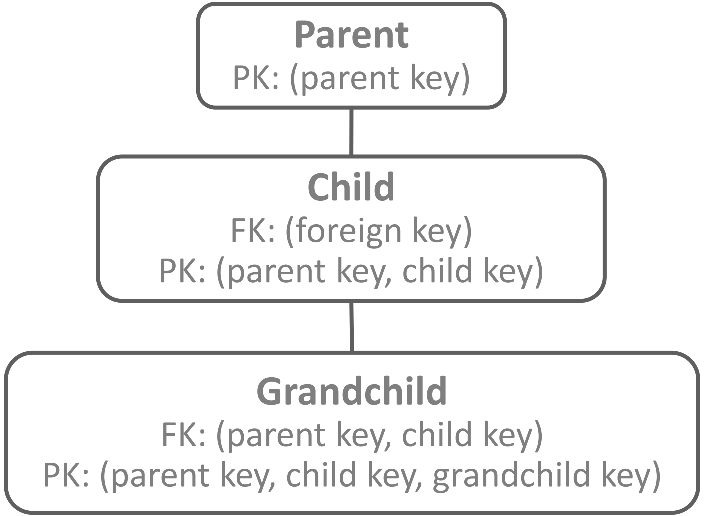
Use Cases

- Traditional relational model
  - Primary key inherits down to all levels of children and becomes part of an (elongated) primary key definition
- Object oriented-like model
  - Several levels of primary-foreign key relationship
  - Primary key on each level is primary key + “object ID”
- Reference Partitioning optimally suited to address both modeling techniques

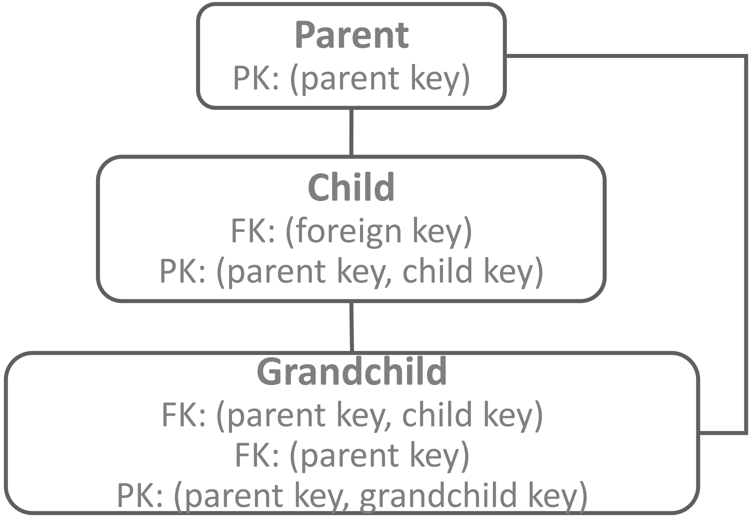
ORACLE

Reference Partitioning

Relational Model



"Object-like" model



Reference Partitioning

Example

```
create table project (project_id number not null,
                      project_number varchar2(30),
                      project_name varchar2(30), …
                      constraint proj_pk primary key (project_id))
partition by list (project_id)
(partition p1 values (1),
 partition p2 values (2),
 partition pd values (DEFAULT));
```

```
create table project_customer (project_cust_id number not null,
                               project_id number not null,
                               cust_name varchar2(30),
                               constraint pk_proj_cust primary key
                                 (project_id, project_cust_id),
                               constraint proj_cust_proj_fk foreign key
                                 (project_id) references project(project_id))
partition by reference (proj_cust_proj_fk);
```

ORACLE

Reference Partitioning

Example, cont.

```
create table proj_cust_address (project_cust_addr_id number not null,
                                project_cust_id number not null,
                                project_id number not null,
                                cust_address varchar2(30),
                                constraint pk_proj_cust_addr primary key
                                   (project_id, project_cust_addr_id),
                                constraint proj_c_addr_proj_cust_fk foreign key
                                   (project_id, project_cust_id)
                                    references project_customer
                                              (project_id, project_cust_id))
partition by reference (proj_c_addr_proj_cust_fk);
```

Reference Partitioning

Some metadata

Table information

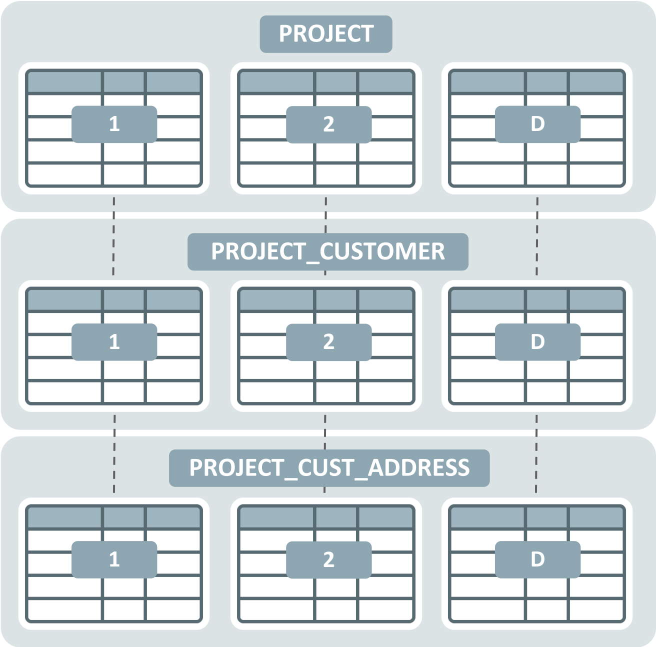
```
SQL> SELECT table_name, partitioning_type, ref_ptn_constraint_name
     FROM   user_part_tables
     WHERE  table_name IN ('PROJECT','PROJECT_CUSTOMER','PROJ_CUST_ADDRESS');

TABLE_NAME                PARTITION   REF_PTN_CONSTRAINT_NAME
------------------------ ---------   ------------------------------
PROJECT                   LIST
PROJECT_CUSTOMER          REFERENCE   PROJ_CUST_PROJ_FK
PROJ_CUST_ADDRESS         REFERENCE   PROJ_C_ADDR_PROJ_FK
```

Partition information

```
SQL> SELECT table_name, partition_name, high_value
     FROM   user_tab_partitions
     WHERE  table_name in ('PROJECT','PROJECT_CUSTOMER')
     ORDER BY table_name, partition_position;

TABLE_NAME                PARTITION_NAME       HIGH_VALUE
------------------------ ------------------- ---------------------------
PROJECT                    P1                   1
PROJECT                    P2                   2
PROJECT                    PD                   DEFAULT
PROJECT_CUSTOMER           P1
PROJECT_CUSTOMER           P2
PROJECT_CUSTOMER           PD
```

Reference Partitioning

Partition Maintenance



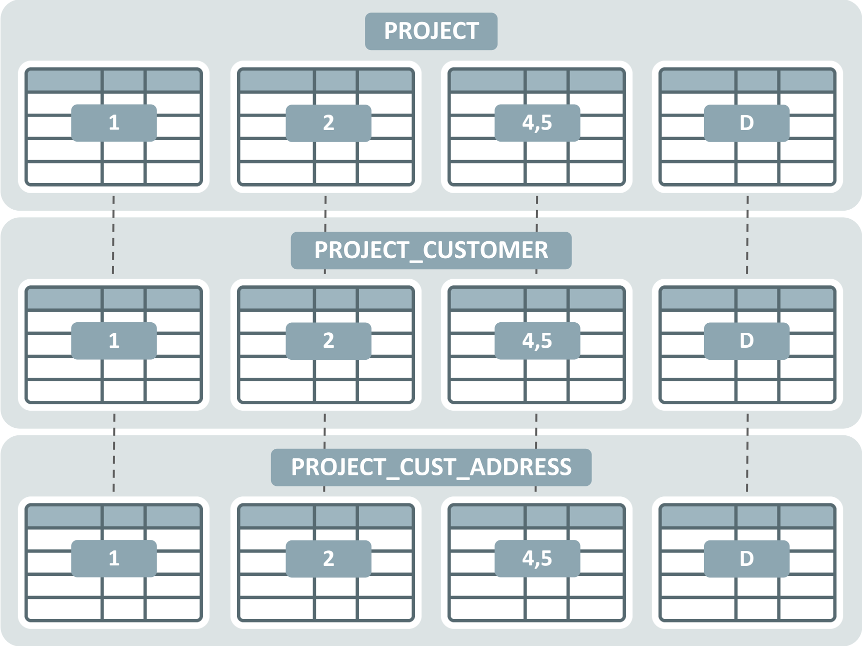

```
ALTER TABLE project
SPLIT PARTITION pd VALUES (4,5) INTO
(PARTITION pd, PARTITION p45);
```

ORACLE

Reference Partitioning

Partition Maintenance

```
ALTER TABLE project
SPLIT PARTITION pd VALUES (4,5) INTO
(PARTITION pd, PARTITION p45);
```



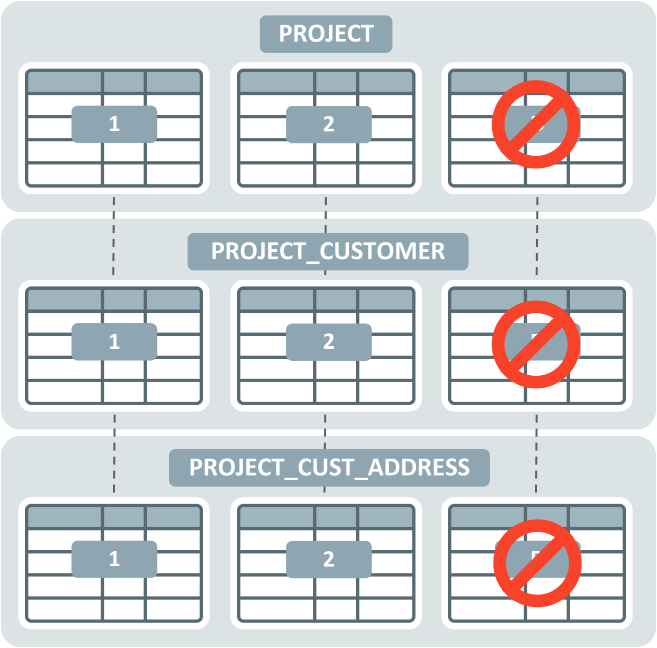
- PROJECT partition PD will be split
  - "Default" and (4,5)
- PROJECT_CUSTOMER will split its dependent partition
  - Co-location with equivalent parent record of PROJECT
  - Parent record in (4,5) means child record in (4.5)
- PROJECT_CUST_ADDRESS will split its dependent partition
  - Co-location with equivalent parent record of PROJECT_CUSTOMER
- One-level lookup required for both placements

ORACLE

Reference Partitioning

Partition Maintenance

```
ALTER TABLE project_cust_address
DROP PARTITION pd;
```



- PROJECT partition PD will be dropped
  - PK-FK is guaranteed not to be violated
- PROJECT_CUSTOMER will drop its dependent partition
- PROJECT_CUST_ADDRESS will drop its dependent partition
- Unlike “normal” partitioned tables, PK-FK relationship stays enabled
  - You cannot arbitrarily drop or truncate a partition with the PK of a PK-FK relationship
- Same is true for TRUNCATE
  - Bottom-up operation

ORACLE

Multi-Column Range Partitioning

Introduced in Oracle 8i (8.1)

Multi-column Range Partitioning

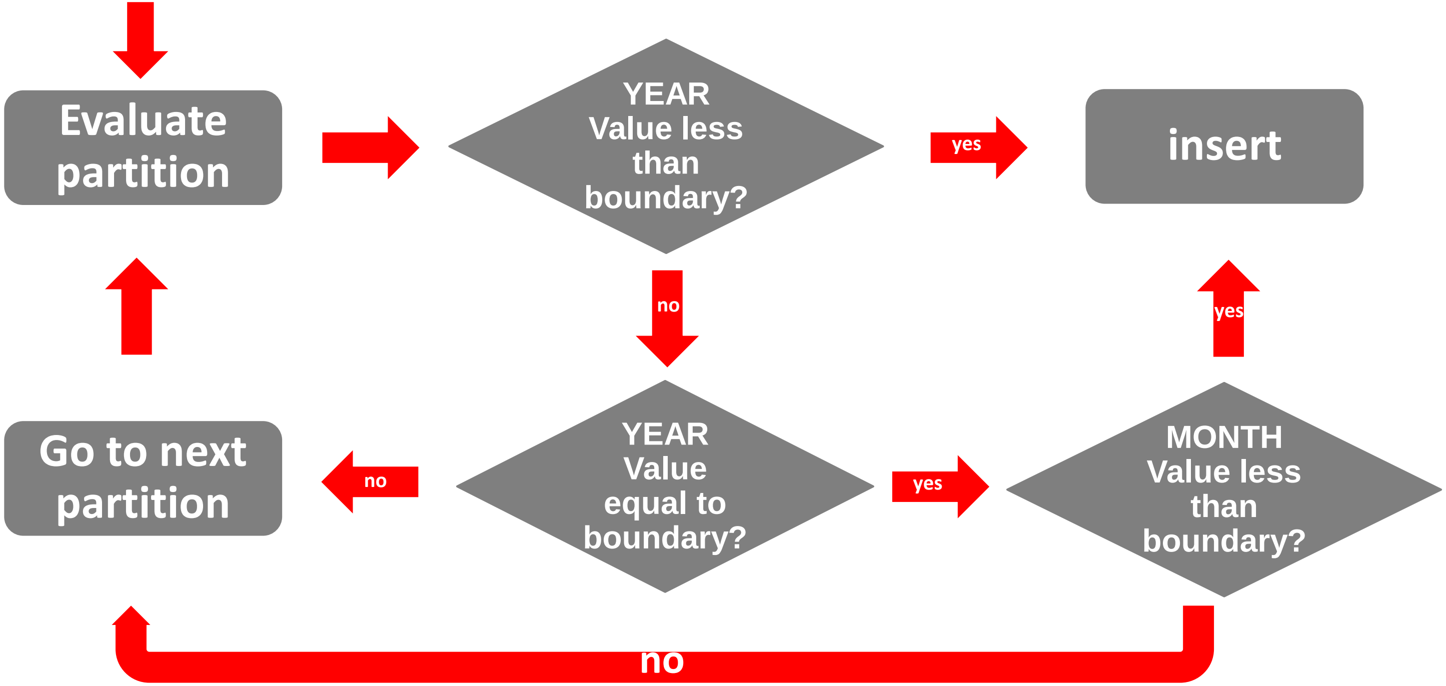
Concept

- Partitioning key is composed of several columns and subsequent columns define a higher granularity than the preceding one
  - E.g. (YEAR, MONTH, DAY)
  - It is NOT an n-dimensional partitioning
- Major watch-out is difference of how partition boundaries are evaluated
  - For simple RANGE, the boundaries are **less than** (exclusive)
  - Multi-column RANGE boundaries are **less than or equal**
    - The n^{th} column is investigated only when all previous (n-1) values of the multicolumn key exactly match the (n-1) bounds of a partition

ORACLE

Multi-Column Range Partition

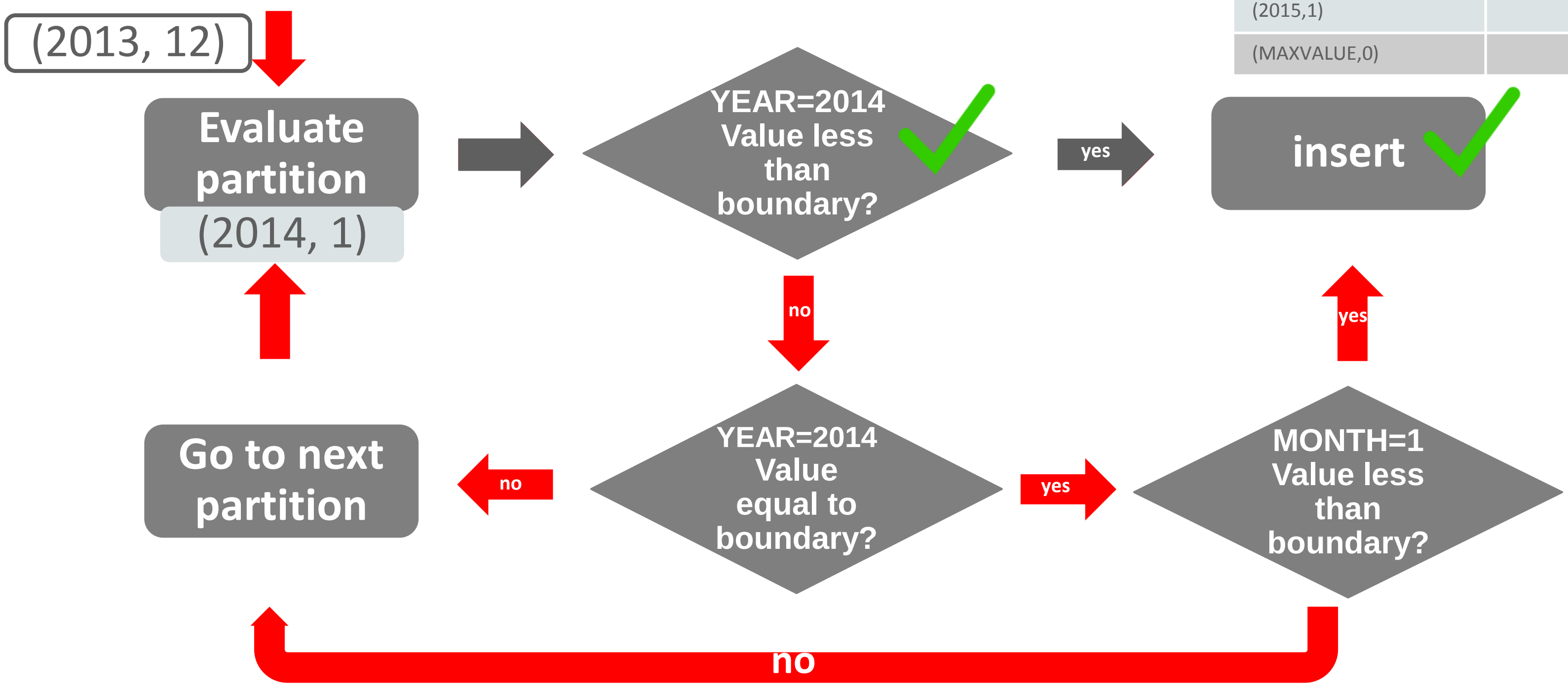
Sample Decision Tree (YEAR, MONTH)



Multi-Column Range Partition

Example

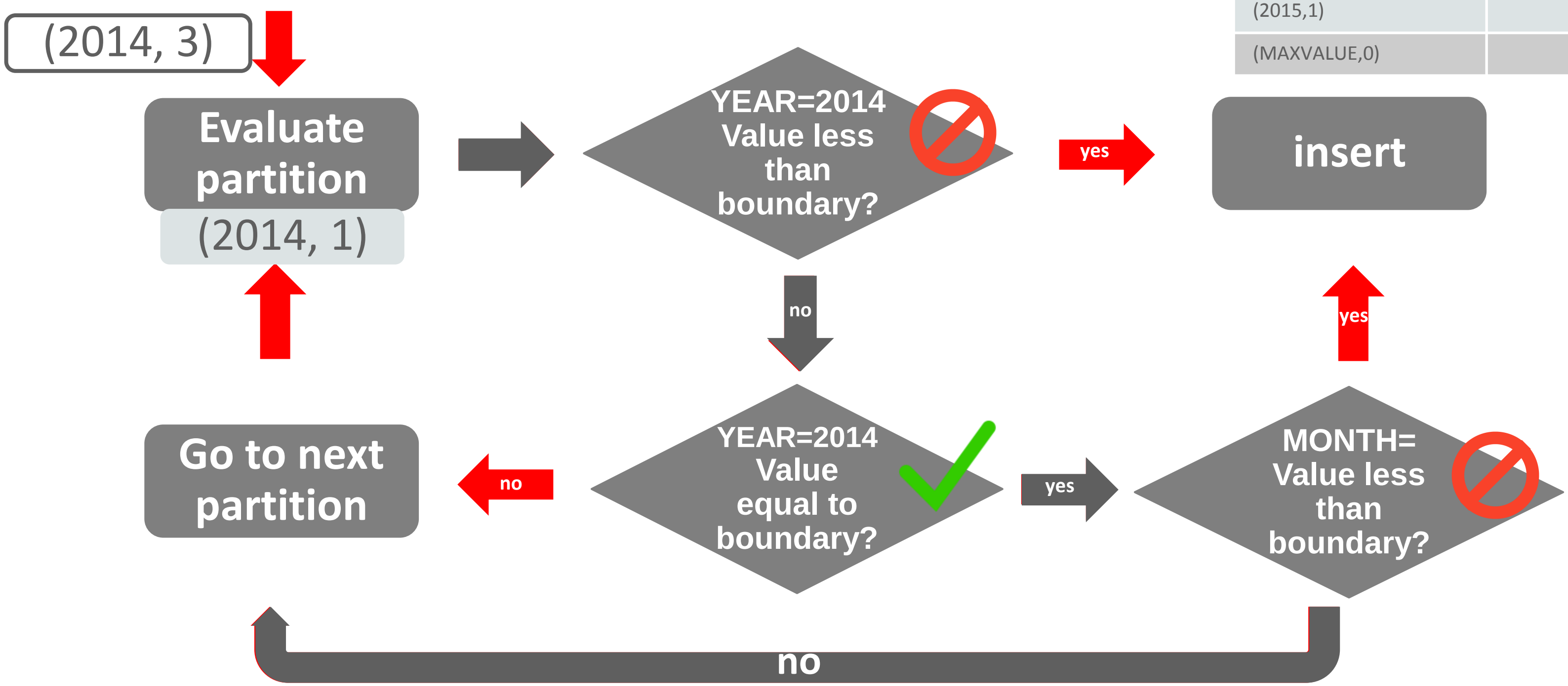
(YEAR,MONTH) Boundaries	Values
(2014,1)	(2013, 12)
(2014,4)	
(2014,7)	
(2014,10)	
(2015,1)	
(MAXVALUE,0)	



Multi-Column Range Partition

Example Cont'd

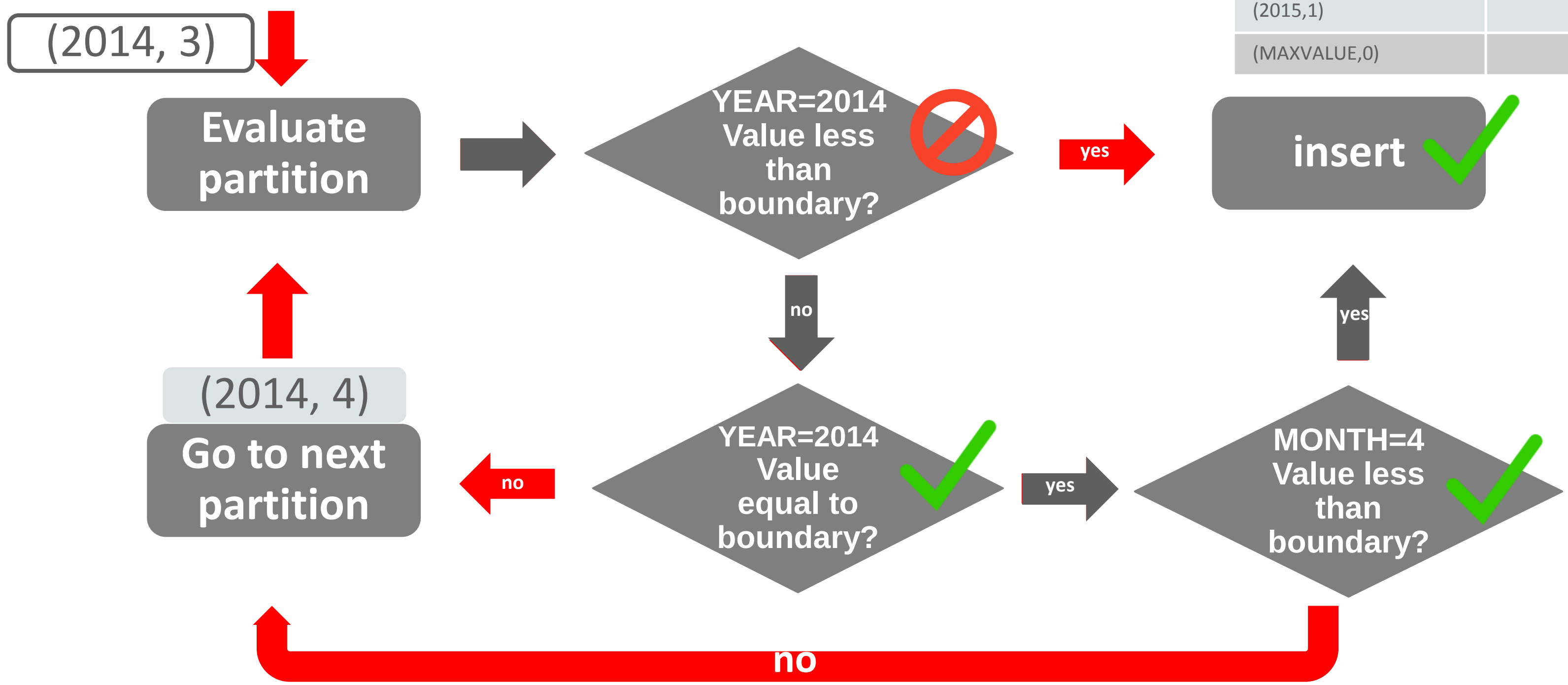
(YEAR,MONTH) Boundaries	Values
(2014,1)	(2013, 12)
(2014,4)	
(2014,7)	
(2014,10)	
(2015,1)	
(MAXVALUE,0)	



Multi-Column Range Partition

Example Cont'd

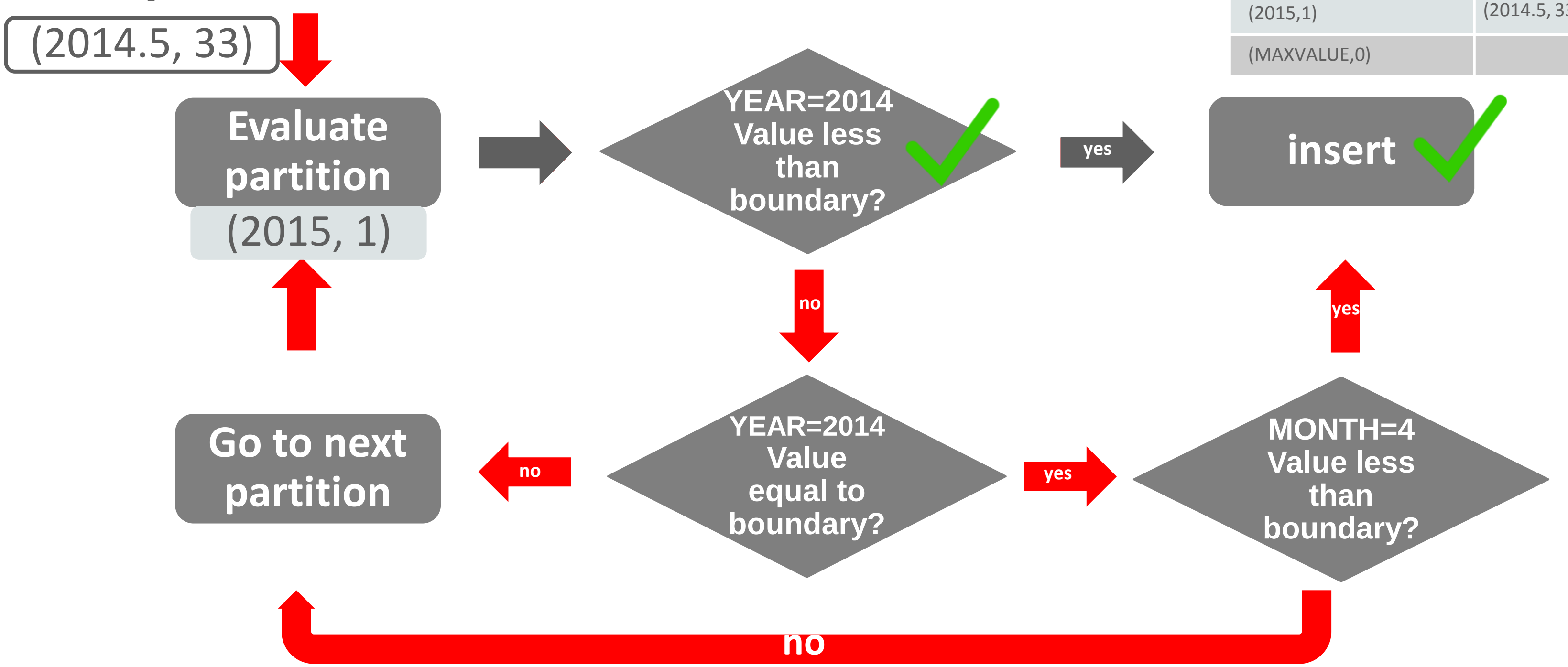
(YEAR,MONTH) Boundaries	Values
(2014,1)	(2013, 12)
(2014,4)	(2014, 3)
(2014,7)	
(2014,10)	
(2015,1)	
(MAXVALUE,0)	



Multi-Column Range Partition

Example Cont'd

(YEAR,MONTH) Boundaries	Values
(2014,1)	(2013, 12)
(2014,4)	(2014, 3)
(2014,7)	
(2014,10)	
(2015,1)	(2014.5, 33)
(MAXVALUE,0)	



Multi-Column Range Partitioning

Some things to bear in mind

- Powerful partitioning mechanism to add a third (or more) dimensions
  - Smaller data partitions
- Pruning works also for trailing column predicates without filtering the leading column(s)
- Boundaries are not enforced by the partition definition
  - Ranges are consecutive
- Logical ADD partition can mean SPLIT partition in the middle of the table

ORACLE

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

```
CREATE TABLE product_sales (prod_id number, site_id CHAR(2),start_date date)
PARTITION BY RANGE (site_id, start_date)
SUBPARTITION BY HASH (prod_id) SUBPARTITIONS 16
(PARTITION de_2013 VALUES LESS THAN ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION de_2014 VALUES LESS THAN ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION us_2013 VALUES LESS THAN ('US',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION us_2014 VALUES LESS THAN ('US',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION za_2013 VALUES LESS THAN ('ZA',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION za_2014 VALUES LESS THAN ('ZA',to_date('01-JAN-2015','dd-mon-yyyy'))
);
```

Character SITE_ID has to be defined in an ordered fashion

ORACLE

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

AC, CN

EE, ES, UK

VE, VN

```
CREATE TABLE product_sales (prod_id number, site_id CHAR(2),start_date date)
PARTITION BY RANGE (site_id, start_date)
SUBPARTITION BY HASH (prod_id) SUBPARTITIONS 16
(PARTITION de_2013 VALUES LESS THAN ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION de_2014 VALUES LESS THAN ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION us_2013 VALUES LESS THAN ('US',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION us_2014 VALUES LESS THAN ('US',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION za_2013 VALUES LESS THAN ('ZA',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION za_2014 VALUES LESS THAN ('ZA',to_date('01-JAN-2015','dd-mon-yyyy'))
);
```

Non-defined SITE_ID will follow the LESS THAN probe and always end in the lowest partition of a defined SITE_ID

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

(DE, 2015)

(US, 2015)

(ZA, 2015)

```
CREATE TABLE product_sales (prod_id number, site_id CHAR(2),start_date date)
PARTITION BY RANGE (site_id, start_date)
SUBPARTITION BY HASH (prod_id) SUBPARTITIONS 16
(PARTITION de_2013 VALUES LESS THAN ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION de_2014 VALUES LESS THAN ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION us_2013 VALUES LESS THAN ('US',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION us_2014 VALUES LESS THAN ('US',to_date('01-JAN-2015','dd-mon-yyyy')),
 PARTITION za_2013 VALUES LESS THAN ('ZA',to_date('01-JAN-2014','dd-mon-yyyy')),
 PARTITION za_2014 VALUES LESS THAN ('ZA',to_date('01-JAN-2015','dd-mon-yyyy'))
);
```

?

Future dates will always go in the lowest partition of the next higher SITE_ID or being rejected

ORACLE

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

AC, CN

EE, ES, UK

```
create table product_sales (prod_id number, site_id CHAR(2),start_date date)
partition by range (site_id, start_date)
subpartition by hash (prod_id) subpartitions 16
(partition below_de  values less than ('DE',to_date('01-JAN-1492','dd-mon-yyyy')),
partition de_2013 values less than ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
partition de_2014 values less than ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
partition de_max  values less than ('DE',MAXVALUE),
partition below_us  values less than ('US',to_date('01-JAN-1492','dd-mon-yyyy')),
 …
partition za_max  values less than ('ZA',MAXVALUE),
partition pmax   values less than (MAXVALUE,MAXVALUE));
```

Introduce a dummy 'BELOW_X' partition to catch "lower" nondefined SITE_ID

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

(DE, 2015)

(ZA, 2015)

```
create table product_sales (prod_id number, site_id CHAR(2),start_date date)
partition by range (site_id, start_date)
subpartition by hash (prod_id) subpartitions 16
(partition below_de  values less than ('DE',to_date('01-JAN-1492','dd-mon-yyyy')),
partition de_2013 values less than ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
partition de_2014 values less than ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
partition de_max  values less than ('DE',MAXVALUE),
partition below_us  values less than ('US',to_date('01-JAN-1492','dd-mon-yyyy')),
 …
partition za_max  values less than ('ZA',MAXVALUE),
partition pmax   values less than (MAXVALUE,MAXVALUE));
```

Introduce a MAXVALUE 'X_FUTURE' partition to catch future dates

Multi-Column Range Partition

A slightly different real-world scenario

- Multi-column range used to introduce a third (non-numerical) dimension

```
create table product_sales (prod_id number, site_id CHAR(2),start_date date)
partition by range (site_id, start_date)
subpartition by hash (prod_id) subpartitions 16
(partition below_de  values less than ('DE',to_date('01-JAN-1492','dd-mon-yyyy')),
partition de_2013 values less than ('DE',to_date('01-JAN-2014','dd-mon-yyyy')),
partition de_2014 values less than ('DE',to_date('01-JAN-2015','dd-mon-yyyy')),
partition de_max  values less than ('DE',MAXVALUE),
partition below_us  values less than ('US',to_date('01-JAN-1492','dd-mon-yyyy')),
 …
partition za_max  values less than ('ZA',MAXVALUE),
partition pmax   values less than (MAXVALUE,MAXVALUE));
```

If necessary, catch the open-ended SITE_ID (leading key column)

ORACLE

Virtual Column Based Partitioning

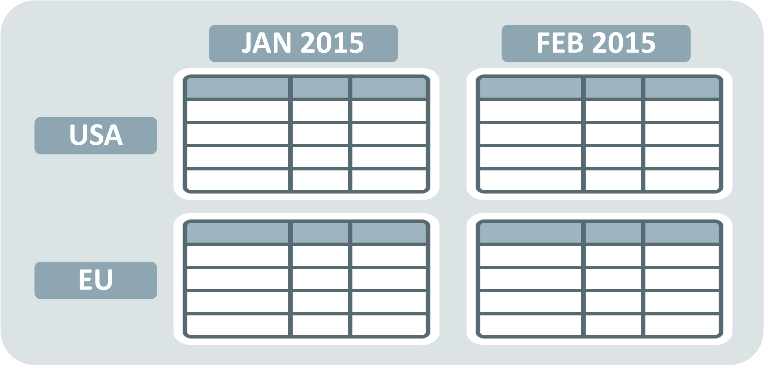
Introduced in Oracle 11g Release 1 (11.1)

ORACLE

Virtual Column Based Partitioning

```
ORDERS

ORDER_ID    ORDER_DATE  CUSTOMER_ID... REGION AS (SUBSTR(ORDER_ID,6,2))
---------- ----------- ----------- -- ---------------------------------
9834-US-14 12-JAN-2015       65920          US
8300-EU-97 14-FEB-2015       39654          EU
3886-EU-02 16-JAN-2015        4529          EU
2566-US-94 19-JAN-2015       15327          US
3699-US-63 02-FEB-2015       18733          US
```

- REGION requires no storage
- Partition by ORDER_DATE, REGION



ORACLE

Virtual Columns

Example

- Base table with all attributes ...

```
CREATE TABLE accounts
(acc_no      number(10)    not null,
 acc_name    varchar2(50) not null, …
```

12500	Adams	
12507	Blake	
12666	King	
12875	Smith	

ORACLE

Virtual Columns

Example

- Base table with all attributes ...

  ... is extended with the virtual (derived) column

```
CREATE TABLE accounts
(acc_no     number(10)   not null,
 acc_name   varchar2(50) not null, ...
 acc_branch number(2)    generated always as
   (to_number(substr(to_char(acc_no),1,2)))
```

12500	Adams	12
12507	Blake	12
12666	King	12
12875	Smith	12

ORACLE

Virtual Columns

Example

- Base table with all attributes ...

... is extended with the virtual (derived) column

... and the virtual column is used as partitioning key

```
CREATE TABLE accounts
(acc_no      number(10)    not null,
 acc_name    varchar2(50) not null, ...
 acc_branch number(2)     generated always as
   (to_number(substr(to_char(acc_no),1,2)))
partition by list (acc_branch) …
```

12500	Adams	12
12507	Blake	12
12666	King	12
12875	Smith	12

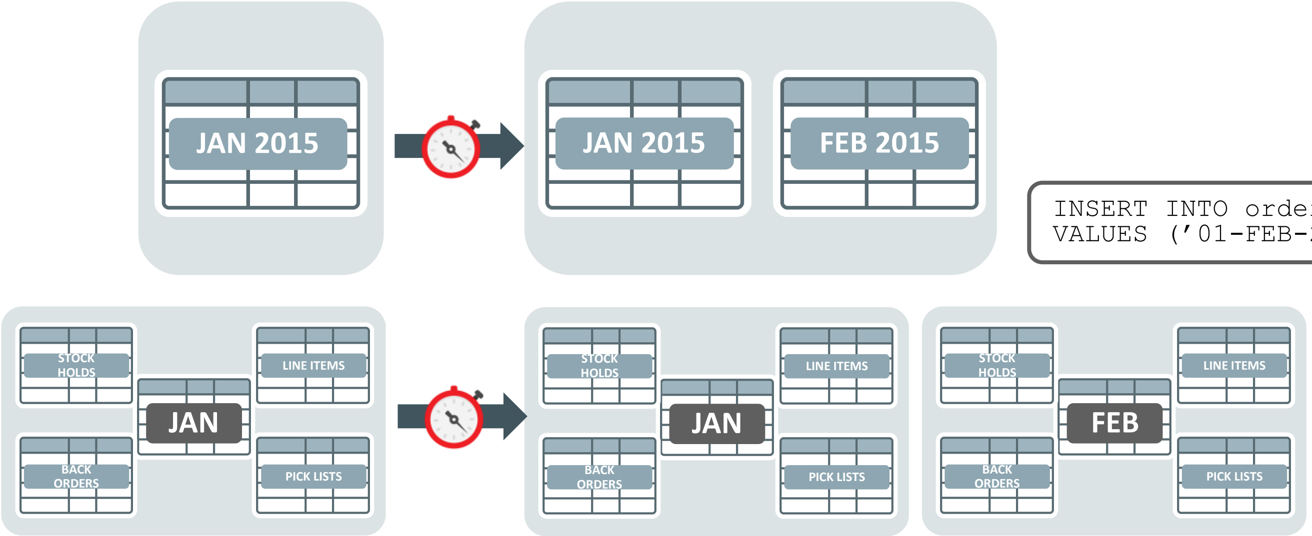
...

32320	Jones	32
32407	Clark	32
32758	Hurd	32
32980	Kelly	32

Interval Reference Partitioning

Introduced in Oracle 12c

Interval-Reference Partitioning




```
INSERT INTO orders
VALUES ('01-FEB-2015', ... );
```

- New partitions are automatically created when new data arrives
- All child tables will be automatically maintained
- Combination of two successful partitioning strategies for better business modeling

ORACLE

Interval-Reference Partitioning

```
SQL> REM create some interval-referenced tables ..
SQL> create table intRef_p (pkcol number not null, col2 varchar2(200),
  2                         constraint pk_intref primary key (pkcol))
  3  partition by range (pkcol) interval (10)
  4  (partition p1 values less than (10));

Table created.

SQL>
SQL> create table intRef_c1 (pkcol number not null, col2 varchar2(200), fkcol number not null,
  2                          constraint pk_c1 primary key (pkcol),
  3                          constraint fk_c1 foreign key (fkcol) references intRef_p(pkcol) ON DELETE CASCADE)
  4  partition by reference (fk_c1);

Table created.

SQL>
SQL> create table intRef_c2 (pkcol number primary key not null, col2 varchar2(200), fkcol number not null,
  2                          constraint fk_c2 foreign key (fkcol) references intRef_p(pkcol) ON DELETE CASCADE)
  3  partition by reference (fk_c2);

Table created.
```

Interval-Reference Partitioning

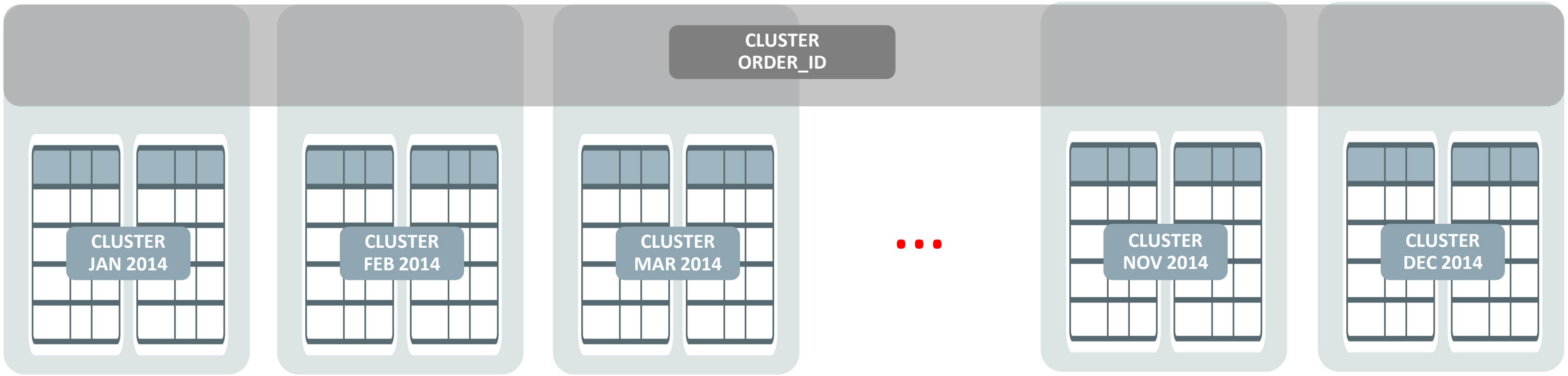
- New partitions only created when data arrives
  - No automatic partition instantiation for complete reference tree
  - Optimized for sparsely populated reference partitioned tables
- Partition names inherited from already existent partitions
  - Name inheritance from direct relative
  - Parent partition p100 will result in child partition p100
  - Parent partition p100 and child partition c100 will result in grandchild partition c100

ORACLE

Range-Partitioned Hash Cluster

Introduced in Oracle 12c (Release 12.102)

Range-Partitioned Hash Cluster

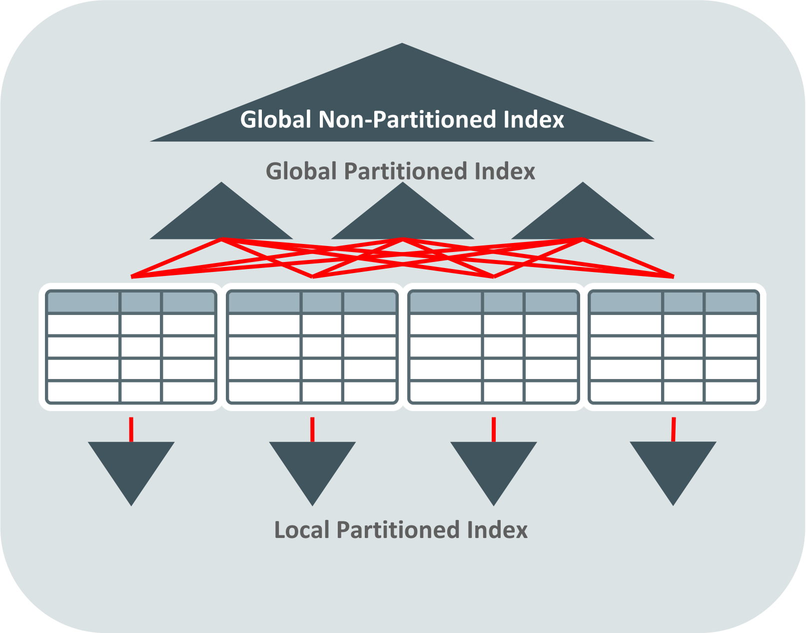


- Single-level range partitioning
  - No composite partitioning
  - No index clusters

Indexing of Partitioned Tables

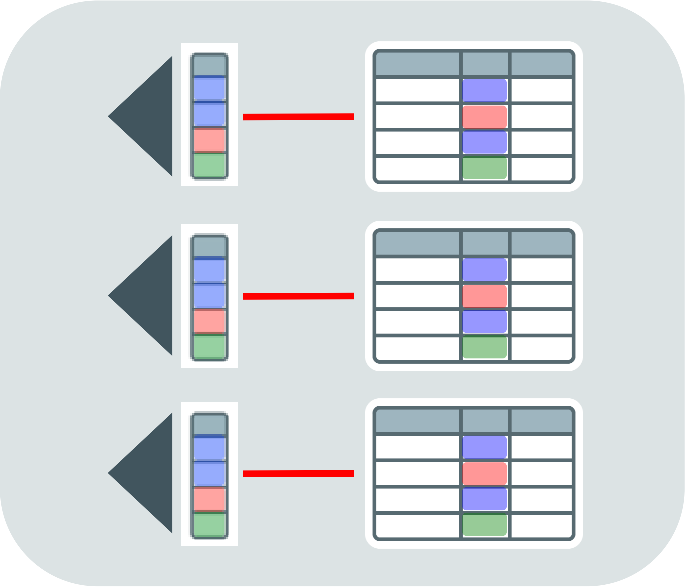
Index Partitioning

- GLOBAL index points to rows in any partition
  - Index can be partitioned or not
  - Partition maintenance affects entire index
- LOCAL index is partitioned same as table
  - Index partitioning key can be different from index key
  - Index partitions can be maintained separately



Local Index

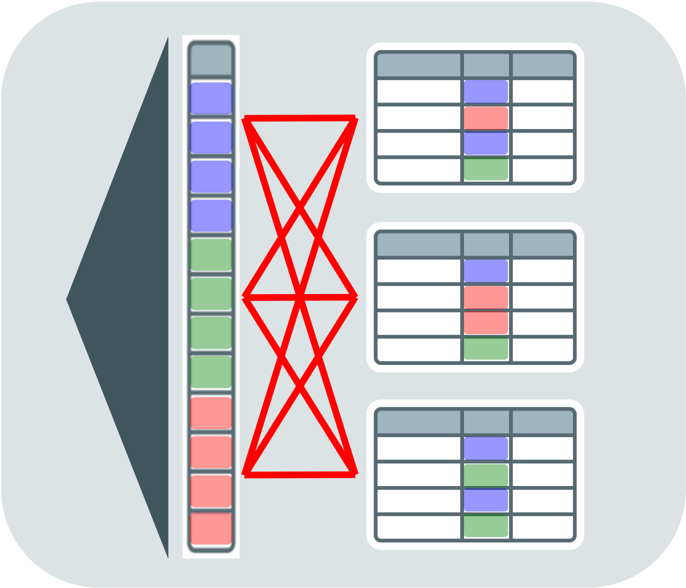
- Index is partitioned along same boundaries as data
  - B-tree or bitmap
- Pros
  - Easy to manage
  - Parallel index scans
- Cons
  - Less efficient for retrieving small amounts of data



ORACLE

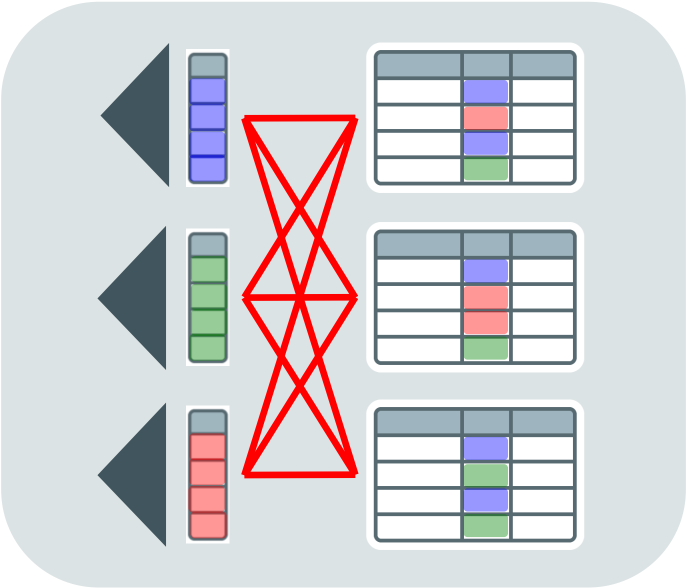
Global Non-Partitioned Index

- One index b-tree structure that spans all partitions
- Pros
  - Efficient access to any individual record
- Cons
  - Partition maintenance always involves index maintenance



Global Partitioned Index

- Index is partitioned independently of data
  - Each index structure may reference any and all partitions.
- Pros
  - Availability and manageability
- Cons
  - Partition maintenance always involves index maintenance



ORACLE

Partial Indexing

Introduced in Oracle 12c

Enhanced Indexing with Oracle Partitioning

Indexing prior to Oracle Database 12c

- Local indexes
- Non-partitioned or partitioned global indexes
- Usable or unusable index segments
  - Non-persistent status of index, no relation to table

ORACLE

Enhanced Indexing with Oracle Partitioning

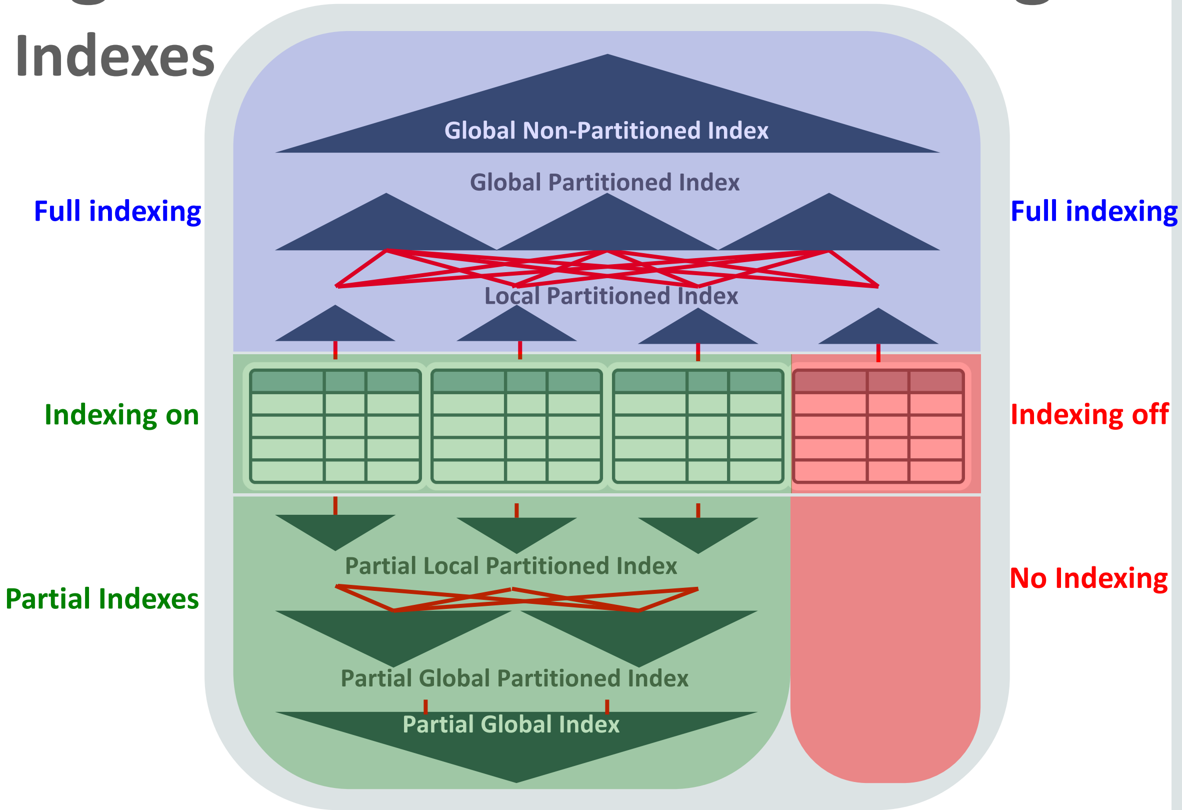
Indexing with Oracle Database 12c

- Local indexes
- Non-partitioned or partitioned global indexes
- Usable or unusable index segments
  - Non-persistent status of index, no relation to table
- Partial local and global indexes
  - Partial indexing introduces table and [sub]partition level metadata
  - Leverages usable/unusable state for local partitioned indexes
  - Policy for partial indexing can be overwritten

Enhanced Indexing with Oracle Partitioning

Partial Local and Global Indexes

- Partial indexes span only some partitions
- Applicable to local and global indexes
- Complementary to full indexing
- Enhanced business modeling



Enhanced Indexing with Oracle Partitioning

Partial Local and Global Indexes

Before

```
SQL> create table pt (col1, col2, col3, col4)
  2  indexing off
  3  partition by range (col1)
  4  interval (1000)
  5  (partition p100 values less than (101) indexing on,
  6   partition p200 values less than (201) indexing on,
  7   partition p300 values less than (301) indexing on);

Table created.

SQL> REM partitions and its indexing status
SQL> select partition_name, high_value, indexing
  2  from user_tab_partitions where table_name='PT';

PARTITION_NAME                 HIGH_VALUE                     INDEXING
------------------------------ ------------------------------ --------
P100                           101                            ON
P200                           201                            ON
P300                           301                            ON
SYS_P1256                      1301                           OFF
```

After

```
SQL> REM local indexes
SQL> create index i_l_partpt on pt(col1) local indexing partial;
SQL> create index i_l_pt on pt(col4) local;

SQL> REM global indexes
SQL> create index i_g_partpt on pt(col2) indexing partial;
SQL> create index i_g_pt on pt(col3);


SQL> REM index status
SQL> select index_name, partition_name, status, null
  2  from user_ind_partitions where index_name in ('I_L_PARTPT','I_L_PT')
  3  union all
  4  select index_name, indexing, status, orphaned_entries
  5  from user_indexes where index_name in ('I_G_PARTPT','I_G_PT');

INDEX_NAME                     PARTITION_NAME                 STATUS     ORPHAN
------------------------------ ------------------------------ ---------- ------
I_L_PARTPT                     P100                           USABLE
I_L_PARTPT                     P200                           USABLE
I_L_PARTPT                     P300                           USABLE
I_L_PARTPT                     SYS_P1257                      UNUSABLE
I_L_PT                         P200                           USABLE
I_L_PT                         P300                           USABLE
I_L_PT                         SYS_P1258                      USABLE
I_L_PT                         P100                           USABLE
I_G_PT                         FULL                           VALID      NO
I_G_PARTPT                     PARTIAL                        VALID      NO

10 rows selected.
```

ORACLE

Enhanced Indexing with Oracle Partitioning

Partial Local and Global Indexes

- Partial global index excluding partition 4

```
SQL> explain plan for select count(*) from pt where col2 = 3;

Explained.

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

-----------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                   | Name       | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                            |            |     1 |    22 |    54  (12)| 00:00:01 |       |       |
|   1 |  SORT AGGREGATE                             |            |     1 |    22 |            |          |       |       |
|   2 |   VIEW                                      | VW_TE_2    |     2 |       |    54  (12)| 00:00:01 |       |       |
|   3 |    UNION-ALL                                |            |       |       |            |          |       |       |
|*  4 |     TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| PT       |     1 |    26 |     2   (0)| 00:00:01 | ROWID | ROWID |
|*  5 |      INDEX RANGE SCAN                       | I_G_PARTPT |     1 |       |     1   (0)| 00:00:01 |       |       |
|   6 |     PARTITION RANGE SINGLE                  |            |     1 |    26 |    52  (12)| 00:00:01 |     4 |     4 |
|*  7 |      TABLE ACCESS FULL                      | PT         |     1 |    26 |    52  (12)| 00:00:01 |     4 |     4 |
-----------------------------------------------------------------------------------------------------------------------------

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

   4 - filter("PT"."COL1"<301)
   5 - access("COL2"=3)
   7 - filter("COL2"=3)
```

Partitioning for Performance

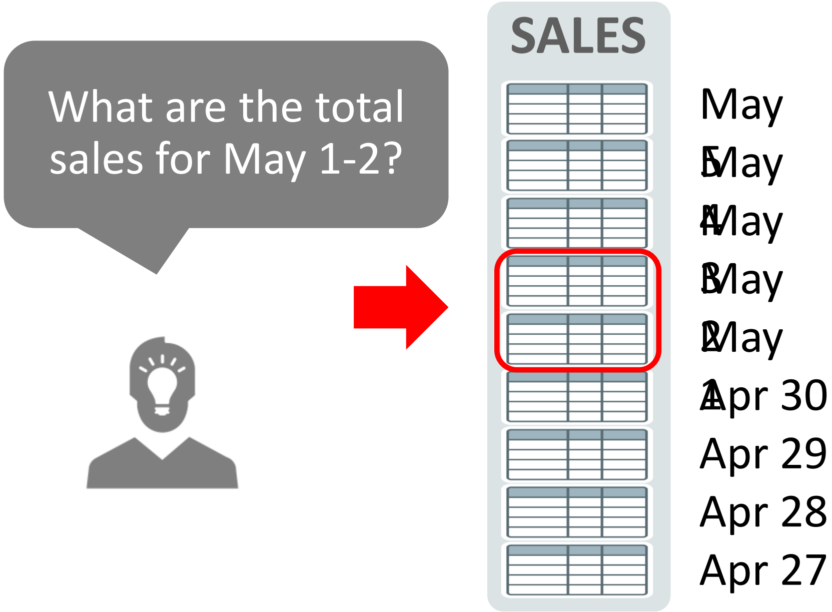
Partitioning for Performance

- Partitioning is transparently leveraged to improve performance
- Partition pruning
  – Using partitioning metadata to access only partitions of interest
- Partition-wise joins
  – Join equi-partitioned tables with minimal resource consumption
  – Process co-location capabilities for RAC environments
- Partition-Exchange loading
  – “Load” new data through metadata operation

ORACLE

Partitioning for Performance

Partition Pruning



- Partition elimination
  - Dramatically reduces amount of data retrieved from storage
  - Performs operations only on relevant partitions
  - Transparently improves query performance and optimizes resource utilization

Partition Pruning

- Works for simple and complex SQL statements
- Transparent to any application
- Two flavors of pruning
  - Static pruning at compile time
  - Dynamic pruning at runtime
- Complementary to Exadata Storage Server
  - Partitioning prunes logically through partition elimination
  - Exadata prunes physically through storage indexes
    - Further data reduction through filtering and projection

Exadata Database Machine

Optimized for large scans

10 TB of user data
Requires 10 TB of IO

1 TB
with compression

100 GB
with partition pruning

20 GB
with Storage Indexes

5 GB
with Smart Scans

Subseconds
On Database
Machine

2000x less data needs to be processed

ORACLE

Static Partition Pruning

```
SELECT sum(amount_sold) FROM sales
WHERE times_id
BETWEEN '01-MAR-2014' and '31-MAY-2014';
```

14-JAN	14-FEB	14-MAR	14-APR	14-MAY	14-JUN

- Relevant Partitions are known at compile time
  - Look for actual values in PSTART/PSTOP columns in the plan
- Optimizer has most accurate information for the SQL statement

Static Pruning

Sample Plan

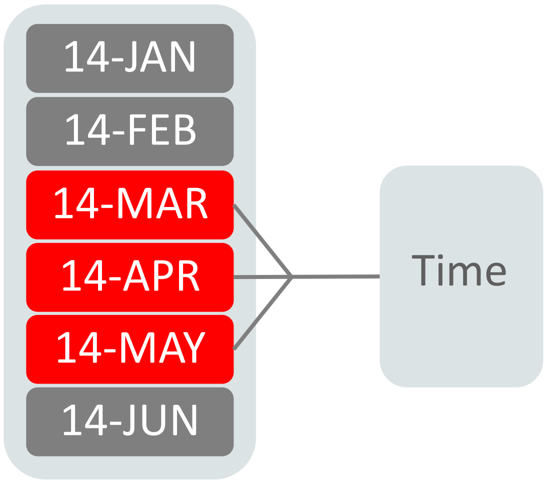
```
SELECT sum(amount_sold)
FROM sh.sales s, sh.times t
WHERE s.time_id = t.time_id
AND s.time_id between TO_DATE('01-JAN-2014', 'DD-MON-YYYY')
   and TO_DATE('01-JAN-2015', 'DD-MON-YYYY')

Plan hash value: 2025449199

-----------------------------------------------------------------------------------------------------
| Id | Operation                  | Name   | Rows   | Bytes  | Cost (%CPU)| Time      | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT           |        |        |        |    3  (100)|           |       |       |
|  1 |  SORT AGGREGATE            |        |     1  |    12  |            |           |       |       |
|  2 |   PARTITION RANGE ITERATOR |        |   313  |  3756  |    3    (0)| 00:00:01  |     9 |    13 |
|* 3 |    TABLE ACCESS FULL       | SALES  |   313  |  3756  |    3    (0)| 00:00:01  |     9 |    13 |
-----------------------------------------------------------------------------------------------------

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

   3 - filter("S"."TIME_ID"<=TO_DATE(' 2015-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

22 rows selected.
```

Static Pruning

Sample Plan

```
SELECT sum(amount_sold)
FROM sh.sales s, sh.times t
WHERE s.time_id = t.time_id
AND s.time_id between TO_DATE('01-JAN-2014', 'DD-MON-YYYY')
   and TO_DATE('01-JAN-2015', 'DD-MON-YYYY')

Plan hash value: 2025449199

----------------------------------------------------------------------------------------------------
| Id | Operation                  | Name    | Rows   | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT           |         |        |       |     3  (100)|           |       |       |
|  1 |  SORT AGGREGATE            |         |      1 |    12 |             |           |       |       |
|  2 |   PARTITION RANGE ITERATOR |         |    313 |  3756 |     3    (0)| 00:00:01  |     9 |    13 |
|* 3 |    TABLE ACCESS FULL       | SALES   |    313 |  3756 |     3    (0)| 00:00:01  |     9 |    13 |
----------------------------------------------------------------------------------------------------

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

  3 - filter("S"."TIME_ID"<=TO_DATE(' 2015-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

22 rows selected.
```

Dynamic Partition Pruning




```
SELECT sum(amount_sold)
FROM sales s, times t
WHERE t.time_id = s.time_id
AND   t.calendar_month_desc IN
        (‘MAR-2014’, ‘APR-2014’, ‘MAY-2014’);
```

- Advanced Pruning mechanism for complex queries
- Relevant partitions determined at runtime
  - Look for the word ‘KEY’ in PSTART/PSTOP columns in the Plan

Dynamic Partition Pruning

Sample Plan – Nested Loop

```
SELECT sum(amount_sold)
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.calendar_month_desc in ('MAR-2014', 'APR-2014', 'MAY-2014')

Plan hash value: 1350851517

-------------------------------------------------------------------------------------------------
| Id | Operation                   | Name   | Rows   | Bytes | Cost  (%CPU)| Time       | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT            |        |        |       |   13   (100)|            |       |       |
|  1 |  SORT AGGREGATE             |        |      1 |    28 |             |            |       |       |
|  2 |   NESTED LOOP               |        |      2 |    56 |   13     (0)| 00:00:01   |       |       |
|* 3 |    TABLE ACCESS FULL        | TIMES  |      2 |    32 |   13     (8)| 00:00:01   |       |       |
|  4 |    PARTITION RANGE ITERATOR |        |      2 |    24 |    0     (0)|            |   KEY |   KEY |
|* 5 |     TABLE ACCESS FULL       | SALES  |      2 |    24 |    0     (0)|            |   KEY |   KEY |
-------------------------------------------------------------------------------------------------

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

   3 - filter(("T"."CALENDAR_MONTH_DESC"='MAR-2014' OR "T"."CALENDAR_MONTH_DESC"='APR-2014'
              OR "T"."CALENDAR_MONTH_DESC"='MAY-2014'))
   5 - filter("T"."TIME_ID"="S"."TIME_ID")

26 rows selected.
```

ORACLE

Dynamic Partition Pruning

Sample Plan – Nested Loop

```
SELECT sum(amount_sold)
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.calendar_month_desc in ('MAR-2014', 'APR-2014', 'MAY-2014')

Plan hash value: 1350851517

---------------------------------------------------------------------------------------------------
| Id | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
---------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT           |       |       |       |   13  (100)|           |       |       |
|  1 |  SORT AGGREGATE            |       |     1 |    28 |            |           |       |       |
|  2 |   NESTED LOOP              |       |     2 |    56 |   13    (0)| 00:00:01  |       |       |
|* 3 |    TABLE ACCESS FULL       | TIMES |     2 |    32 |   13    (8)| 00:00:01  |       |       |
|  4 |    PARTITION RANGE ITERATOR|       |     2 |    24 |    0    (0)|           |   KEY |   KEY |
|* 5 |     TABLE ACCESS FULL      | SALES |     2 |    24 |    0    (0)|           |   KEY |   KEY |
---------------------------------------------------------------------------------------------------

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

   3 - filter(("T"."CALENDAR_MONTH_DESC"='MAR-2014' OR "T"."CALENDAR_MONTH_DESC"='APR-2014'
              OR "T"."CALENDAR_MONTH_DESC"='MAY-2014'))
   5 - filter("T"."TIME_ID"="S"."TIME_ID")

26 rows selected.
```

ORACLE

Dynamic Partition Pruning

Sample Plan - Subquery pruning

```
SELECT /*+ FULL(s) USE_HASH(s, t) CARDINALITY(s, 10000000000) */ sum(amount_sold)
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.calendar_month_desc in ('MAR-2014', 'APR-2014', 'MAY-2014')

Plan hash value: 2475767165

----------------------------------------------------------------------------------------------------
| Id | Operation                  | Name   | Rows   | Bytes  | Cost  (%CPU)| Time       | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT           |        |        |        |  2000K(100)|            |       |       |
|  1 |  SORT AGGREGATE            |        |      1 |     28 |            |            |       |       |
|* 2 |   HASH JOIN                |        |    24M|    646M|  2000K(100)| 06:40:01   |       |       |
|* 3 |    TABLE ACCESS FULL       | TIMES  |      2 |     32 |    43   (8)| 00:00:01   |       |       |
|  4 |    PARTITION RANGE SUBQUERY|        |    10G|    111G|  1166K(100)| 03:53:21   |KEY(SQ)|KEY(SQ)|
|  5 |     TABLE ACCESS FULL      | SALES  |    10G|    111G|  1166K(100)| 03:53:21   |KEY(SQ)|KEY(SQ)|
----------------------------------------------------------------------------------------------------

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

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   3 - filter(("T"."CALENDAR_MONTH_DESC"='MAR-2014' OR "T"."CALENDAR_MONTH_DESC"='APR-2014'
              OR "T"."CALENDAR_MONTH_DESC"='MAY-2014'))

26 rows selected.
```

ORACLE

Dynamic Partition Pruning

Sample Plan - Bloom filter pruning

```
SELECT sum(amount_sold)
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.calendar_month_desc in ('MAR-2014', 'APR-2014', 'MAY-2014')

Plan hash value: 365741303

---------------------------------------------------------------------------------------------------------
| Id | Operation                    | Name    | Rows  | Bytes | Cost  (%CPU)| Time       | Pstart| Pstop |
---------------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |         |       |       |    19  (100)|            |       |       |
|  1 |  SORT AGGREGATE              |         |     1 |    28 |             |            |       |       |
|* 2 |   HASH JOIN                  |         |     2 |    56 |    19  (100)| 00:00:01   |       |       |
|  3 |    PART JOIN FILTER CREATE   | :BF0000 |     2 |    32 |    13    (8)| 00:00:01   |       |       |
|* 4 |     TABLE ACCESS FULL        | TIMES   |     2 |    32 |    13    (8)| 00:00:01   |       |       |
|  5 |    PARTITION RANGE JOIN-FILTER|        |   960 | 11520 |     5    (0)| 00:00:01   |:BF0000|:BF0000|
|  6 |     TABLE ACCESS FULL        | SALES   |   960 | 11520 |     5    (0)| 00:00:01   |:BF0000|:BF0000|
---------------------------------------------------------------------------------------------------------

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

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter(("T"."CALENDAR_MONTH_DESC"='MAR-2014' OR "T"."CALENDAR_MONTH_DESC"='APR-2014'
              OR "T"."CALENDAR_MONTH_DESC"='MAY-2014'))

27 rows selected.
```

“AND” Pruning

Dynamic pruning

Static pruning

```
FROM sales s, times t …
WHERE s.time_id = t.time_id ..
AND t.fiscal_year in (2014,2015)
AND s.time_id
   between TO_DATE('01-JAN-2014','DD-MON-YYYY')
   and TO_DATE('01-JAN-2015','DD-MON-YYYY')
```

- All predicates on partition key will used for pruning
  - Dynamic and static predicates will now be used combined
- Example:
  - Star transformation with pruning predicate on both the FACT table and a dimension

ORACLE

“AND” Pruning

Sample Plan

```
Plan hash value: 552669211

--------------------------------------------------------------------------------------------------------
| Id | Operation                   | Name      | Rows  | Bytes | Cost (%CPU)| Time       | Pstart| Pstop |
--------------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT            |           |     1 |    24 |   17   (12)| 00:00:01 |        |        |
|  1 |  SORT AGGREGATE             |           |     1 |    24 |            |          |        |        |
|* 2 |   HASH JOIN                 |           |   204 |  4896 |   17   (12)| 00:00:01 |        |        |
|  3 |    PART JOIN FILTER CREATE| :BF0000   |   185 |  2220 |   13    (8)| 00:00:01 |        |        |
|* 4 |     TABLE ACCESS FULL       | TIMES     |   185 |  2220 |   13    (8)| 00:00:01 |        |        |
|  5 |    PARTITION RANGE AND      |           |   313 |  3756 |    3    (0)| 00:00:01 |KEY(AP)|KEY(AP)|
|* 6 |     TABLE ACCESS FULL       | SALES     |   313 |  3756 |    3    (0)| 00:00:01 |KEY(AP)|KEY(AP)|
--------------------------------------------------------------------------------------------------------

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

   2 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."TIME_ID"<=TO_DATE(' 2015-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              ("T"."FISCAL_YEAR"=2014 OR "T"."FISCAL_YEAR"=2015) AND "T"."TIME_ID">=TO_DATE(' 2014-01-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
   6 - filter("S"."TIME_ID"<=TO_DATE(' 2015-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

22 rows selected.
```

Ensuring Partition Pruning

Don’t use functions on partition key filter predicates

```
SELECT sum(amount_sold)
FROM sh.sales s, sh.times t
WHERE s.time_id = t.time_id
AND TO_CHAR(s.time_id, 'YYYYMMDD') between '20140101' and '20150101'

Plan hash value: 672559287

----------------------------------------------------------------------------------------------
| Id | Operation              | Name   | Rows   | Bytes | Cost (%CPU)| Time       | Pstart| Pstop |
----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |        |        |       |    6  (100)|            |       |       |
|  1 |  SORT AGGREGATE        |        |     1  |    12 |            |            |       |       |
|  2 |   PARTITION RANGE ALL|          |     2  |    24 |    6   (17)| 00:00:01   |     1 |    16 |
|* 3 |    TABLE ACCESS FULL | SALES    |     2  |    24 |    6   (17)| 00:00:01   |     1 |    16 |
----------------------------------------------------------------------------------------------

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

   3 - filter((TO_CHAR(INTERNAL_FUNCTION("S"."TIME_ID"),'YYYYMMDD')>='20140101' AND
              TO_CHAR(INTERNAL_FUNCTION("S"."TIME_ID"),'YYYYMMDD')<='20150101'))

23 rows selected.
```

ORACLE

Ensuring Partition Pruning

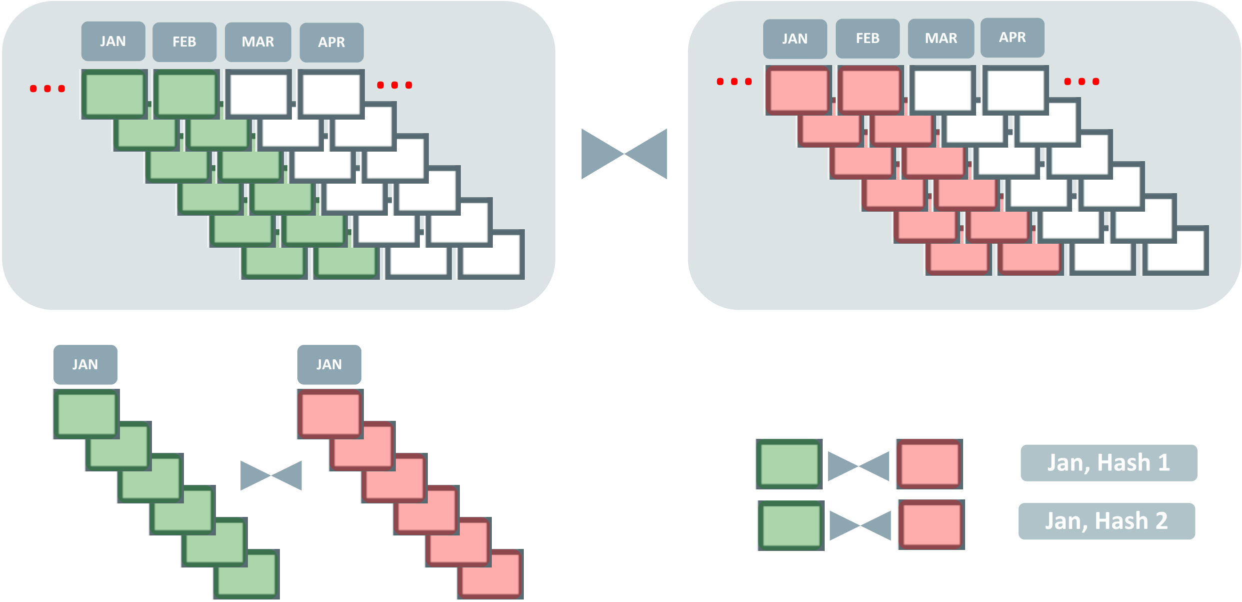
Don’t use functions on partition key filter predicates

```
SELECT sum(amount_sold)
FROM sh.sales s, sh.times t
WHERE s.time_id = t.time_id
AND TO_CHAR(s.time_id, 'YYYYMMDD') between '20140101' and '20150101'

Plan hash value: 672559287

------------------
| Pstart| Pstop |
------------------
|       |       |
|       |       |
|     1 |    16 |
|     1 |    16 |
------------------

01' AND
1'))
```

```
SELECT sum(amount_sold)
FROM sh.sales s, sh.times t
WHERE s.time_id = t.time_id
AND s.time_id between TO_DATE('20140101','YYYYMMDD') and TO_DATE('20150101','YYYYMMDD')

Plan hash value: 2025449199

---------------------------------------------------------------------------------------------
| Id | Operation                 | Name  | Rows  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
---------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT          |       |       |       |    3  (100)|           |       |       |
|  1 |  SORT AGGREGATE           |       |     1 |    12 |            |           |       |       |
|  2 |   PARTITION RANGE ITERATOR|       |   313 |  3756 |    3     (0)| 00:00:01 |     9 |    13 |
|* 3 |    TABLE ACCESS FULL      | SALES |   313 |  3756 |    3     (0)| 00:00:01 |     9 |    13 |
---------------------------------------------------------------------------------------------

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

   3 - filter("S"."TIME_ID"<=TO_DATE(' 2015-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

22 rows selected.
```

Partition-wise Joins

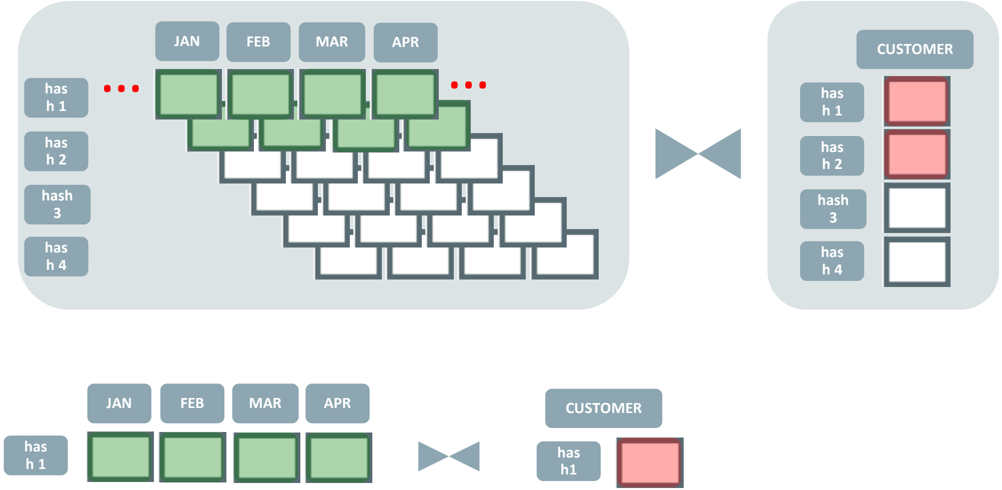
Partition pruning and PWJ's "at work"



- A large join is divided into multiple smaller joins, executed in parallel
  - # of partitions to join must be a multiple of DOP
  - Both tables must be partitioned the same way on the join column

Partition-wise Joins

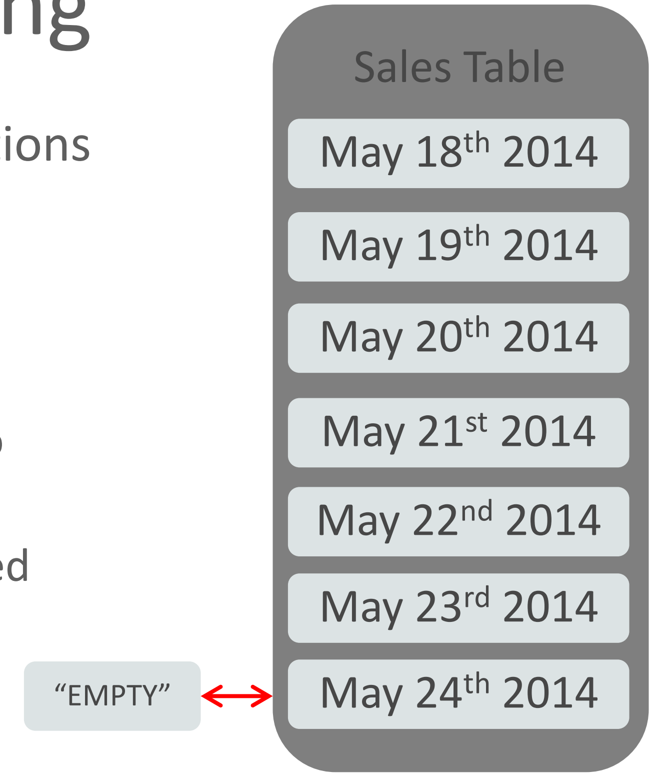
Partition pruning and PWJ's "at work"



- A large join is divided into multiple smaller joins, executed in parallel
  - \# of partitions to join must be a multiple of DOP
  - Both tables must be partitioned the same way on the join column

Partition Purging and Loading

- Remove and add data as metadata only operations
  - Exchange the metadata of partitions
- Exchange standalone table w/ arbitrary single partition
  - Data load: standalone table contains new data to being loaded
  - Data purge: partition containing data is exchanged with empty table
- Drop partition alternative for purge
  - Data is gone forever



Partitioning Maintenance

Partition Maintenance

Fundamental Concepts for Success

- While performance seems to be the most visible one, don't forget about the rest, e.g.
  - Partitioning must address all business-relevant areas of Performance, Manageability, and Availability
- Partition autonomy is crucial
  - Fundamental requirement for any partition maintenance operations
  - Acknowledge partitions as metadata in the data dictionary

ORACLE

Partition Maintenance

Fundamental Concepts for Success

- Provide full partition autonomy
  - Use local indexes whenever possible
  - Enable partition all table-level operations for partitions, e.g. TRUNCATE, MOVE, COMPRESS
- Make partitions visible and usable for database administration
  - Partition naming for ease of use
- Maintenance operations must be partition-aware
  - Also true for indexes
- Maintenance operations must not interfere with online usage of a partitioned table

ORACLE

Partition Maintenance

Table Partition Maintenance Operations

```
ALTER TABLE ADD PARTITION(S)
ALTER TABLE DROP PARTITION(S)
ALTER TABLE EXCHANGE PARTITION
ALTER TABLE MODIFY PARTITION
ALTER TABLE MOVE PARTITION [PARALLEL]
ALTER TABLE RENAME PARTITION
ALTER TABLE MOVE PARTITION [PARALLEL]
ALTER TABLE SPLIT PARTITION [PARALLEL]
ALTER TABLE MERGE PARTITION(S) [PARALLEL]
ALTER TABLE COALESCE PARTITION [PARALLEL]
ALTER TABLE ANALYZE PARTITION
ALTER TABLE TRUNCATE PARTITION(S)
Export/Import [by partition]
Transportable tablespace [by partition]
```

Index Maintenance Operations

```
ALTER INDEX MODIFY PARTITION
ALTER INDEX DROP PARTITION(S)
ALTER INDEX REBUILD PARTITION
ALTER INDEX RENAME PARTITION
ALTER INDEX RENAME
ALTER INDEX SPLIT PARTITION
ALTER INDEX ANALYZE PARTITION
```

All partitions remain available all the time

- DML Lock on impacted partitions
- Move partition online no lock at all

ORACLE

Partition Maintenance on Multiple Partitions

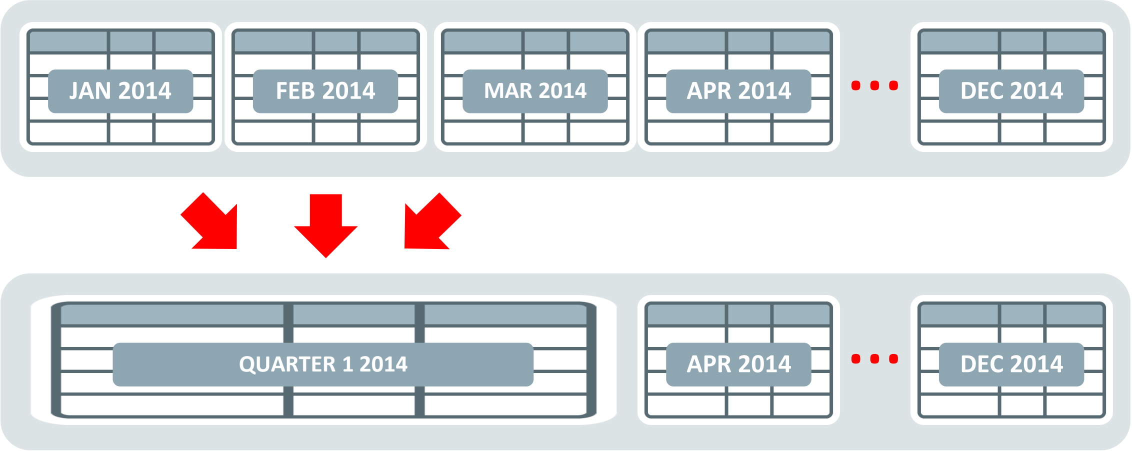
Introduced in Oracle 12c

ORACLE

Enhanced Partition Maintenance Operations

Operate on multiple partitions

- Partition Maintenance on multiple partitions in a single operation
- Full parallelism
- Transparent maintenance of local and global indexes




```
ALTER TABLE orders
MERGE PARTITIONS Jan2014, Feb2014, Mar2014
INTO PARTITION Quarter1_2014 COMPRESS FOR ARCHIVE HIGH;
```

ORACLE

Enhanced Partition Maintenance Operations

Operate on multiple partitions

- Specify multiple partitions in order

```
SQL > alter table pt merge partitions for (5), for (15), for (25) into partition p30;
Table altered.
```

- Specify a range of partitions

```
SQL > alter table pt merge partitions part10 to part30 into partition part30;
Table altered.
```

```
SQL > alter table pt split partition p30
   into
2  (partition p10 values less than (10),
3   partition p20 values less than (20),
4   partition p30);
Table altered.
```

- Works for all PMOPS
  - Supports optimizations like fast split

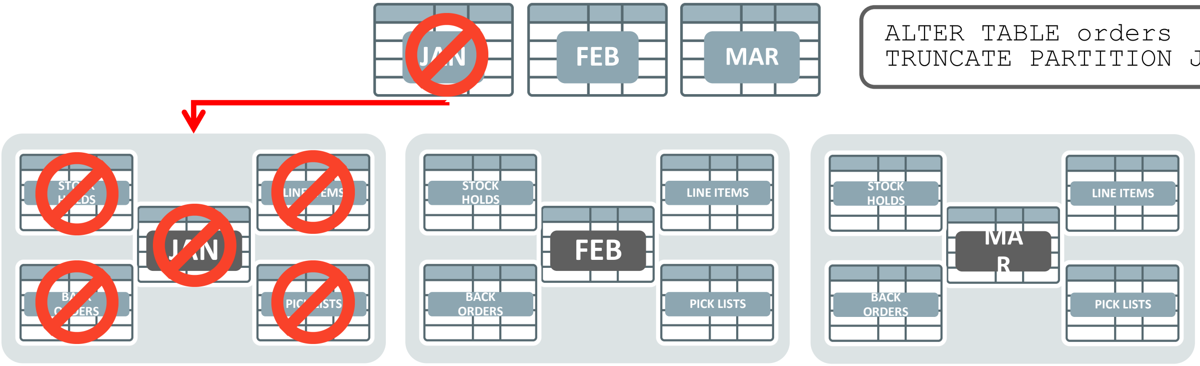
Cascading Truncate and Exchange for Reference Partitioning

Introduced in Oracle 12c

ORACLE

Advanced Partitioning Maintenance

Cascading TRUNCATE and EXCHANGE PARTITION

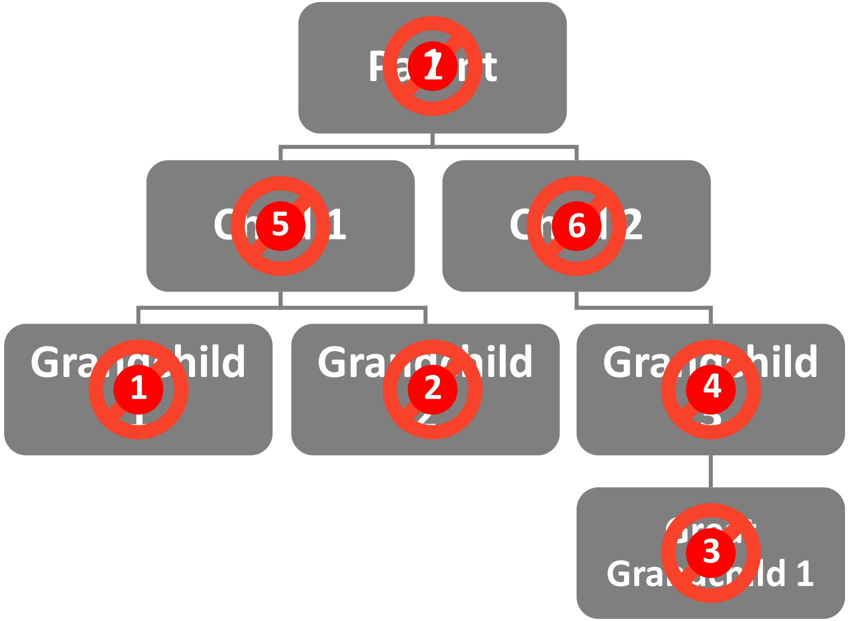


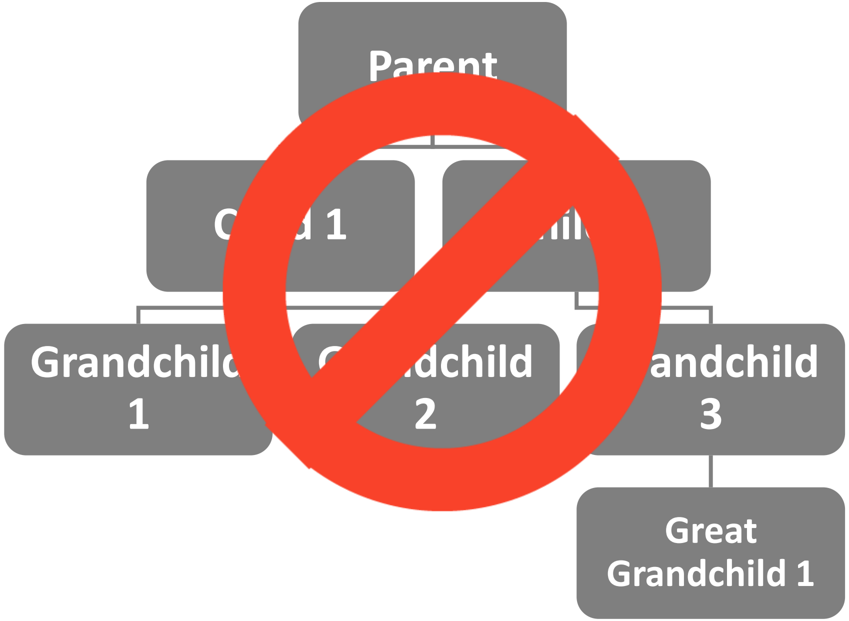

```
ALTER TABLE orders
TRUNCATE PARTITION Jan2015 CASCADE;
```

- Cascading TRUNCATE and EXCHANGE for improved business continuity
- Single atomic transaction preserves data integrity
- Simplified and less error prone code development

ORACLE

Cascading TRUNCATE PARTITION





- Proper bottom-up processing required
- Seven individual truncate operations

- One truncate operation

Cascading TRUNCATE PARTITION

```
SQL> create table intRef_p (pkcol number not null, col2 varchar2(200),
  2                          constraint pk_intref primary key (pkcol))
  3  partition by range (pkcol) interval (10)
  4  (partition p1 values less than (10));

Table created.

SQL>
SQL> create table intRef_c1 (pkcol number not null, col2 varchar2(200), fkcol number not null,
  2                           constraint pk_c1 primary key (pkcol),
  3                           constraint fk_c1 foreign key (fkcol) references intRef_p(pkcol) ON DELETE CASCADE)
  4  partition by reference (fk_c1);

Table created.
```

Cascading TRUNCATE PARTITION

```
SQL> create table intRef_p (pkcol nu
  2                        constrai
  3  partition by range (pkcol) inte
  4  (partition p1 values less than

Table created.

SQL>
SQL> create table intRef_c1 (pkcol n
  2                          constra
  3                          constra
  4  partition by reference (fk_c1);

Table created.
```

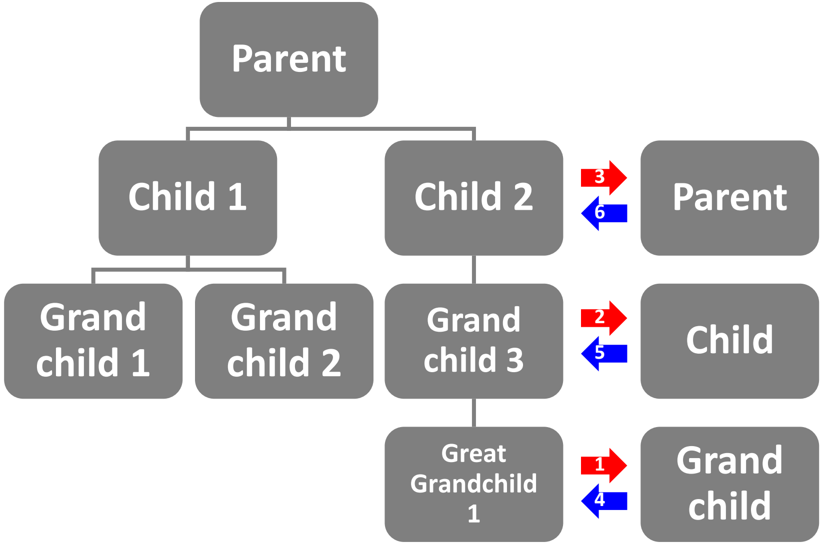
```
SQL> select * from intRef_p;

     PKCOL COL2
---------- ------------------------------------
       333  data for truncate - p
       999  data for truncate - p

SQL> select * from intRef_c1;

     PKCOL COL2                                      FKCOL
---------- ------------------------------------ ----------
      1333  data for truncate - c1                     333
      1999  data for truncate - c1                     999

SQL> alter table intRef_p truncate partition for (999) cascade update indexes;

Table truncated.

SQL> select * from intRef_p;

     PKCOL COL2
---------- ------------------------------------
       333  data for truncate - p

SQL> select * from intRef_c1;

     PKCOL COL2                                      FKCOL
---------- ------------------------------------ ----------
      1333  data for truncate - c1                     333
```

ORACLE

Cascading TRUNCATE PARTITION

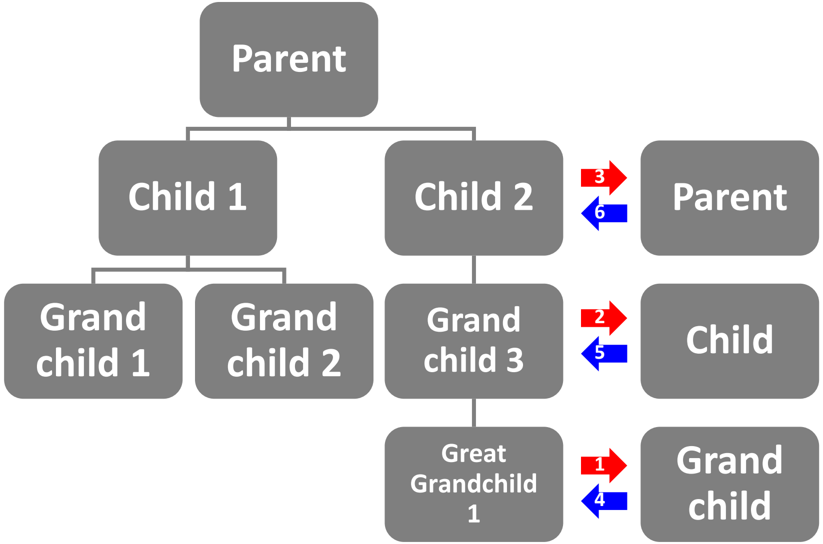
- CASCADE applies for whole reference tree
  - Single atomic transaction, all or nothing
  - Bushy, deep, does not matter
  - Can be specified on any level of a reference-partitioned table
- ON DELETE CASCADE for all foreign keys required
- Cascading TRUNCATE available for non-partitioned tables as well
  - Dependency tree for non-partitioned tables can be interrupted with disabled foreign key constraints
- Reference-partitioned hierarchy must match for target and table to-be-exchanged
- For bushy trees with multiple children on the same level, each child on a given level must reference to a different key in the parent table
  - Required to unambiguously pair tables in the hierarchy tree

Cascading EXCHANGE PARTITION

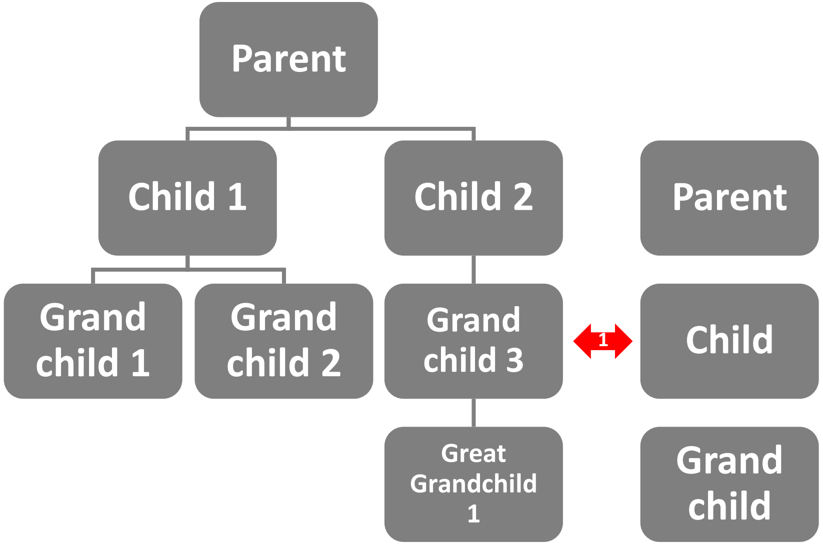


- Exchange (clear) out of target bottom-up
- Exchange (populate) into target top-down

Cascading EXCHANGE PARTITION



- Exchange (clear) out of target bottom-up
- Exchange (populate) into target top-down



- Exchange complete hierarchy tree
- One exchange operation

Cascading EXCHANGE PARTITION

```
SQL> create table intRef_p (pkcol number not null, col2 varchar2(200),
  2                        constraint pk_intref primary key (pkcol))
  3  partition by range (pkcol) interval (10)
  4  (partition p1 values less than (10));

SQL> create table intRef_c1 (pkcol number not null, col2 varchar2(200), fkcol number not null,
  2                         constraint pk_c1 primary key (pkcol),
  3                         constraint fk_c1 foreign key (fkcol) references intRef_p(pkcol) ON DELETE CASCADE)
  4  partition by reference (fk_c1);

SQL> create table intRef_gc1 (col1 number not null, col2 varchar2(200), fkcol number not null,
  2                          constraint fk_gc1 foreign key (fkcol) references intRef_c1(pkcol) ON DELETE CASCADE)
  3  partition by reference (fk_gc1);
```

Cascading EXCHANGE PARTITION

```
SQL> REM create some PK-FK equivalent table construct for exchange
SQL> create table XintRef_p (pkcol number not null, col2 varchar2(200),
  2                         constraint xpk_intref primary key (pkcol));

SQL> create table XintRef_c1 (pkcol number not null, col2 varchar2(200), fkcol number not null,
  2                          constraint xpk_c1 primary key (pkcol),
  3                          constraint xfk_c1 foreign key (fkcol) references XintRef_p(pkcol) ON DELETE CASCADE);

SQL> create table XintRef_gc1 (col1 number not null, col2 varchar2(200), fkcol number not null,
  2                           constraint xfk_gc1 foreign key (fkcol) references XintRef_c1(pkcol) ON DELETE CASCADE);
```

ORACLE

Cascading EXCHANGE PARTITION

```
SQL> select * from intRef_p;

     PKCOL COL2
---------- -----------------------------------
       333 p333 - data BEFORE exchange - p
       999 p999 - data BEFORE exchange - p

SQL> select * from intRef_c1;

     PKCOL COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333 p333 - data BEFORE exchange - c1           333
      1999 p999 - data BEFORE exchange - c1           999

SQL> select * from intRef_gc1;

      COL1 COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333 p333 - data BEFORE exchange - gc1         1333
      1999 p999 - data BEFORE exchange - gc1         1999
```

```
SQL> select * from XintRef_p;

     PKCOL COL2
---------- -----------------------------------
       333 p333 - data AFTER exchange - p

SQL> select * from XintRef_c1;

     PKCOL COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333 p333 - data AFTER exchange - c1            333

SQL> select * from XintRef_gc1;

      COL1 COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333 p333 - data AFTER exchange - gc1          1333
```

ORACLE

Cascading EXCHANGE PARTITION

```
SQL> alter table intRef_p exchange partition for (333) with table XintRef_p cascade update indexes;

Table altered.
```

Cascading EXCHANGE PARTITION

```
SQL> select * from intRef_p;

     PKCOL COL2
---------- -----------------------------------
       333  p333 - data AFTER exchange - p
       999  p999 - data BEFORE exchange - p

SQL> select * from intRef_c1;

     PKCOL COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333  p333 - data AFTER exchange - c1           333
      1999  p999 - data BEFORE exchange - c1          999

SQL> select * from intRef_gc1;

      COL1 COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333  p333 - data AFTER exchange - gc1         1333
      1999  p999 - data BEFORE exchange - gc1        1999
```

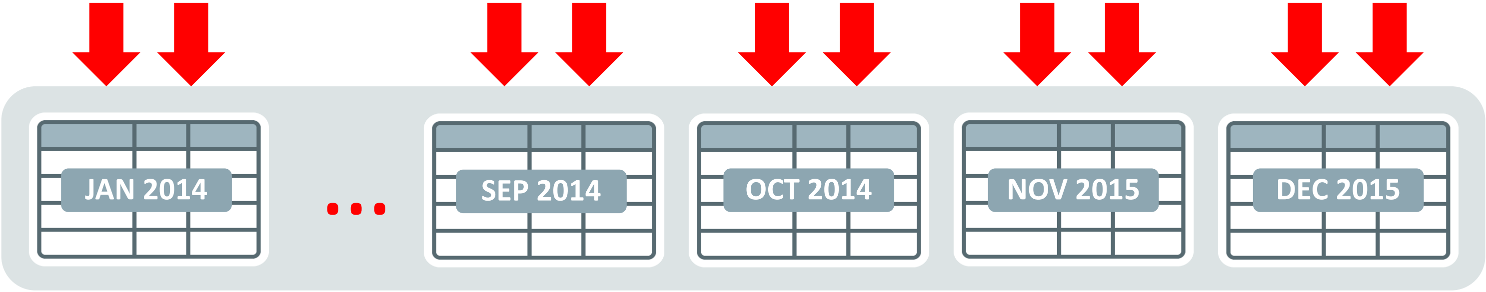
```
SQL> select * from XintRef_p;

     PKCOL COL2
---------- -----------------------------------
       333  p333 - data BEFORE exchange - p

SQL> select * from XintRef_c1;

     PKCOL COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333  p333 - data BEFORE exchange - c1          333

SQL> select * from XintRef_gc1;

      COL1 COL2                                     FKCOL
---------- ----------------------------------- ----------
      1333  p333 - data BEFORE exchange - gc1        1333
```

Online Move Partition

Introduced in Oracle 12c

Enhanced Partition Maintenance Operations

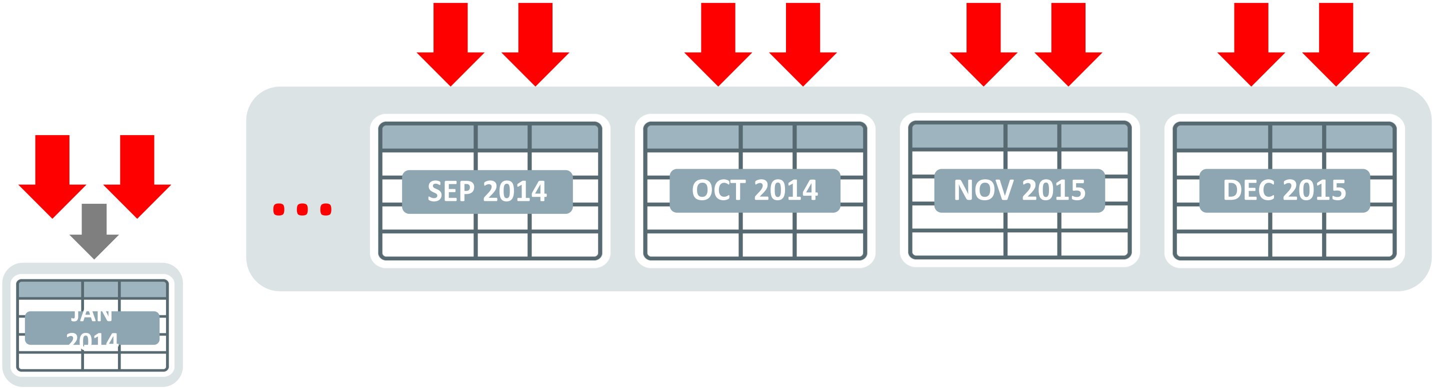
Online Partition Move



- Transparent MOVE PARTITION ONLINE operation
- Concurrent DML and Query
- Index maintenance for local and global indexes

Enhanced Partition Maintenance Operations

Online Partition Move



- Transparent MOVE PARTITION ONLINE operation
- Concurrent DML and Query
- Index maintenance for local and global indexes

Enhanced Partition Maintenance Operations

Online Partition Move – Best Practices

- Minimize concurrent DML operations if possible
  - Require additional disk space and resources for journaling
  - Journal will be applied recursively after initial bulk move
  - The larger the journal, the longer the runtime
- Concurrent DML has impact on compression efficiency
  - Best compression ratio with initial bulk move

ORACLE

Asynchronous Global Index Maintenance

Introduced in Oracle 12c

Enhanced Partition Maintenance Operations

Asynchronous Global Index Maintenance

- Usable global indexes after DROP and TRUNCATE PARTITION without index maintenance
  – Affected partitions are known internally and filtered out at data access time
- DROP and TRUNCATE become fast, metadata-only operations
  – Significant speedup and reduced initial resource consumption
- Delayed Global index maintenance
  – Deferred maintenance through ALTER INDEX REBUILD|COALESCE
  – Automatic cleanup using a scheduled job

ORACLE

Enhanced Partition Maintenance Operations

Asynchronous Global Index Maintenance

Before

```
SQL> select count(*) from pt partition for (9999);

  COUNT(*)
----------
  25341440

Elapsed: 00:00:01.00
SQL> select index_name, status, orphaned_entries from user_indexes;

INDEX_NAME                     STATUS   ORPHANED_ENTRIES
------------------------------ -------- --------------------
I1_PT                          VALID    NO

Elapsed: 00:00:01.04
SQL>
SQL> alter table pt drop partition for (9999) update indexes;

Table altered.

Elapsed: 00:02:04.52
SQL>
SQL> select index_name, status, orphaned_entries from user_indexes;

INDEX_NAME                     STATUS   ORPHANED_ENTRIES
------------------------------ -------- --------------------
I1_PT                          VALID    NO

Elapsed: 00:00:00.10
```

After

```
SQL> select count(*) from pt partition for (9999);

  COUNT(*)
----------
  25341440

Elapsed: 00:00:00.98
SQL> select index_name, status, orphaned_entries from user_indexes;

INDEX_NAME                     STATUS   ORPHANED_ENTRIES
------------------------------ -------- --------------------
I1_PT                          VALID    NO

Elapsed: 00:00:00.33
SQL>
SQL> alter table pt drop partition for (9999) update indexes;

Table altered.

Elapsed: 00:00:00.04
SQL>
SQL> select index_name, status, orphaned_entries from user_indexes;

INDEX_NAME                     STATUS   ORPHANED_ENTRIES
------------------------------ -------- --------------------
I1_PT                          VALID    YES

Elapsed: 00:00:00.05
```

ORACLE

Statistics Management for Partitioning

Statistics Gathering

- You must gather Optimizer statistics
  - Using dynamic sampling is not an adequate solution
  - Statistics on global and partition level recommended
    - Subpartition level optional
- Run all queries against empty tables to populate column usage
  - This helps identify which columns automatically get histograms created on them
- Optimizer statistics should be gathered after the data has been loaded but before any indexes are created
  - Oracle will automatically gather statistics for indexes as they are being created

ORACLE

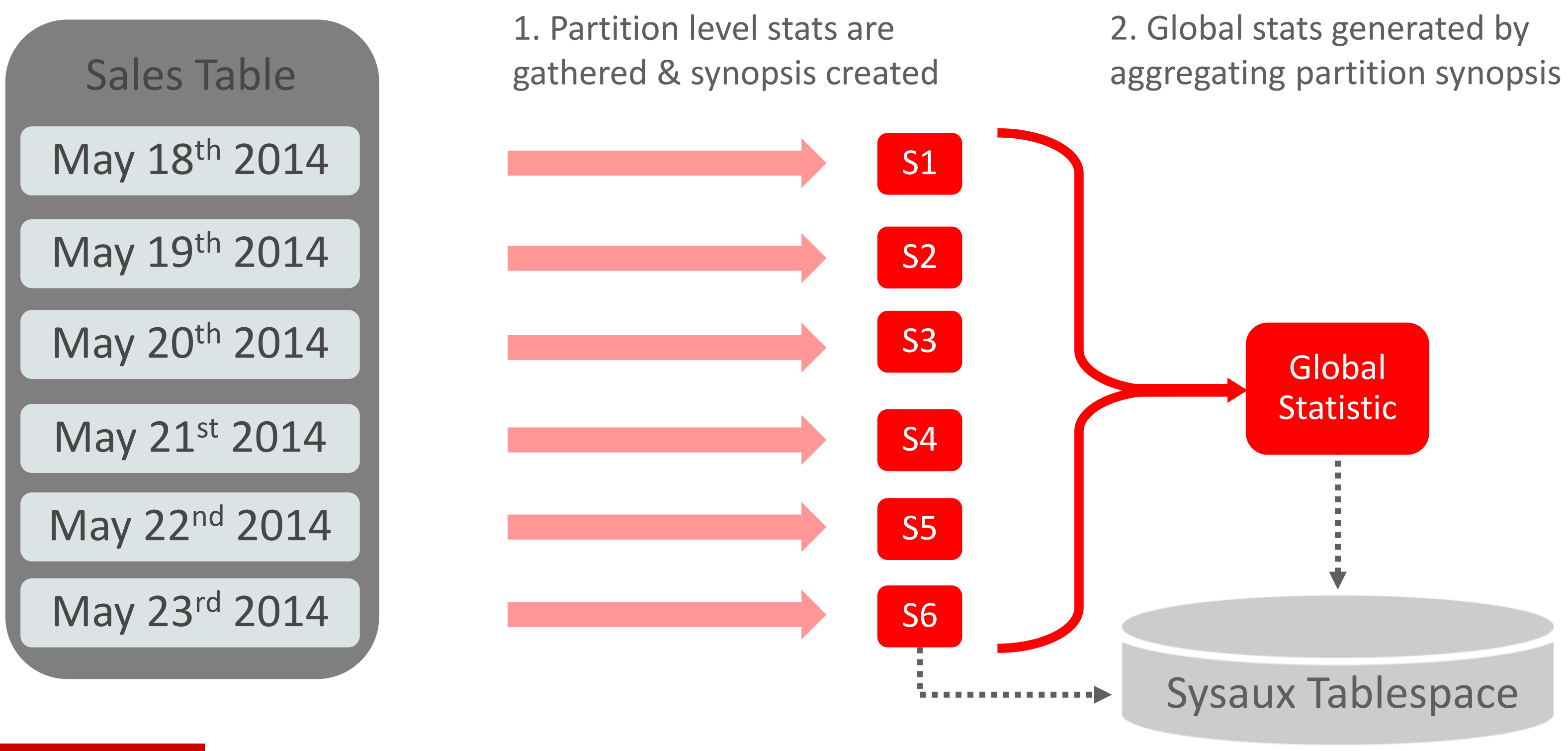
Statistics Gathering

- By default DBMS_STATS gathers the following stats for each table
  - global (table level), partition level, sub-partition level
- Optimizer uses global stats if query touches two or more partitions
- Optimizer uses partition stats if queries do partition elimination and only one partition is necessary to answer the query
  - If queries touch two or more partitions the optimizer will use a combination of global and partition level statistics
- Optimizer uses sub-partition level statistics only if your queries do partition elimination and one sub-partition is necessary to answer query

Efficient Statistics Management

- Use AUTO_SAMPLE_SIZE
  - The only setting that enables new efficient statistics collection
  - Hash based algorithm, scanning the whole table
    - Speed of sampling, accuracy of compute
- Enable incremental global statistics collection
  - Avoids scan of all partitions after changing single partitions
    - Prior to 11.1, scan of all partitions necessary for global stats
  - Managed on per table level
    - Static setting
  - Create synopsis for non-partitioned table to being exchanged (Oracle Database 12c)

Incremental Global Statistics



Incremental Global Statistics Cont'd

Sales Table

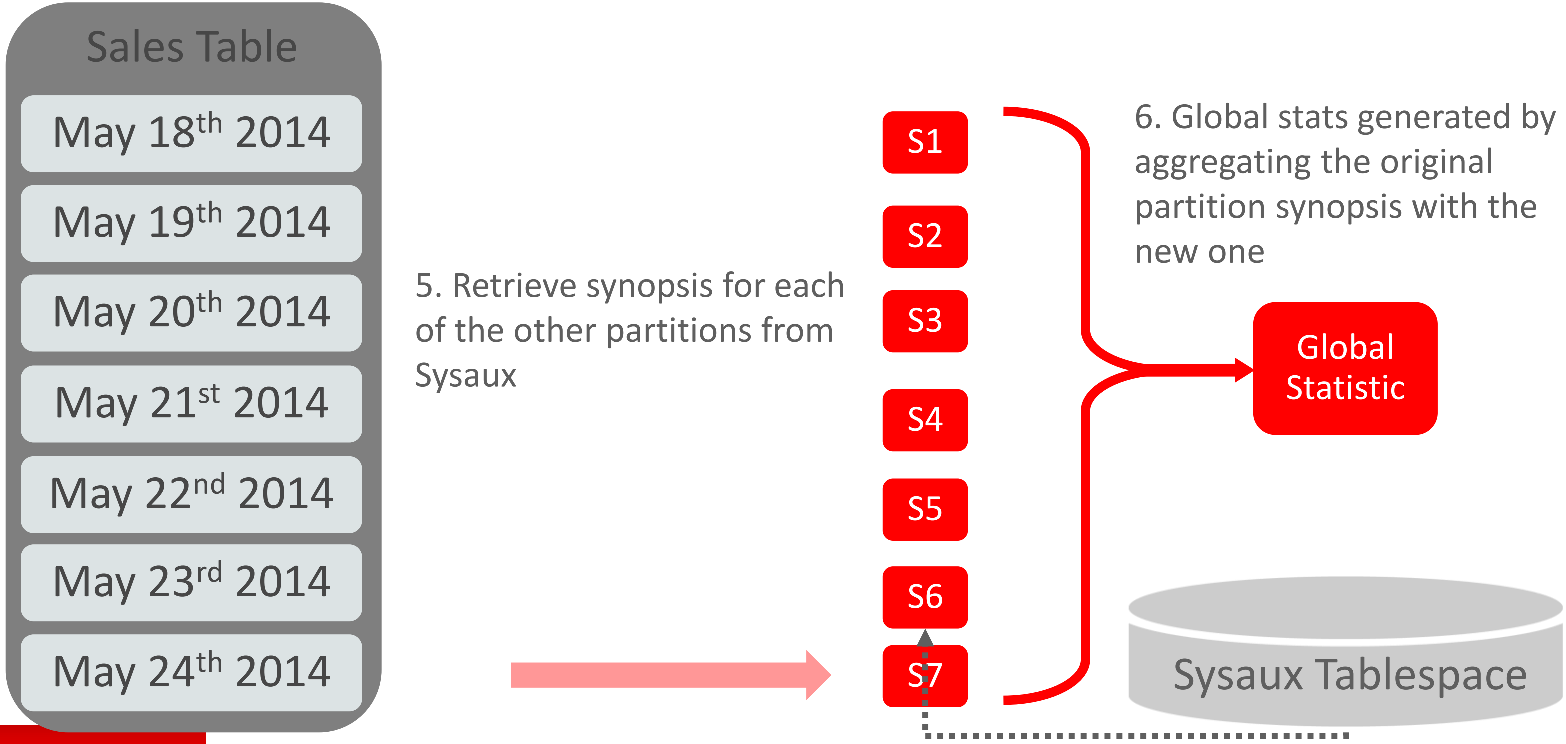
- May 18th 2014
- May 19th 2014
- May 20th 2014
- May 21st 2014
- May 22nd 2014
- May 23rd 2014
- May 24th 2014

3. A new partition is added to the table and data is loaded

4. Gather partition statistics for new partition

S7

Incremental Global Statistics Cont'd



Step necessary to gather accurate statistics

- Turn on incremental feature for the table

```
EXEC DBMS_STATS.SET_TABLE_PREFS('SH','SALES','INCREMENTAL','TRUE');
```

- After load gather table statistics using GATHER_TABLE_STATS
  - No need to specify parameters

```
EXEC DBMS_STATS.GATHER_TABLE_STATS('SH','SALES');
```

- The command will collect statistics for partitions and update the global statistics based on the partition level statistics and synopsis
- Possible to set incremental to true for all tables
  - Only works for already existing tables

```
EXEC DBMS_STATS.SET_GLOBAL_PREFS('INCREMENTAL','TRUE');
```

Partitioning and Unusable Indexes

Unusable Indexes

- Unusable index partitions are commonly used in environments with fast load requirements
  - "Save" the time for index maintenance at data insertion
  - Unusable index segments do not consume any space (11.2)
- Unusable indexes are ignored by the optimizer

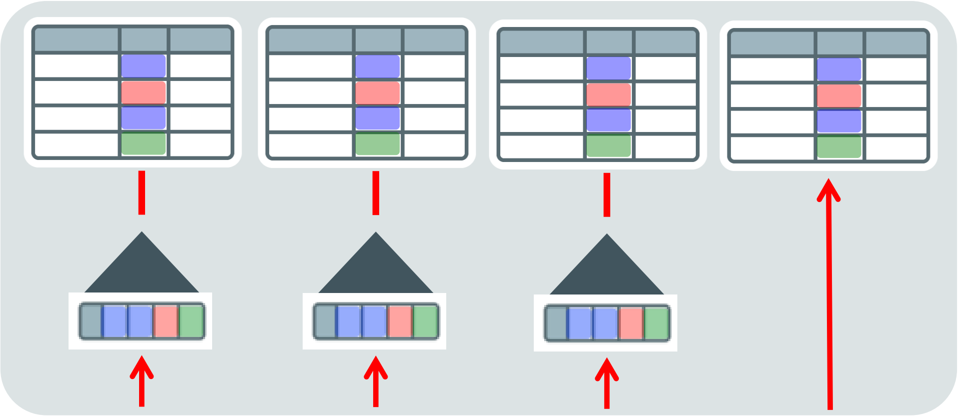
```
SKIP_UNUSABLE_INDEXES = [TRUE | FALSE ]
```

- Partitioned indexes can be used by the optimizer even if some partitions are unusable
  - Prior to 11.2, static pruning and only access of usable index partitions mandatory
  - With 11.2, intelligent rewrite of queries using UNION ALL

ORACLE

Table-OR-Expansion

Multiple SQL branches are generated and executed



- Intelligent UNION ALL expansion in the presence of partially unusable indexes
  - Transparent internal rewrite
  - Usable index partitions will be used
  - Full partition access for unusable index partitions

ORACLE

Table-OR-Expansion

Sample Plan - Multiple SQL branches are generated and executed

```
 select count(*) from toto where name ='FOO' and rn between 1300 and
1400

Plan hash value: 2830852558

-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name     | Rows  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |          |       |       |   27M(100)|           |       |       |
|   1 |  SORT AGGREGATE                     |          |     1 |    21 |            |           |       |       |
|   2 |   VIEW                              | VW_TE_2  |     2 |       |   27M   (3)| 92:15:22  |       |       |
|   3 |    UNION-ALL                        |          |       |       |            |           |       |       |
|   4 |     PARTITION RANGE SINGLE          |          |     1 |    20 |    2   (0)| 00:00:01  |    14 |    14 |
|   5 |      TABLE ACCESS BY LOCAL INDEX ROWID| TOTO   |     1 |    20 |    2   (0)| 00:00:01  |    14 |    14 |
|*  6 |       INDEX RANGE SCAN              | I_TOTO   |     1 |       |    1   (0)| 00:00:01  |    14 |    14 |
|   7 |     PARTITION RANGE SINGLE          |          |     1 |    22 |   27M   (3)| 92:15:22  |    15 |    15 |
|*  8 |      TABLE ACCESS FULL              | TOTO     |     1 |    22 |   27M   (3)| 92:15:22  |    15 |    15 |
-----------------------------------------------------------------------------------------------------------------

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

   6 - access("NAME"='FOO')
   8 - filter(("NAME"='FOO' AND "TOTO"."RN"=1400))


27 rows selected.
```

ORACLE

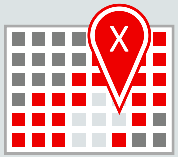
Attribute Clustering and Zone Maps

Introduced in Oracle 12c (Release 12.102)

Zone Maps with Attribute Clustering

Attribute Clustering

Orders data so that columns values are stored together on disk



Zone maps

Stores min/max of specified columns per zone

Used to filter un-needed data during query execution

- Combined Benefits
- Improved query performance and concurrency
  - Reduced physical data access
  - Significant IO reduction for highly selective operations
- Optimized space utilization
  - Less need for indexes
  - Improved compression ratios through data clustering
- Full application transparency
  - Any application will benefit

Attribute Clustering

Concepts and Benefits

- Orders data so that it is in close proximity based on selected columns values: “attributes”
- Attributes can be from a single table or multiple tables
  - e.g. from fact and dimension tables
- Significant IO pruning when used with zone maps
- Reduced block IO for table lookups in index range scans
- Queries that sort and aggregate can benefit from pre-ordered data
- Enable improved compression ratios
  - Ordered data is likely to compress more than unordered data

Attribute Clustering for Zone Maps

Ordered rows

```
ALTER TABLE sales
ADD CLUSTERING BY
LINER ORDER (category);

ALTER TABLE sales MOVE;
```

Category	Country
BOYS	AR
BOYS	JP
BOYS	SA
BOYS	US
GIRLS	AR
GIRLS	JP
GIRLS	SA
GIRLS	US
MEN	AR
MEN	JP
MEN	SA
MEN	US
WOMEN	AR
WOMEN	JP
WOMEN	SA
WOMEN	US

- Ordered rows containing category values BOYS, GIRLS and MEN.
- *Zone maps* catalogue regions of rows, or *zones,* that contain particular column value ranges.
- By default, each zone is up to 1024 blocks.
- For example, we only need to scan this zone if we are searching for category “GIRLS”. We can skip all other zones.

ORACLE

Attribute Clustering

Basics

- Two types of attribute clustering
  - LINEAR ORDER BY
    - Classical ordering
  - INTERLEAVED ORDER BY
    - Multi-dimensional ordering
- Simple attribute clustering on a single table
- Join attribute clustering
  - Cluster on attributes derived through join of multiple tables
    - Up to four tables
    - Non-duplicating join (PK or UK on joined table is required)

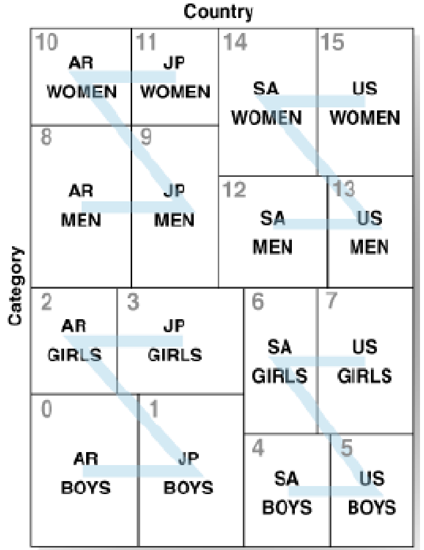
ORACLE

Attribute Clustering

Example

- CLUSTERING BY LINEAR ORDER (category, country)
- CLUSTERING BY INTERLEAVED ORDER (category, country)

Category	Country
BOYS	AR
BOYS	JP
BOYS	SA
BOYS	US
GIRLS	AR
GIRLS	JP
GIRLS	SA
GIRLS	US
MEN	AR
MEN	JP
MEN	SA
MEN	US
WOMEN	AR
WOMEN	JP
WOMEN	SA
WOMEN	US



- LINEAR ORDER

INTERLEAVED ORDER

Attribute Clustering

Basics

- Clustering directive specified at table level
  - ALTER TABLE ... ADD CLUSTERING ...
- Directive applies to new data and data movement
- Direct path operations
  - INSERT APPEND, MOVE, SPLIT, MERGE
  - Does not apply to conventional DML
- Can be enabled and disabled on demand
  - Hints and/or specific syntax

Zone Maps

Concepts and Basics

- Stores minimum and maximum of specified columns
  - Information stored per zone
  - [Sub]Partition-level rollup information for partitioned tables for multi-dimensional partition pruning
- Analogous to a coarse index structure
  - Much more compact than an index
  - Zone maps filter out what you don't need, indexes find what you do need
- Significant performance benefits with complete application transparency
  - IO reduction for table scans with predicates on the table itself or even a joined table using join zone maps (a.k.a. "hierarchical zone map")
- Benefits are most significant with ordered data
  - Used in combination with attribute clustering or data that is naturally ordered

ORACLE

Zone Maps

Basics

- Independent access structure built for a table
  - Implemented using a type of materialized view
  - For partitioned and non-partitioned tables
- One zone map per table
  - Zone map on partitioned table includes aggregate entry per [sub]partition
- Used transparently
  - No need to change or hint queries
- Implicit or explicit creation and column selection
  - Through Attribute Clustering: CREATE TABLE … CLUSTERING
  - CREATE MATERIALIZED ZONEMAP … AS SELECT …

Attribute Clustering With Zone Maps

- CLUSTERING BY LINEAR ORDER (category, country)
  - Zone map benefits are most significant with ordered data

Category	Country
BOYS	AR
BOYS	JP
BOYS	SA
BOYS	US
GIRLS	AR
GIRLS	JP
GIRLS	SA
GIRLS	US
MEN	AR
MEN	JP
MEN	SA
MEN	US
WOMEN	AR
WOMEN	JP
WOMEN	SA
WOMEN	US

- **LINEAR ORDER**

Pruning with:

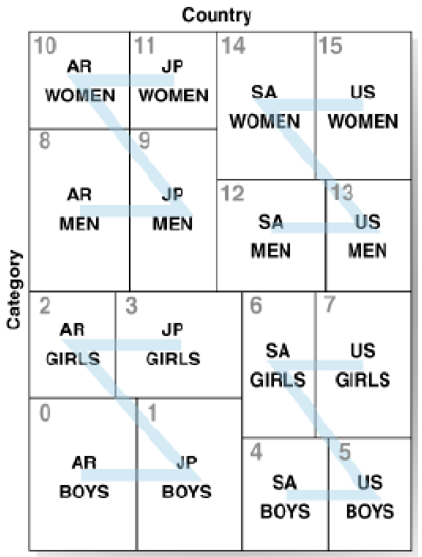
```
SELECT ..
FROM table
WHERE category =
   'BOYS';
```

```
SELECT ..
FROM table
WHERE category =
   'BOYS';
AND country = 'US';
```

ORACLE

Attribute Clustering With Zone Maps

- CLUSTERING BY INTERLEAVED ORDER (category, country)
  - Zone map benefits are most significant with ordered data



- **INTERLEAVED ORDER**

Pruning with:

```
SELECT ..
FROM table
WHERE category =
    'BOYS';
```

```
SELECT ..
FROM table
AND country = 'US';
```

```
SELECT ..
FROM table
WHERE category =
    'BOYS'
AND country = 'US';
```

Zone Maps

Staleness

- DML and partition operations can cause zone maps to become fully or partially stale
  - Direct path insert does not make zone maps stale
- Single table ‘local’ zone maps
  - Update and insert marks impacted zones as stale (and any aggregated partition entry)
  - No impact on zone maps for delete
- Joined zone map
  - DML on fact table equivalent behavior to single table zone map
  - DML on dimension table makes dependent zone maps fully stale

Zone Maps

Refresh

- Incremental and full refresh, as required by DML
  - Zone map refresh does require a materialized view log
    - Only stale zones are scanned to refresh the MV
  - For joined zone map
    - DML on fact table: incremental refresh
    - DML on dimension table: full refresh
- Zone map maintenance through
  - DBMS_MVIEW.REFRESH()
  - ALTER MATERIALIZED ZONEMAP <xx> REBUILD;

ORACLE

Example – Dimension Hierarchies

ORDERS

id	product_id	location_id	amount
1	3	23	2.00
2	88	55	43.75
3	31	99	33.55
4	33	62	23.12
5	21	11	38.00
6	33	21	5.00
7	44	71	10.99

LOCATIONS

location_id	State	county
23	California	Inyo
102	New Mexico	Union
55	California	Kern
1	Ohio	Lake
62	California	Kings

Note: a zone typically contains many more rows than show here.
This is for illustrative purposes only.

```
CREATE TABLE orders ( ... )
CLUSTERING orders
JOIN locations ON (orders.location_id = locations.location_id)
BY INTERLEAVED ORDER (locations.state, locations.county)
WITH MATERIALIZED ZONEMAP …
```

ORACLE

Example – Dimension Hierarchies

ORDERS

id	product_id	location_id	amount
1	3	23	2.00
2	88	55	43.75
3	31	99	33.55
4	33	62	23.12
5	21	11	38.00
6	33	21	5.00
7	44	71	10.99

Scan Zone

LOCATIONS

location_id	State	county
23	California	Inyo
102	New Mexico	Union
55	California	Kern
1	Ohio	Lake
62	California	Kings

```
SELECT SUM(amount)
FROM orders
JOIN locations ON (orders.location.id = locations.location.id)
WHERE state = 'California';
```

Note: a zone typically contains many more rows than show here.
This is for illustrative purposes only.

Example – Dimension Hierarchies

ORDERS

id	product_id	location_id	amount
1	3	23	2.00
2	88	55	43.75
3	31	99	33.55
4	33	62	23.12
5	21	11	38.00
6	33	21	5.00
7	44	71	10.99

Scan Zone

LOCATIONS

location_id	State	county
23	California	Inyo
102	New Mexico	Union
55	California	Kern
1	Ohio	Lake
62	California	Kings

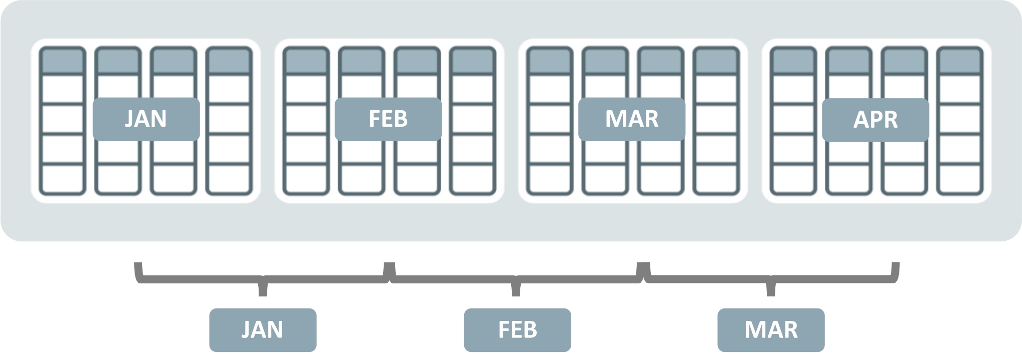
```
SELECT SUM(amount)
FROM orders
JOIN locations ON (orders.location.id = locations.location.id)
WHERE state = 'California'
AND county = 'Kern';
```

Note: a zone typically contains many more rows than show here.
This is for illustrative purposes only.

ORACLE

Zone Maps and Partitioning

Partition Key:
ORDER_DATE



Zone map:
SHIP_DATE

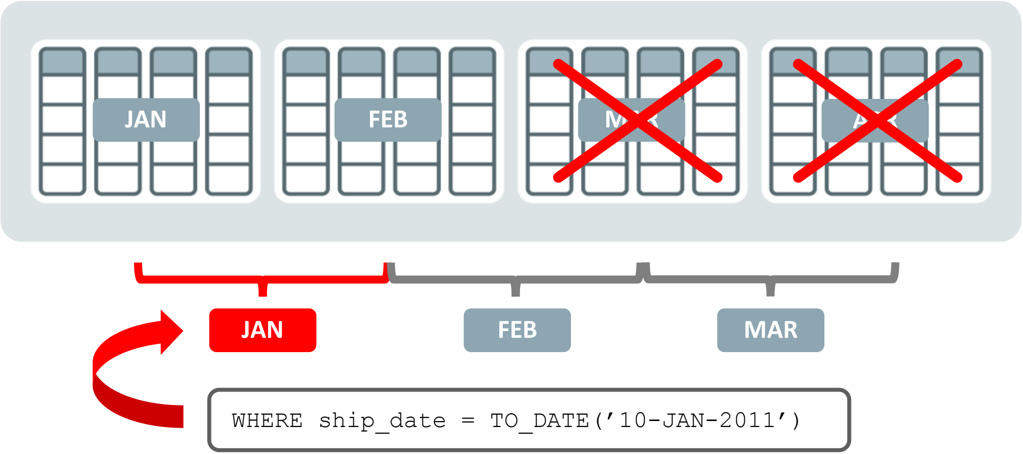
Zone map column SHIP_DATE correlates with partition key ORDER_DATE

- Zone maps can prune partitions for columns that are not included in the partition (or subpartition) key

ORACLE

Zone Maps and Partitioning

Partition Key:
ORDER_DATE



MAR and APR partitions are pruned

Zone map:
SHIP_DATE

- Zone maps can prune partitions for columns that are not included in the partition (or subpartition) key

Zone Maps and Storage Indexes

- Attribute clustering and zone maps work transparently with Exadata storage indexes
  - The benefits of Exadata storage indexes continue to be fully exploited
- In addition, zone maps (when used with attribute clustering)
  - Enable additional and significant IO optimization
    - Provide an alternative to indexes, especially on large tables
    - Join and fact-dimension queries, including dimension hierarchy searches
    - Particularly relevant in star and snowflake schemas
  - Are able to prune entire partitions and sub-partitions
  - Are effective for both direct and conventional path reads
  - Include optimizations for joins and index range scans
  - Part of the physical database design: explicitly created and controlled by the DBA

ORACLE