
CERN

# Advanced SQL

Marcin.Blaszczyk@cern.ch

Oracle Tutorials – May 2013

# Outline

- Advanced Queries
- Analytical Functions & Set Operators
- Indexes & IOTs
- Partitioning
- Undo & Flashback technologies

*Disclaimer: “this is not a SQL tuning tutorial”*

# Advanced queries

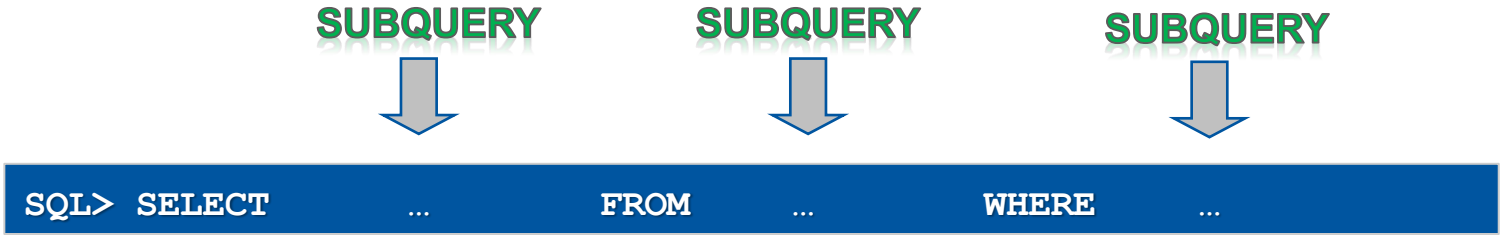
Correlated Subqueries & Nested Subqueries
Inline Views
Top-n queries
Hierarchical queries

# Advanced Queries

| Query Type | Question Answered |
|---|---|
| Correlated Subquery | Who are the employees that receive more than the average salary of their department? |
| Nested Subquery | Who works in the same department as King OR Smith? |
| Inline Views | What are the employees salary and the minimum salary in their department? |
| Top-N QUERIES | What are the 5 most well paid employees? |
| Hierarchical QUERIES | What is the hierarchy of management in my enterprise? |

# Subqueries

- A subquery is a query within a query and it is used to answer multiple-part questions.
  - A subquery can reside in the WHERE clause, the FROM clause or the SELECT clause.
- Main types
  - Correlated – when main query is executed first
  - Nested - when subquery is executed first
  - Inline View – acts as a data source



# Subquery Types

- Single-row (and single-column)

  ***Who works in the same department as King?***

```
SQL> SELECT … WHERE dep = (SELECT dep FROM … WHERE name = 'KING');
```

- Multiple-row (and single-column)

  ***Who works in the same department as King OR Smith?***

```
SQL> SELECT … WHERE dep IN (SELECT dep
                            FROM …
                            WHERE name ='KING' or name = 'SMITH');
```

- Multiple-column

  ***Who works in the same department(s) AND under the same boss as Smith?***

```
SQL> SELECT … WHERE (dep, mgr) = (SELECT dep, mgr
                                  FROM …
                                  WHERE name = 'SMITH');
```

# Correlated Subqueries

- A correlated subquery is a query that is evaluated for each row produced by the parent query.
- ***Which employees receive more than the average salary of their department?***

```
SELECT e.emp_id, e.dept_id,
       e.last_name, e.salary
FROM employees e
WHERE e.salary > (SELECT avg(i.salary)
                  FROM employees i
                  WHERE e.dept_id = i.dept_id);
```

| EMP_ID | DEPT_ID | LAST_NAME | SALARY |
|---|---|---|---|
| 201 | 20 | Hartstein | 13000 |
| 114 | 30 | Raphaely | 11000 |
| 123 | 50 | Vollman | 6500 |
| 122 | 50 | Kaufling | 7900 |
| 120 | 50 | Weiss | 8000 |
| 121 | 50 | Fripp | 8200 |
| 103 | 60 | Hunold | 9000 |
| 147 | 80 | Errazuriz | 12000 |
| 146 | 80 | Partners | 13500 |
| 145 | 80 | Russell | 14000 |
| 100 | 90 | King | 24000 |
| 108 | 100 | Greenberg | 12000 |

- *In this case, the correlated subquery specifically computes, for each employee, the average salary for the employee's department*

# Nested Subqueries

- A nested subquery is executed first and its results are inserted into WHERE clause of a the main query
- ***Who works in the same department as King or Smith?***

```
SELECT emp_id, dept_id, last_name, salary
FROM employees e
WHERE dept_id IN (SELECT dept_id
                  FROM employees
                  WHERE last_name ='King' or
                        last_name = 'Smith');
```

| EMP_ID | DEPT_ID | LAST_NAME | SAL |
|---|---|---|---|
| 145 | 80 | Russell | 14000 |
| 146 | 80 | Partners | 13500 |
| 147 | 80 | Errazuriz | 12000 |
| 150 | 80 | Tucker | 10000 |
| 162 | 80 | Vishney | 10500 |
| 164 | 80 | Marvins | 7200 |
| 170 | 80 | Fox | 9600 |
| 171 | 80 | Smith | 7400 |
| 173 | 80 | Kumar | 6100 |
| 100 | 90 | King | 24000 |
| 101 | 90 | Kochhar | 17000 |
| 102 | 90 | De Haan | 17000 |

- *In this case, the nested subquery returns department_id's for two employees and after that the parent query evaluates the condition*

# Inline Views

- An In-line view is a subquery in the FROM clause of a SQL statement just as if it was a table (acts as a data source)
- ***What are the employees salary and the MINIMAL salary in their department?***

```
SELECT e.emp_id a.dept_id, e.last_name,
          e.salary, a.min_sal,
FROM employees e,
    (SELECT MIN(salary)min_sal, dept_id
     FROM employees
     GROUP BY dept_id) a
WHERE e.dept_id = a.dept_id
ORDER BY e.dept_id, e.salary DESC;
```

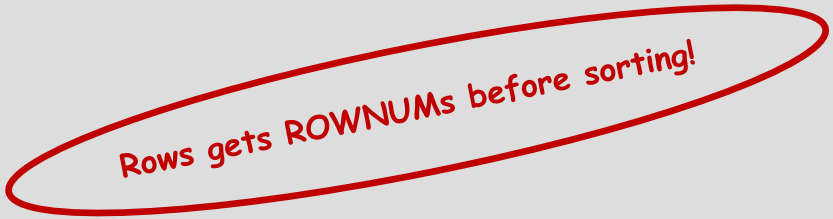
| EMP_ID | DEPT_ID | LAST_NAME | SALARY | MIN_SAL |
|---|---|---|---|---|
| 200 | 10 | Whalen | 4400 | 4400 |
| 201 | 20 | Hartstein | 13000 | 6000 |
| 202 | 20 | Fay | 6000 | 6000 |
| 114 | 30 | Raphaely | 11000 | 2500 |
| 115 | 30 | Khoo | 3100 | 2500 |
| 116 | 30 | Baida | 2900 | 2500 |
| 117 | 30 | Tobias | 2800 | 2500 |
| 118 | 30 | Himuro | 2600 | 2500 |
| 119 | 30 | Colmenares | 2500 | 2500 |
| 203 | 40 | Mavris | 6500 | 6500 |
| 121 | 50 | Fripp | 8200 | 2100 |
| 120 | 50 | Weiss | 8000 | 2100 |
| 122 | 50 | Kaufling | 7900 | 2100 |
| 123 | 50 | Vollman | 6500 | 2100 |
| 124 | 50 | Mourgos | 5800 | 2100 |
| 184 | 50 | Sarchand | 4200 | 2100 |
| 185 | 50 | Bull | 4100 | 2100 |
| 192 | 50 | Bell | 4000 | 2100 |

# TOP-N Queries

- use in-line view together with the ROWNUM pseudocolumn

```
SQL> SELECT str_val, rownum FROM test_rownum WHERE ROWNUM < 4;

STR_VAL  ROWNUM
------- -------
ccc            1
bbb            2
aaa            3
```

```
SQL> SELECT str_val, rownum FROM test_rownum WHERE ROWNUM < 4 order by 1;

STR_VAL  ROWNUM
------- -------
aaa            3
bbb            2
ccc            1
```



# TOP-N Queries

- use in-line view together with the ROWNUM pseudocolumn
- ***What are the top 5 most well paid employees?***

```
SELECT * FROM
          (SELECT emp_id, last_name, salary
           FROM employees
           ORDER BY salary desc)
WHERE rownum < 6;
```

| EMP_ID | LAST_NAME | SALARY |
|---|---|---|
| 100 | King | 24000 |
| 101 | Kochhar | 17000 |
| 102 | De Haan | 17000 |
| 145 | Russell | 14000 |
| 146 | Partners | 13500 |

- ***What are the next 5 most well paid employees?***

```
SELECT emp_id, last_name, salary
FROM (SELECT emp_id,last_name, salary,  rownum as rnum
      FROM employees
      ORDER BY salary desc)
WHERE rnum between 6 and 10;
```

| EMP_ID | LAST_NAME | SALARY |
|---|---|---|
| 108 | Greenberg | 12000 |
| 109 | Faviet | 9000 |
| 106 | Pataballa | 4800 |
| 105 | Austin | 4800 |
| 107 | Lorentz | 4200 |

# Hierarchical Queries

- If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause
- Syntax:

```
SELECT … FROM … WHERE …
       START WITH  <condition>
-- specifies the starting point of the hierarchy (tree)
       CONNECT BY PRIOR child_row = parent_row (TOP-DOWN)
                        parent_row = child_row (BOTTOM-UP)
-- relationship between parent row and child rows of the hierarchy
```

```
SQL> desc employees
 Name                 Null?    Type
 -------------------- -------- ------------
 EMP_ID                        NUMBER(5)
 LAST_NAME                     VARCHAR2(10)
 MGR_ID                        NUMBER(5)
```

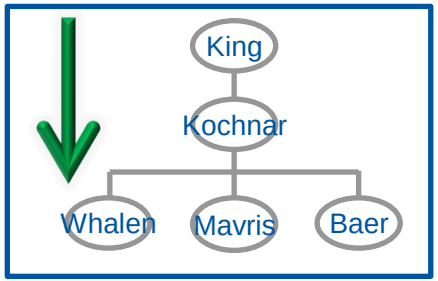
```
SQL> SELECT * FROM employees;

        EMP_ID LAST_NAME           MGR_ID
-------------- ---------- --------------
           204 Baer                  101
           203 Mavris                101
           204 King
           101 Kochhar               100
           200 Whalen                101
           204 Baer                  101
```

# Hierarchical Queries - Example

- Top down

```
SELECT emp_id, last_name, mgr_id,
LEVEL FROM employees
WHERE LEVEL <= 3
START WITH emp_id = 100
CONNECT BY PRIOR
emp_id = mgr_id;
```

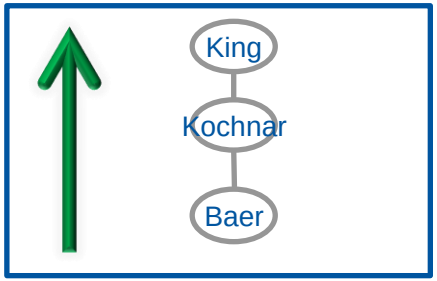
```
EMP_ID LAST_NAME MGR_ID LEVEL
------ --------- ------ -----
   100 King                1
   101 Kochhar       100   2
   200 Whalen        101   3
   203 Mavris        101   3
   204 Baer          101   3
```




- Bottom up

```
SELECT emp_id, last_name, mgr_id,
LEVEL FROM employees
START WITH emp_id = 204
CONNECT BY PRIOR
mgr_id = emp_id;
```

```
EMP_ID LAST_NAM MGR_ID  LEVEL
------ -------- ------ ------
   204 Baer        101      1
   101 Kochhar     100      2
   100 King                 3
```



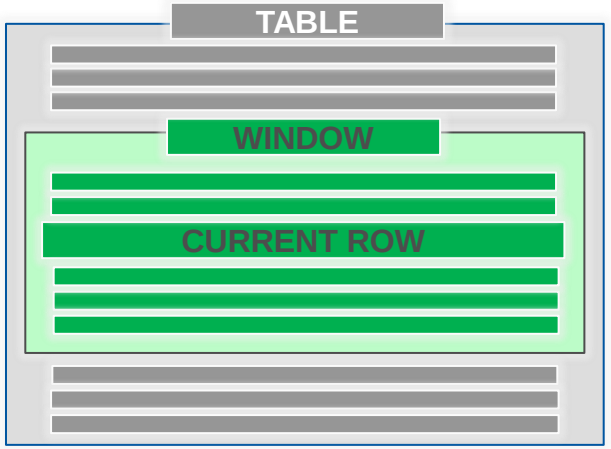
# Analytical Functions & Set operators

Ordered Analytical Window & Range Specification
Partitioned Analytical Window
Analytical Function Summary & Example
Set Operators

# Analytical Functions Overview

- General syntax of analytical function:

```
SELECT analytical-function(col-expr)
       OVER (window-spec) [AS col-alias]
FROM [TABLE];
```




- Window specification syntax

```
[PARTITION BY [expr list]]
ORDER BY [sort spec] [range spec]
```

- Example for range specification (for more check oracle docs)

```
• ROWS UNBOUNDED PRECEDING AND CURRENT ROW (default)
• ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
• RANGE BETWEEN 2 PRECEDING AND 2 FOLLOWING
```

# Ordered Analytical Window

- Analytical functions applied to all window rows

```
SQL>  select employee_id, last_name, manager_id, salary
      sum(salary) over (order by  employee_id, last_name, salary)
      as cumulative from employees;

EMPLOYEE_ID LAST_NAME MANAGER_ID SALARY CUMULATIVE
----------- --------- ---------- ------ ----------
        100 King                24000      24000
        101 Kochhar          100  17000      41000
        102 De Haan          100  17000      58000 = 24000 + 17000 + 17000
        103 Hunold           102   9000      67000
        104 Ernst            103   6000      73000
        105 Austin           103   4800      77800
        106 Pataballa        103   4800      82600
        107 Lorentz          103   4200      86800
        108 Greenberg        101  12000      98800
        109 Faviet           108   9000     107800
        110 Chen             108   8200     116000
```

# Range Specification – Example (1)

**RANGE BETWEEN 2 PRECEDING AND 2 FOLLOWING**

```
SQL> select manager_id, last_name, salary, sum(salary) over (order by
          last_name, salary rows between 2 preceding and 1 following) as
          cumulative from employees;

MANAGER_ID LAST_NAME SALARY CUMULATIVE
---------- --------- ------ ----------
       103 Austin      4800      10800
       103 Ernst       6000      22800
       101 Greenberg  12000      31800
       102 Hunold      9000      51000 = 6000 + 12000 + 9000 + 24000
           King       24000      62000
       100 Kochhar    17000      54200
       103 Lorentz     4200      45200
```

# Range Specification – Example (2)

**ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING**

```
SQL> select manager_id, last_name, salary, sum(salary) over (order by
          last_name, salary rows between current row and unbounded
          following) as cumulative from emp_part;

MANAGER_ID LAST_NAME  SALARY CUMULATIVE
---------- --------- ------ ----------
       103 Austin      4800      77000
       103 Ernst       6000      72200
       101 Greenberg  12000      66200
       102 Hunold      9000      54200 = 9000 + 24000 + 17000 + 4200
           King       24000      45200
       100 Kochhar    17000      21200
       103 Lorentz     4200       4200
```

# Partitioned Analytical Window

- Analytical functions start again for each partition

```
SQL> break on manager_id
SQL> SELECT manager_id, last_name, employee_id, salary,
     sum(salary) over (PARTITION BY manager_id order by employee_id) as cumulative
     FROM employees order by manager_id, employee_id, last_name;

MANAGER_ID LAST_NAME       EMPLOYEE_ID SALARY CUMULATIVE
---------- --------------- ----------- ------ ----------
       100 Kochhar                 101  17000      17000
           De Haan                 102  17000      34000
           Raphaely                114  11000      45000
           Weiss                   120   8000      53000
       101 Greenberg               108  12000      12000
           Whalen                  200   4400      16400
           Mavris                  203   6500      22900
           Baer                    204  10000      32900
       102 Hunold                  103   9000       9000
       103 Ernst                   104   6000       6000
           Austin                  105   4800      10800
           Pataballa               106   4800      15600
```

# Partitioned Analytical Window

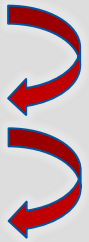
- Analytical functions start again for each partition

```
SQL> break on manager_id
SQL> SELECT manager_id, last_name, employee_id, salary,
     sum(salary) over (PARTITION BY manager_id order by employee_id) as cumulative
     FROM employees order by manager_id, employee_id, last_name;

MANAGER_ID LAST_NAME       EMPLOYEE_ID SALARY CUMULATIVE
---------- --------------- ----------- ------ ----------
       100 Kochhar                 101  17000      17000
           De Haan                 102  17000      34000
           Raphaely                114  11000      45000
           Weiss                   120   8000      53000
       101 Greenberg               108  12000      12000
           Whalen                  200   4400      16400
           Mavris                  203   6500      22900
           Baer                    204  10000      32900
       102 Hunold                  103   9000       9000
       103 Ernst                   104   6000       6000
           Austin                  105   4800      10800
           Pataballa               106   4800      15600
```

# Partitioned Analytical Window

- Analytical functions start again for each partition

```
SQL> break on manager_id
SQL> SELECT manager_id, last_name, employee_id, salary,
     sum(salary) over (PARTITION BY manager_id order by employee_id) as cumulative
     FROM employees order by manager_id, employee_id, last_name;

MANAGER_ID LAST_NAME       EMPLOYEE_ID SALARY CUMULATIVE
---------- --------------- ----------- ------ ----------
       100 Kochhar                 101  17000      17000
           De Haan                 102  17000      34000
           Raphaely                114  11000      45000
           Weiss                   120   8000      53000
       101 Greenberg               108  12000      12000
           Whalen                  200   4400      16400
           Mavris                  203   6500      22900
           Baer                    204  10000      32900
       102 Hunold                  103   9000       9000
       103 Ernst                   104   6000       6000
           Austin                  105   4800      10800
           Pataballa               106   4800      15600
```



# Partitioned Analytical Window

- Analytical functions start again for each partition

```
SQL> break on manager_id
SQL> SELECT manager_id, last_name, employee_id, salary,
     sum(salary) over (PARTITION BY manager_id order by employee_id) as cumulative
     FROM employees order by manager_id, employee_id, last_name;

MANAGER_ID LAST_NAME       EMPLOYEE_ID SALARY CUMULATIVE
---------- --------------- ----------- ------ ----------
       100 Kochhar                 101  17000      17000
           De Haan                 102  17000      34000
           Raphaely                114  11000      45000
           Weiss                   120   8000      53000
       101 Greenberg               108  12000      12000
           Whalen                  200   4400      16400
           Mavris                  203   6500      22900
           Baer                    204  10000      32900
       102 Hunold                  103   9000       9000
       103 Ernst                   104   6000       6000
           Austin                  105   4800      10800
           Pataballa               106   4800      15600
```

# Partitioned Analytical Window

- Analytical functions start again for each partition

```
SQL> break on manager_id
SQL> SELECT manager_id, last_name, employee_id, salary,
     sum(salary) over (PARTITION BY manager_id order by employee_id) as cumulative
     FROM employees order by manager_id, employee_id, last_name;

MANAGER_ID LAST_NAME       EMPLOYEE_ID SALARY CUMULATIVE
---------- --------------- ----------- ------ ----------
       100 Kochhar                 101  17000      17000
           De Haan                 102  17000      34000
           Raphaely                114  11000      45000
           Weiss                   120   8000      53000
       101 Greenberg               108  12000      12000
           Whalen                  200   4400      16400
           Mavris                  203   6500      22900
           Baer                    204  10000      32900
       102 Hunold                  103   9000       9000
       103 Ernst                   104   6000       6000
           Austin                  105   4800      10800
           Pataballa               106   4800      15600
```

= 6000 + 4800 + 4800

# Analytical Functions

- For analytic functions, you can use all of the regular group functions:
  - SUM
  - MAX
  - MIN
  - AVG
  - COUNT
- Plus list of additional analytical functions that can be used only for window queries:
  - LAG
  - LEAD
  - FIRST
  - LAST
  - FIRST VALUE
  - LAST VALUE
  - ROW_NUMBER
  - DENSE_RANK

# Analytical Function - Example

- LAG function example

```
SQL> select * from currency order by 1;

DAY                   EURCHF
-------------------- ------
01-JUN-2012 00:00:00  1.240
02-JUN-2012 00:00:00  1.223
03-JUN-2012 00:00:00  1.228
04-JUN-2012 00:00:00  1.217
05-JUN-2012 00:00:00  1.255
06-JUN-2012 00:00:00  1.289
07-JUN-2012 00:00:00  1.291
08-JUN-2012 00:00:00  1.247
09-JUN-2012 00:00:00  1.217
10-JUN-2012 00:00:00  1.265
```

```
SQL> select day, EURCHF, lag(EURCHF,1) over
(order by day) as prev_eurchf from currency;

DAY                   EURCHF    PREV_EURCHF
-------------------- ------ --------------
01-JUN-2012 00:00:00  1.240
02-JUN-2012 00:00:00  1.223          1.240
03-JUN-2012 00:00:00  1.228          1.223
04-JUN-2012 00:00:00  1.217          1.228
05-JUN-2012 00:00:00  1.255          1.217
06-JUN-2012 00:00:00  1.289          1.255
07-JUN-2012 00:00:00  1.291          1.289
08-JUN-2012 00:00:00  1.247          1.291
09-JUN-2012 00:00:00  1.217          1.247
10-JUN-2012 00:00:00  1.265          1.217
```

# Set Operators

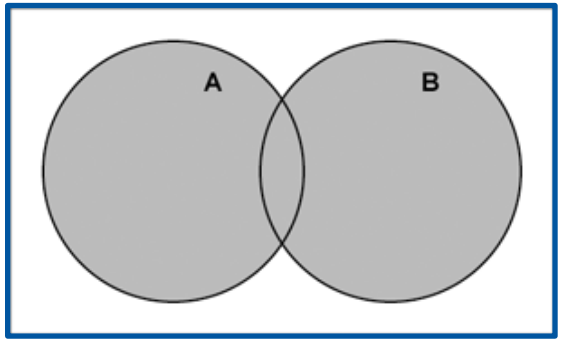
- Combine multiple queries
  - Union without duplicates and with the duplicates

```
SELECT name, email FROM employees
UNION
SELECT name, email FROM visitors;
```

```
SELECT cit_id FROM employees
UNION ALL
SELECT cit_id FROM visitors;
```

  - Intersect

```
SELECT  name FROM employees
INTERSECT
SELECT  name FROM visitors;
```




  - Minus

```
SELECT  name FROM employees
MINUS
SELECT  name FROM visitors;
```

# Indexes & IOTs

Indexes Overview
B-Tree Index
Bitmap Index
Composite & Function Based Index
Index Organized Tables

# Why Indexing?

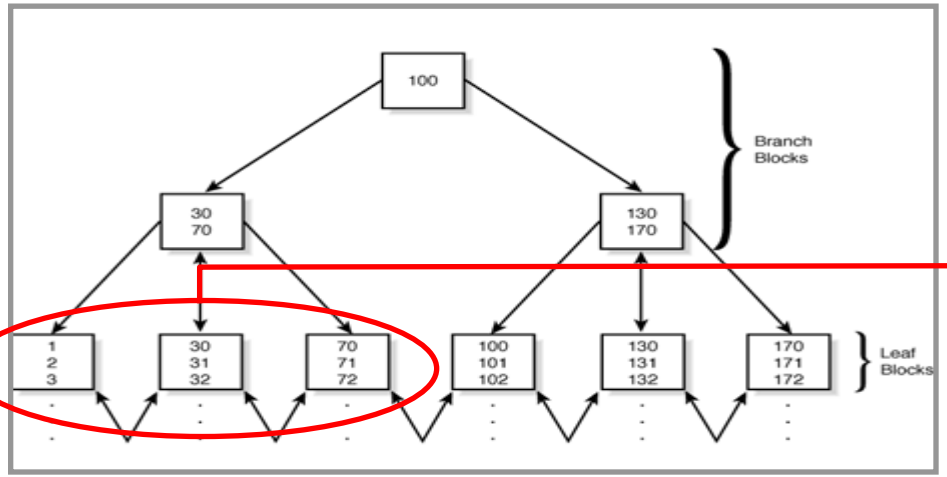
- Index creates an entry for each value that appears in the indexed columns
  - B-Tree index (default)
  - Bitmap index
- Syntax:

```
SQL> CREATE [BITMAP] INDEX index_name  ON table_name (column1,... column_n);
```

- Indexes
  - Allow faster retrieval of record
  - Have maintenance overhead

# B-Tree Index

- Index with a balanced tree
- When to use?
  - OLTP systems
  - High cardinality columns (primary key columns)
- Size: B-tree index will be significantly smaller than Bitmap index for high cardinality column.

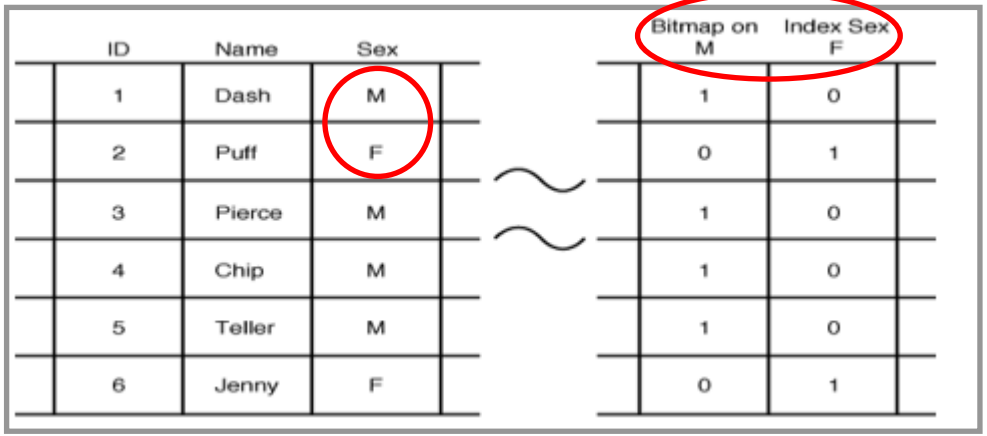



```
SQL> CREATE INDEX
i_employee_id ON employee
(empid);
```

```
SELECT *
FROM employee
WHERE empid < 7
```

# Bitmap Index

- Index with a bitmap of the column values
- When to use?
  - DSS systems (bitmap indexes can cause a serious locking problem in systems where data is frequently updated by many concurrent systems)
  - Low cardinality columns (columns with few discrete values)
- Size: Bitmap index will be significantly smaller than B-tree index on low cardinality column



| ID | Name | Sex |
|---|---|---|
| 1 | Dash | M |
| 2 | Puff | F |
| 3 | Pierce | M |
| 4 | Chip | M |
| 5 | Teller | M |
| 6 | Jenny | F |

| Bitmap on M | Index Sex F |
|---|---|
| 1 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 0 |
| 1 | 0 |
| 0 | 1 |

```
SQL> CREATE BITMAP INDEX
i_employee_sex ON employee
(sex);
```

```
SELECT * FROM employee
WHERE sex='F';
```

# Composite & Function Based Index

- Composite index: Index over multiple columns in a table
- When to use?
  - When WHERE clause uses more than one column
  - To increase selectivity joining columns of low selectivity

```
SQL> CREATE INDEX mgr_deptno_idx ON emp(mgr, deptno);
```

- Function-based index: Is an index created on a function that involves columns in the table being indexed (b-tree or bitmap)
  - They speed up queries that evaluate those functions to select data

```
SQL> CREATE INDEX emp_name_idx ON employee (UPPER(ename));
```

# Index Organized Tables

- IOT stores all of the table's data in the B-tree index structure

```
CREATE TABLE orders (
          order_id NUMBER(10),
          (…)
          CONSTRAINT pk_orders PRIMARY KEY (order_id)
          )
ORGANIZATION INDEX;
```

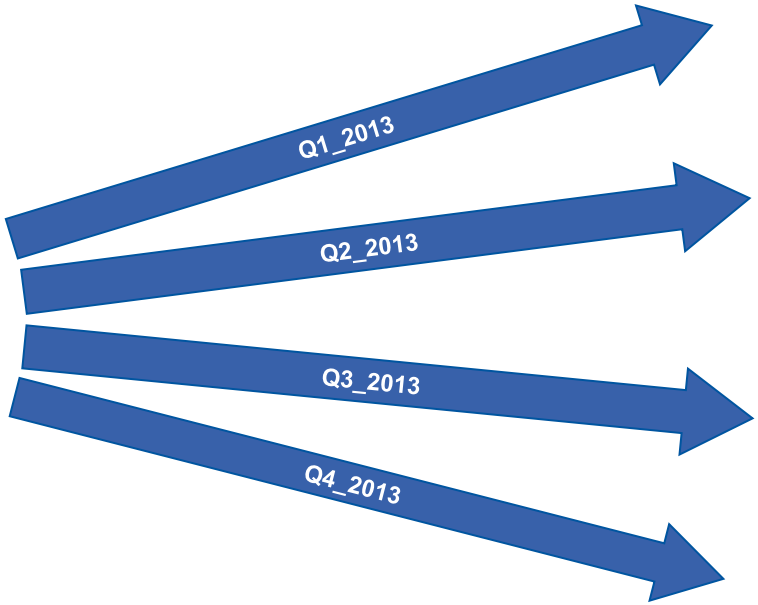
- Efficient when:
  - table is usually accessed by the primary key
- Inefficient when:
  - there's a heavy DML activity especially not primary key based
  - access to table's data not via primary key is slower comparing to a cheap table

# Partitioning in Oracle

Partitioning Overview
Partitioning Types in Oracle
Partition Pruning
Partition Wise Joins
Partitioned Indexes

# Partitioning Overview

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 71 | 25 | 15-JAN-2013 |
| 102 | 365 | 28-FEB-2013 |
| 111 | 245 | 21-MAR-2013 |
| 225 | 312 | 01-APR-2013 |
| 317 | 425 | 17-MAY-2013 |
| 441 | 5 | 27-JUN-2013 |
| 492 | 315 | 21-JUL-2013 |
| 511 | 55 | 11-AUG-2013 |
| 573 | 221 | 09-SEP-2013 |
| 623 | 710 | 16-OCT-2013 |
| 699 | 115 | 29-NOV-2013 |
| 726 | 36 | 30-DEC-2013 |




| TX_ID | ITEM_ID | TS |
|---|---|---|
| 71 | 25 | 15-JAN-2013 |
| 102 | 365 | 28-FEB-2013 |
| 111 | 245 | 21-MAR-2013 |

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 225 | 312 | 01-APR-2013 |
| 317 | 425 | 17-MAY-2013 |
| 441 | 5 | 27-JUN-2013 |

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 623 | 710 | 16-OCT-2013 |
| 699 | 115 | 29-NOV-2013 |
| 726 | 36 | 30-DEC-2013 |

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 492 | 315 | 21-JUL-2013 |
| 511 | 55 | 11-AUG-2013 |
| 573 | 221 | 09-SEP-2013 |

# Partitioning Overview (2)

- Tables and indexes can be divided into smaller and more manageable physical pieces called partitions which are treated as a single logical unit
- Advantages:
  - Manageability: data management operations at the partition level (data load, index creation, backup/recovery, etc)
  - Performance: Improves query performance, possibility of concurrent maintenance operations on different partitions of the same table/index.
- Partitioning can be implemented without requiring any modifications to your applications.

# Partitioning Types in Oracle

- There are different criteria to split the data:
  - **List**: partition by lists of predefined discrete values
  - **Range**: partition by predefined ranges of continuous values
  - **Hash**: partition according to hashing algorithm applied by Oracle
  - **Interval**: partition by predefined interval – partitions created automatically (11g)
  - **Reference**: child table inherits the partitioning type from a parent table (11g)
  - **System**: no partition keys, application control partition selection (11g)
  - **Composite**: e.g. range-partition by key1, hash-subpartition by key2
    - Some new combinations in 11g

# Partitioning Types – Example (1)

- Range: partition by predefined ranges of continuous values

```
CREATE TABLE SALES_2013
(
    tx_id  NUMBER(5),
    item_id NUMBER(5),
    ts DATE
)
PARTITION BY RANGE(ts) (
PARTITION q1_2013 VALUES LESS THAN(TO_DATE('01/04/2013','DD/MM/YYYY')),
PARTITION q2_2013 VALUES LESS THAN(TO_DATE('01/07/2013','DD/MM/YYYY')),
PARTITION q3_2013 VALUES LESS THAN(TO_DATE('01/10/2013','DD/MM/YYYY')),
PARTITION q4_2013 VALUES LESS THAN(TO_DATE('01/01/2014','DD/MM/YYYY'))
);
```

# Partitioning Types – Example (2)

- Composite: e.g. range-partition by key1, hash-subpartition by key2

```
CREATE TABLE SALES_REGIONS_2013
(    tx_id  NUMBER(5),
    item_id number(5),
    ts DATE,
    region VARCHAR2(1)
)
PARTITION BY RANGE(ts)
SUBPARTITION BY LIST(region) SUBPARTITION TEMPLATE (
SUBPARTITION p_emea VALUES ('E'),
SUBPARTITION p_asia VALUES ('A'),
SUBPARTITION p_nala VALUES ('N'))
(
PARTITION q1_2013 VALUES LESS THAN(TO_DATE('01/04/2013','DD/MM/YYYY')),
PARTITION q2_2013 VALUES LESS THAN(TO_DATE('01/07/2013','DD/MM/YYYY')),
PARTITION q3_2013 VALUES LESS THAN(TO_DATE('01/10/2013','DD/MM/YYYY')),
PARTITION q4_2013 VALUES LESS THAN(TO_DATE('01/01/2014','DD/MM/YYYY'))
);
```

# Partitioning Types – Example (3)

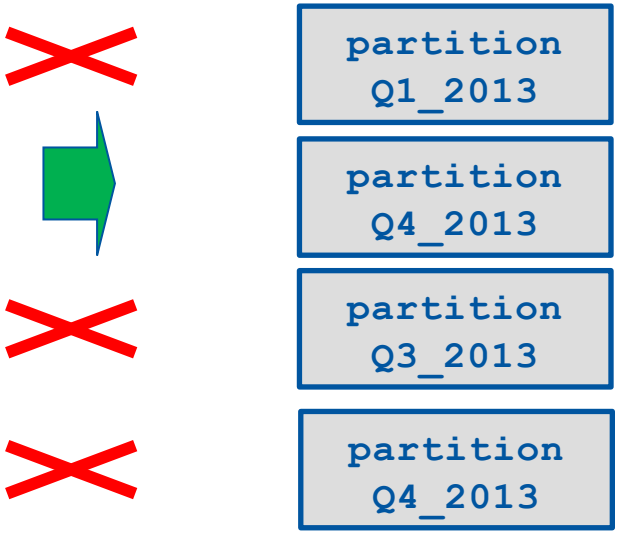
- Interval : partition by predefined interval – partitions created automatically

```
CREATE TABLE SALES_2013
(
    tx_id  NUMBER(5),
    item_id NUMBER(5),
    ts DATE
)
PARTITION BY RANGE (ts) INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))  (
PARTITION jan2013 VALUES LESS THAN(TO_DATE('01/01/2013','DD/MM/YYYY')),
PARTITION feb2013 VALUES LESS THAN(TO_DATE('01/02/2013','DD/MM/YYYY'))
);
```

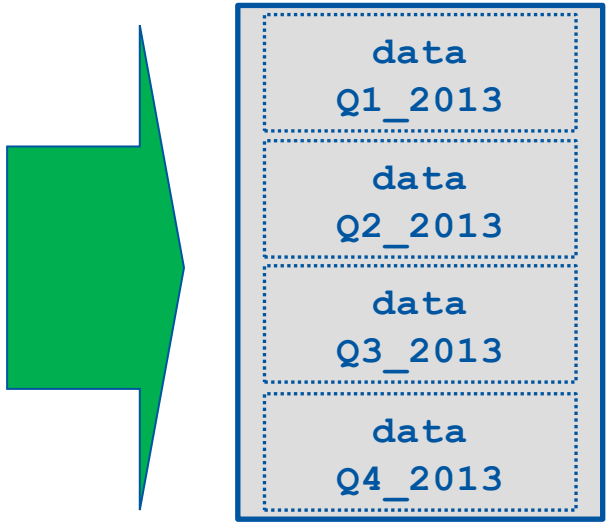
# Partition Pruning (1)

```
SQL> insert into sales values(726,36,TO_DATE('30/05/2013','DD/MM/YYYY'));
```

With partitioning



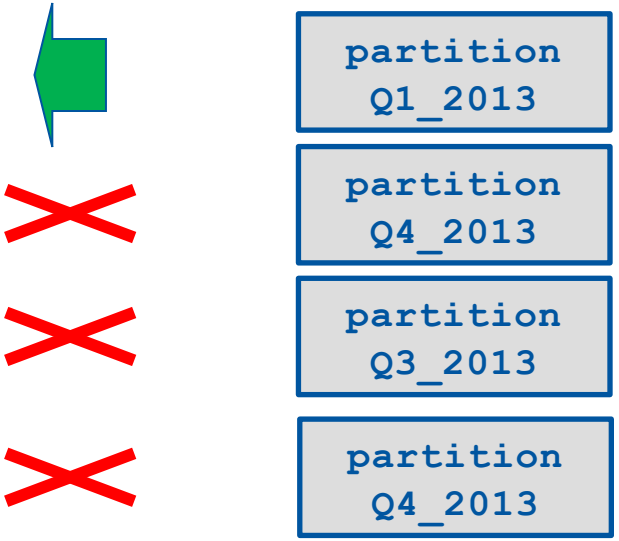

Without partitioning



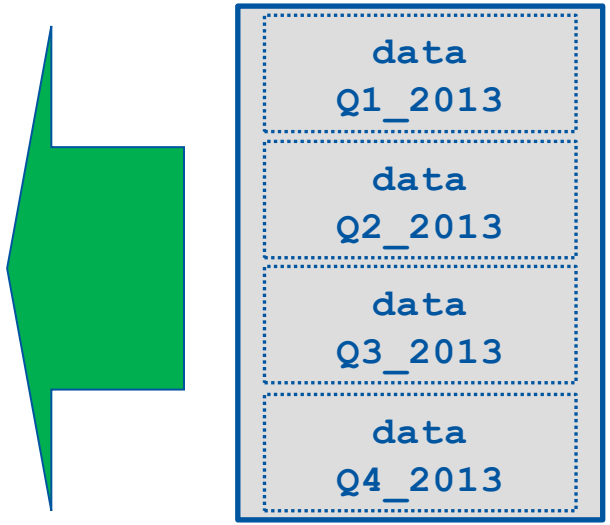
# Partition Pruning (2)

```
SQL> SELECT * FROM sales WHERE ts < TO_DATE('01/04/2013','DD/MM/YYYY');
```

With partitioning

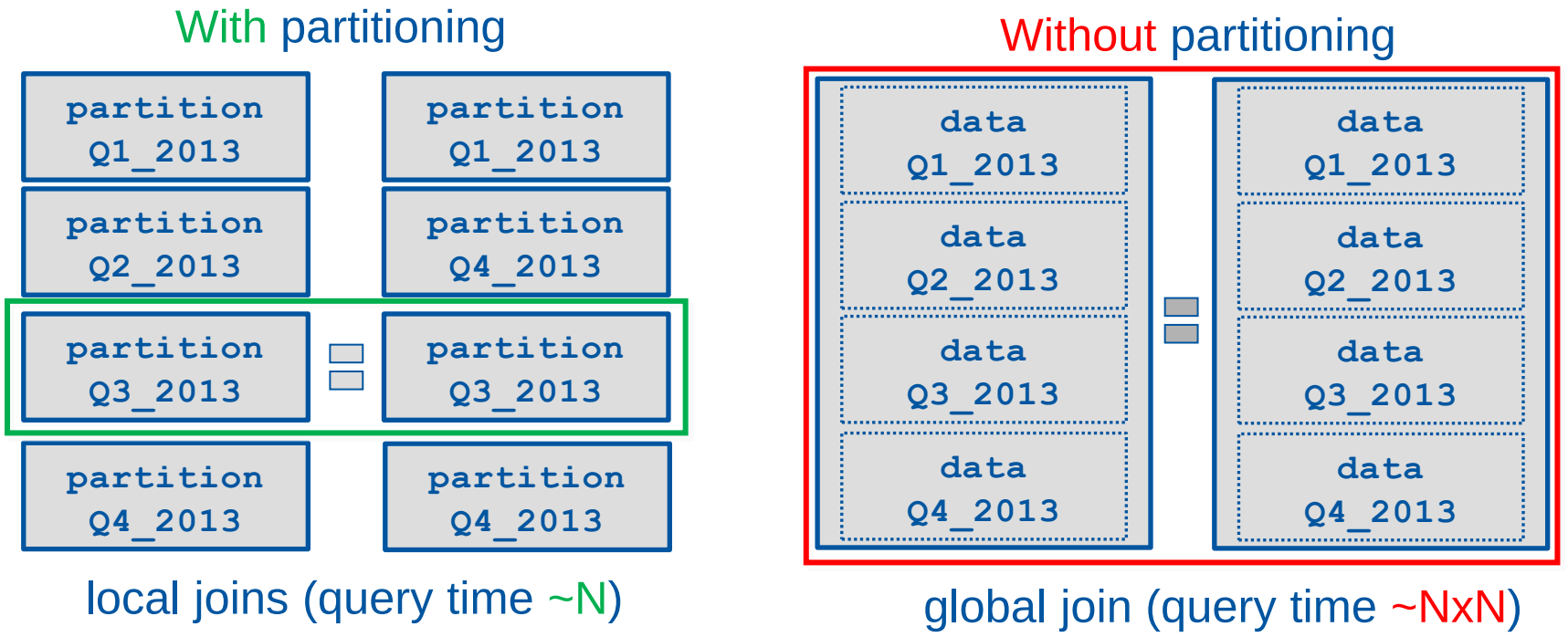



Without partitioning



# Partition Wise Joins

```
SQL> SELECT … FROM table1,table2 WHERE table1.key = table2.key;
```



# Partitioned Indexes

- Local index: partitioned on the same key as table

```
SQL> CREATE INDEX day_idx ON table (day) LOCAL;
```

- Global index: not partitioned on the same key as table

```
SQL> CREATE INDEX day_idx ON table (day) GLOBAL;
```

- Combine the advantages of partitioning and indexing:
  - Partitioning improves query performance by pruning
  - Local index improves performance on full scan of partition
- Bitmap indexes on partitioned tables are always local
  - The concept of global index only applies to B*-tree indexes

# Undo & Flashback technologies

Undo Tablespace & Rollback Segments
Flashback Technologies Overview
Flashback Query
Flashback Version Query
Flashback Table
Flashback Drop

# Undo Tablespace - Overview

- Special tablespace for changed data called rollback segments
- Contains committed and uncommitted data
- Managed automatically
- Committed data can be overwritten (no guarantee)

UNDO TABLESPACE

# Undo Tablespc & Rollback Segments

```
SQL> UPDATE sales SET item_id=1000 where tx_id=441;
```

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 71 | 25 | 15-JAN-2013 |
| 102 | 365 | 28-FEB-2013 |
| 111 | 245 | 21-MAR-2013 |
| 225 | 312 | 01-APR-2013 |
| 317 | 425 | 17-MAY-2013 |
| 441 | 1000 | 27-JUN-2013 |
| 492 | 315 | 21-JUL-2013 |
| 511 | 55 | 11-AUG-2013 |
| 573 | 221 | 09-SEP-2013 |
| 623 | 710 | 16-OCT-2013 |
| 699 | 115 | 29-NOV-2013 |
| 726 | 36 | 30-DEC-2013 |

# Undo Tablespc & Rollback Segments

```
SQL> ROLLBACK;
```

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 71 | 25 | 15-JAN-2013 |
| 102 | 365 | 28-FEB-2013 |
| 111 | 245 | 21-MAR-2013 |
| 225 | 312 | 01-APR-2013 |
| 317 | 425 | 17-MAY-2013 |
| 441 | 5 | 27-JUN-2013 |
| 492 | 315 | 21-JUL-2013 |
| 511 | 55 | 11-AUG-2013 |
| 573 | 221 | 09-SEP-2013 |
| 623 | 710 | 16-OCT-2013 |
| 699 | 115 | 29-NOV-2013 |
| 726 | 36 | 30-DEC-2013 |


UNDO TABLESPACE

# Undo Tablespc & Rollback Segments

```
SQL> COMMIT;
```

| TX_ID | ITEM_ID | TS |
|---|---|---|
| 71 | 25 | 15-JAN-2013 |
| 102 | 365 | 28-FEB-2013 |
| 111 | 245 | 21-MAR-2013 |
| 225 | 312 | 01-APR-2013 |
| 317 | 425 | 17-MAY-2013 |
| 441 | 1000 | 27-JUN-2013 |
| 492 | 315 | 21-JUL-2013 |
| 511 | 55 | 11-AUG-2013 |
| 573 | 221 | 09-SEP-2013 |
| 623 | 710 | 16-OCT-2013 |
| 699 | 115 | 29-NOV-2013 |
| 726 | 36 | 30-DEC-2013 |

```
TX_ID:441; ITEM_ID:5; TS:15-JAN-2013
COMMITED;
```

UNDO TABLESPACE

# Flashback Technologies - Overview

- For COMMITED data
- Flashback technologies support recovery at all levels:
  - Row
  - Table
  - Transaction (this is not in the scope of this tutorial)
  - Entire Database (this is not in the scope of this tutorial)
- We DO NOT GUARANTEE that past data will be always accessible (UNDO is a circular buffer)
- SCN System Change Number - is an ever-increasing value that uniquely identifies a committed version of the database. In simple words: *“it’s an Oracle's clock - every time we commit, the clock increments.”* – Tom Kyte

# Flashback Technologies

- For error analysis
  - Flashback Query
  - Flashback Version Query
  - Flashback Transaction Query (not part of this tutorial)
- For error recovery
  - Flashback Transaction Backout (not part of this tutorial)
  - Flashback Table
  - Flashback Drop
  - Flashback Database (not part of this tutorial)

# Flashback Query

- to perform queries of a certain time

```
SQL> SELECT * FROM <TABLE> AS OF [ TIMESTAMP | SCN ];
```

```
SQL> select DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER from
dual;

GET_SYSTEM_CHANGE_NUMBER
------------------------
           6268302650456

SQL> delete from test;

3 rows deleted.

SQL> commit;

Commit complete.
```

```
SQL> SELECT * FROM test;

no rows selected

SQL> SELECT * FROM test
     AS OF SCN 6268302650456;

            ID STR_VAL
-------------- ----------
             1 one
             2 two
             3 three
```

# Flashback Version Query

- To retrieve all the versions of the rows that exist between two points in time (SCNs)
- Pseudocolumns:
  - VERSIONS_STARTTIME (start timestamp of version)
  - VERSIONS_ENDTIME (end timestamp of version)
  - VERSIONS_STARTSCN (start SCN of version)
  - VERSIONS_ENDSCN (end SCN of version)
  - VERSIONS_XID (transaction ID of version)
  - VERSIONS_OPERATION (DML operation of version)
- The VERSIONS clause cannot span DDL commands

```
SQL> SELECT versions_xid, versions_operation, salary
     FROM employees
     VERSIONS BEETWEN TIMESTAMP | SCN <t1> and <t2>;
```

# Flashback Table

- For error correction
- Flashback Table provides a way for users to easily and quickly recover from accidental modifications without a DBA's involvement☺

```
FLASHBACK TABLE employees TO [ TIMESTAMP | SCN <t1> ];
```

```
SQL> SELECT * FROM test;

no rows selected

SQL> ALTER TABLE test ENABLE ROW MOVEMENT;

Table altered.

SQL> FLASHBACK TABLE test TO SCN 6268302650456;

Flashback complete.
```

```
SQL> SELECT * FROM test

             ID STR_VAL
--------------- ----------
              1 one
              2 two
              3 three
```

# Flashback Drop

- For error correction
- The RECYCLEBIN initialization parameter is used to control whether the Flashback Drop capability is turned ON or OFF.
- It's RECYCLEBIN is set to ON for CERN Physics databases

```
SQL> FLASHBACK TABLE <table> TO BEFORE DROP;
```

```
SQL> DROP TABLE test;

Table dropped.

SQL> FLASHBACK TABLE test TO BEFORE DROP;

Flashback complete.
```

# Flashback – Example (0)

```
SQL> select versions_xid, versions_operation, versions_startscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                   1 one
 21001D00F8B50F00 I     6268303135869   6268303136686   3 three
 21001D00F8B50F00 I     6268303135869   6268303136686   2 two
 23000600BAFB0D00 U     6268303136686                   9 nine
 23000600BAFB0D00 D     6268303136686                   3 three
 23000400B9FC0D00 I     6268303136698                  11 eleven
 23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> desc TEST

 Name                 Null?    Type
 -------------------- -------- ------------
 ID                            NUMBER(5)
 STR_VAL                       VARCHAR2(10)
```

# Flashback – Example (1)

```
SQL> select versions_xid, versions_operation, versions_startscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                   1 one
 21001D00F8B50F00 I     6268303135869   6268303136686   3 three
 21001D00F8B50F00 I     6268303135869   6268303136686   2 two
 23000600BAFB0D00 U     6268303136686                   9 nine
 23000600BAFB0D00 D     6268303136686                   3 three
 23000400B9FC0D00 I     6268303136698                  11 eleven
 23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> select * from test;
(as of scn 6268303136698)

 ID STR_VAL
--- ----------
  1 one
  9 nine
 10 ten
 11 eleven
```

# Flashback – Example (2)

```
SQL> select versions_xid, versions_operation, versionsstartscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
---------------- - ----------------- --------------- --- ----------
21001D00F8B50F00 I     6268303135869                   1 one
21001D00F8B50F00 I     6268303135869   6268303136686   3 three
21001D00F8B50F00 I     6268303135869   6268303136686   2 two
23000600BAFB0D00 U     6268303136686                   9 nine
23000600BAFB0D00 D     6268303136686                   3 three
23000400B9FC0D00 I     6268303136698                  11 eleven
23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> select * from test;
(as of scn 6268303136698)

 ID STR_VAL
--- ----------
  1 one
  9 nine
 10 ten
 11 eleven
```

```
SQL> select * from test
     as of scn 6268303136686;

 ID STR_VAL
--- ----------
  1 one
  9 nine
```

# Flashback – Example (3)

```
SQL> select versions_xid, versions_operation, versions_startscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                  1 one
 21001D00F8B50F00 I     6268303135869   6268303136686  3 three
 21001D00F8B50F00 I     6268303135869   6268303136686  2 two
 23000600BAFB0D00 U     6268303136686                  9 nine
 23000600BAFB0D00 D     6268303136686                  3 three
 23000400B9FC0D00 I     6268303136698                 11 eleven
 23000400B9FC0D00 I     6268303136698                 10 ten
```

```
SQL> select * from test;
(as of scn 6268303136698)

 ID STR_VAL
--- ----------
  1 one
  9 nine
 10 ten
 11 eleven
```

```
SQL> select * from test
     as of scn 6268303136686;

 ID STR_VAL
--- ----------
  1 one
  9 nine
```

```
select * from test
as of scn 6268303135869;

 ID STR_VAL
--- ----------
  1 one
  2 two
  3 three
```

# Flashback – example (4)

```
SQL> select versions_xid, versions_operation, versions_startscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                   1 one
 21001D00F8B50F00 I     6268303135869   6268303136686   3 three
 21001D00F8B50F00 I     6268303135869   6268303136686   2 two
 23000600BAFB0D00 U     6268303136686                   9 nine
 23000600BAFB0D00 D     6268303136686                   3 three
 23000400B9FC0D00 I     6268303136698                  11 eleven
 23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> CREATE TABLE test (id NUMBER(5), str_val VARCHAR2(10));

SQL> INSERT INTO test VALUES(1, 'one');
SQL> INSERT INTO test VALUES(2, 'two');
SQL> INSERT INTO test VALUES(3, 'three');
SQL> COMMIT;
```

```
select * from test
as of scn 6268303135869;

 ID STR_VAL
--- ----------
  1 one
  2 two
  3 three
```

# Flashback – example (5)

```
SQL> select versions_xid, versions_operation, versionsstartscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                   1 one
 21001D00F8B50F00 I     6268303135869   6268303136686   3 three
 21001D00F8B50F00 I     6268303135869   6268303136686   2 two
 23000600BAFB0D00 U     6268303136686                   9 nine
 23000600BAFB0D00 D     6268303136686                   3 three
 23000400B9FC0D00 I     6268303136698                  11 eleven
 23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> UPDATE test SET id = 9, str_val = 'nine' WHERE id =2;
SQL> DELETE FROM test WHERE id = 3;
SQL> COMMIT;
```

```
SQL> select * from test
     as of scn 6268303136686;

 ID STR_VAL
--- ----------
  1 one
  9 nine
```

# Flashback – Example (6)

```
SQL> select versions_xid, versions_operation, versions_startscn, versions_endscn, id, str_val
     from test versions between timestamp minvalue and maxvalue order by VERSIONS_STARTSCN;

 VERSIONS_XID     V VERSIONS_STARTSCN VERSIONS_ENDSCN  ID STR_VAL
 ---------------- - ----------------- --------------- --- ----------
 21001D00F8B50F00 I     6268303135869                   1 one
 21001D00F8B50F00 I     6268303135869   6268303136686   3 three
 21001D00F8B50F00 I     6268303135869   6268303136686   2 two
 23000600BAFB0D00 U     6268303136686                   9 nine
 23000600BAFB0D00 D     6268303136686                   3 three
 23000400B9FC0D00 I     6268303136698                  11 eleven
 23000400B9FC0D00 I     6268303136698                  10 ten
```

```
SQL> INSERT INTO test VALUES(10, 'ten');
SQL> INSERT INTO test VALUES(11, 'eleven');
SQL> COMMIT;
```

```
SQL> select * from test;
(as of scn 6268303136698)

 ID STR_VAL
--- ----------
  1 one
  9 nine
 10 ten
 11 eleven
```

# Flashback – Example (7)

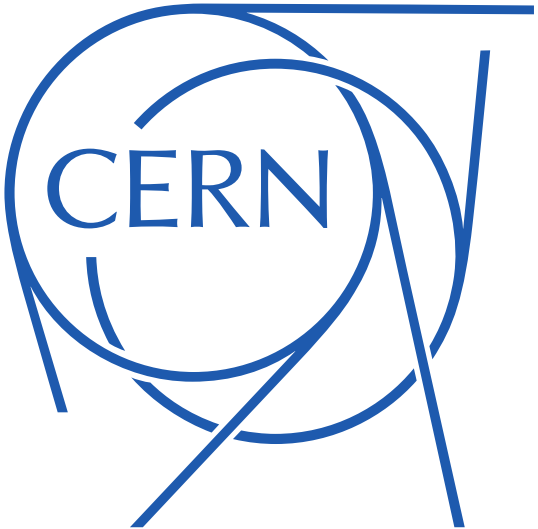
- Full history …

```
SQL> select * from test;
(as of scn 6268303136698)

 ID STR_VAL
--- ----------
  1 one
  9 nine
 10 ten
 11 eleven
```

```
SQL> CREATE TABLE test (id NUMBER(5), str_val VARCHAR2(10));
SQL>
SQL> INSERT INTO test VALUES(1, 'one');
SQL> INSERT INTO test VALUES(2, 'two');
SQL> INSERT INTO test VALUES(3, 'three');
SQL> COMMIT;
SQL>
SQL> UPDATE test SET id = 9, str_val = 'nine' WHERE id =2;
SQL> DELETE FROM test WHERE id = 3;
SQL> COMMIT;
SQL>
SQL> INSERT INTO test VALUES(10, 'ten');
SQL> INSERT INTO test VALUES(11, 'eleven');
SQL> COMMIT;
```

# Thank You!

Marcin.Blaszczyk@cern.ch


CERN