Database Workshop for LHC online/offline developers

SQL (2/2)

Miguel Anjo (IT/ADC)

Miguel.Anjo@cern.ch

http://cern.ch/it-adc

(based on Andrea Valassi' slides on Advanced SQL)

26 January 2005

Previous tutorials: "Database design" and "SQL (1/2)"

- DDL basics
  - Table design, normal forms, constraints, primary/foreign keys
  - CREATE/DROP/ALTER TABLE
  - Data-types, NULL values
- DML basics
  - INSERT, UPDATE, DELETE table rows
- SELECT basics: simple queries
  - Restricting: WHERE, IN, LIKE, AND/OR, +/-…
  - Sorting: ORDER BY, ASC/DESC…
  - Aggregation: COUNT, SUM, MAX, GROUP BY, HAVING…
  - Joins: equijoins, outer joins…

Next hour and half: SQL (2/2)

- **transactions**
- **advanced select**
  - self joins
  - subqueries, inline views, rownum
  - correlated subqueries
  - hierarchical queries
- **granting/revoking**
- **synonyms, db links**
- **views**
- **materialized views**
- **data dictionary**
  - user_% views
- **partitioning**
  - range, hash, composite partitioning
  - global, local indexes
- **Index Organized Tables**
- **other indexes**
  - bitmap, function based, composite
- **flash back queries**
- **multi-dimensional aggregation**
  - cube, rollup

Transactions

- **What if the database crashes in middle of several updates?**
- Transaction is a unit of work that can be either saved to the database (COMMIT) or discarded (ROLLBACK).
- Objective: Read consistency, preview changes before save, group logical related SQL
- Start: Any SQL operation
- End: COMMIT, ROLLBACK, DDL (CREATE TABLE,...)
- Rows changed (UPDATE, DELETE, INSERT) are locked to other users until end of transaction
- Other users wait if try to change locked rows until end of other transaction (READ COMMITTED mode)
- Other users get error if try to change locked rows (SERIALIZABLE mode)
- If crashes, rollbacks.

Transactions

- User A

```
SELECT balance FROM accounts
   WHERE user = A;
      (BALANCE = 300)
```

```
SELECT balance FROM accounts
   WHERE user = A;
      (BALANCE = 300)
```

```
SELECT balance FROM accounts
   WHERE user = A;
      (BALANCE = 300)
```

```
SELECT balance FROM accounts
   WHERE user = A;
      (BALANCE = 50)
```

- User B

```
UPDATE accounts
 SET balance = balance-200
 WHERE user = A;
```

```
SELECT balance FROM
   accounts WHERE user = A;
      (BALANCE = 100)
```

```
UPDATE accounts
  SET balance = balance-50
  WHERE user = A;
```

```
COMMIT;
```

Advanced SQL queries

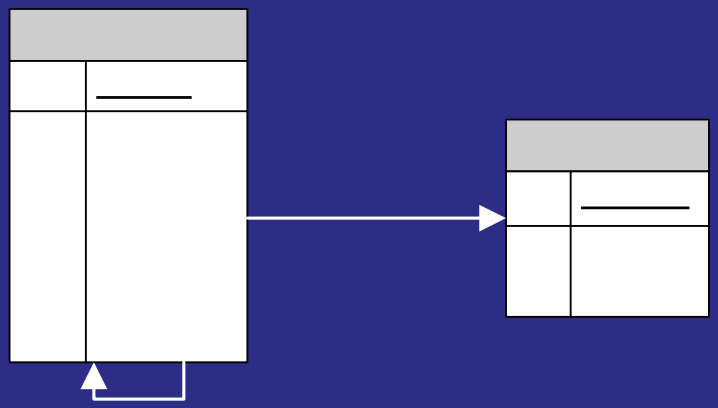
- Queries are often quite complex
  - Selection conditions may depend on results of other queries
  - A query on a table may involve recursive analysis of that table
- *Examples:*
  - *Do some employees earn more than their direct boss?*
  - *Which employees work in the same department as Clark?*
  - *Which employees are the bosses of someone else?*
  - *Display all employees in hierarchical order*
  - *Who are the five employees with higher salary?*
- SQL provides efficient ways to perform such queries
  - Much more efficient than using the application code language!

Self joins (1/2)

- *Normal join*
- relate rows of *two different tables* sharing common values in one or more columns of each table
  - **Typical case: a foreign key referring to a primary key**
  - ***What the name of the employee and his department?***

```
SQL> SELECT e.ename, d.dname
  2  FROM emp e, dept d
  3  WHERE e.deptno = d.deptno;

ENAME      DNAME
---------- --------------
KING       ACCOUNTING
BLAKE      SALES
CLARK      ACCOUNTING
JONES      RESEARCH
(...)
```

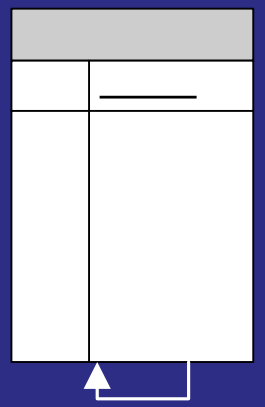


Self joins (2/2)

- *Self joins*
- relate rows of *the same table* sharing common values in two different columns of that table
  - **A foreign key may refer to a primary key in the same table!**
  - ***Which employees receive more than their manager?***

```
SQL> SELECT e.ename,m.ename,
  2  e.sal "EMP SAL", m.sal "MGR SAL"
  3  FROM emp e, emp m
  4  WHERE e.mgr= m.empno
  5  AND e.sal > m.sal;

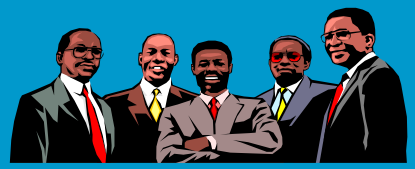
ENAME      ENAME          EMP SAL    MGR SAL
---------- ---------- ---------- ----------
FORD       JONES            3000       2975
SCOTT      JONES            3000       2975
```



Subqueries (1/3)

Who works in the same department as Clark?

Main query



"Which employees work in Clark's department?"

Subquery

("What is Clark's department?")

Subqueries are useful when a query is based on unknown values

Subqueries (2/3)

- ***Who works in the same department as Clark?***

```
SQL> SELECT ename FROM emp
  2   WHERE deptno = (SELECT deptno
  3                   FROM emp
  4                   WHERE ename = 'CLARK');

ENAME
----------
KING
CLARK
MILLER
```

- Logically, think of subqueries in the following way:
  - Subqueries (inner queries) execute once before the main query
  - The subquery results are used by the main query (outer query)

Optimization may actually lead to a different execution implementation
(But you should not worry about that anyway!)

Types of subqueries (3/3)

- Single-row (and single-column) subquery
  - *who works in THE same department as Clark?*

```
SELECT … WHERE dep = (SELECT dep FROM… )
```

- Multiple-row (and single-column) subquery
  - *which are the names of the MANY employees that are someone else's managers?*

```
SELECT … WHERE empno IN (SELECT mgr FROM… )
```

- Multiple-column subquery
  - *who works in the same department(s) AND under the same boss(es) as Clark or Ross?*

```
SELECT … WHERE (dep, mgr) = (SELECT dep, mgr FROM… )
```

- *SQL detects all cardinality inconsistencies*
  - *you cannot*

```
SELECT … WHERE empno = (SELECT empno, mgr FROM… )
```

Correlated subqueries

- ***Who are the employees that receive more than the average salary of their department?***

- In previous subqueries the inner query was executed *ONLY ONCE* before the main query
  - the same inner query result applies to all outer query rows
- Now the inner query is evaluated FOR EACH ROW produced by the outer query

```
SELECT empno, ename, sal, deptno
 FROM emp e
 WHERE sal > (SELECT AVG(sal)
              FROM emp
              WHERE deptno = e.deptno)
              ORDER BY deptno, sal DESC;
```

EMPNO	ENAME	SAL	DEPTNO
7839	KING	5000	10
7902	FORD	3000	20
7788	SCOTT	3000	20
7566	JONES	2975	20
7698	BLAKE	2850	30
7499	ALLEN	1600	30

- In selecting, correlated subqueries are similar to joins
  - Though there may be performance (dis)advantages in both solutions
  - Big difference: they may also be used in updates (for filtering rows)

Subqueries in the FROM clause ("inline view")

- ***What are the employees salary and the maximum salary in their department?***

- We cannot mix group functions with other rows

```
SQL> SELECT ename, sal, MAX(sal), deptno FROM emp;
SELECT ename, sal, MAX(sal), deptno FROM emp
              *
ERROR at line 1:
ORA-00937: not a single-group group function
```

- We can use a "inline view" as the data source on which the main query is executed (FROM clause)

```
SELECT e.ename, e.sal, a.maxsal, a.deptno
  FROM emp e,
       (SELECT max(sal) maxsal, deptno
          FROM emp
          GROUP BY deptno) a
 WHERE e.deptno = a.deptno
 ORDER BY e.deptno, e.sal DESC;
```

ENAME	SAL	MAXSAL	DEPTNO
KING	5000	5000	10
CLARK	2450	5000	10
MILLER	1300	5000	10
SCOTT	3000	3000	20
SMITH	800	3000	20
(...)			

Top-N queries

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

- *We need to use in-line view together with the ROWNUM pseudocolumn)*

```
SELECT empno, ename, job, sal
   FROM
   (SELECT empno, ename, job, sal
      FROM emp
      ORDER BY sal DESC)
   WHERE ROWNUM < 6;
```

```
EMPNO ENAME      JOB       SAL
----- ------ --------- ----
 7839 KING   PRESIDENT 5000
 7902 FORD   ANALYST   3000
 7788 SCOTT  ANALYST   3000
 7566 JONES  MANAGER   2975
 7698 BLAKE  MANAGER   2850
```

- **And the next 5 most well paid?**

```
SELECT empno, ename, job, sal
   FROM (SELECT ROWNUM row#, empno, ename, job, sal
           FROM (SELECT empno, ename, job, sal
                   FROM emp
                   ORDER BY sal DESC))
   WHERE row# BETWEEN 6 and 10;
```

Hierarchical queries

- Display selected data in a hierarchical order (using only one SQL statement!)

  Who sits at the top of the pyramid?

  Who is next in line?

- Syntax:

  SELECT… FROM… WHERE… *START WITH* <condition>
  CONNECT BY key_next_row = *PRIOR* key_last_row

- Pseudo-column LEVEL is the hierarchy level

Hierarchical SQL queries are Oracle-specific

Hierarchical queries: example

```
SELECT empno, ename, mgr, LEVEL
  FROM emp
  CONNECT BY PRIOR empno = mgr;
```

```
EMPNO NAME       MGR LEVEL
----- ---------- --- -----
101   Kochhar    100     1
108   Greenberg  101     2
109   Faviet     108     3
110   Chen       108     3
111   Sciarra    108     3
112   Urman      108     3
113   Popp       108     3
```

Views

- ***I want the users not to see the salary but the department location in a simple query***

```
CREATE VIEW v_emp AS
  (SELECT ename, job, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno);
```

- If emp or dept table changes, v_emp will appear to have changed!
- A view is a stored SQL statement that defines a virtual table

```
SELECT * FROM v_emp;
```

```
ENAME          JOB       DNAME
------ --------- ----------
KING   PRESIDENT ACCOUNTING
BLAKE  MANAGER   SALES
CLARK  MANAGER   ACCOUNTING
(...)
```

Views: benefits and typical usage

- Why use views?
  - To make complex queries easy
    - Hide joins, subqueries, order behind the view
    - Provide different representations of same data
  - To restrict data access
    - Restrict the columns which can be queried
    - Restrict the rows that queries may return
    - Restrict the rows and columns that may be modified
  - To provide abstract interfaces for data independence
    - Users formulate their queries on the views (virtual tables)

Materialized views

- Tables created as subqueries are stored but do not follow changes in base tables
- Views defined as subqueries follow changes in base tables but are not stored
  - Impractical if querying big base table is costly
- *Materialized views* created as subqueries are tables whose stored values follow changes in base tables!
  - They occupy space, but they significantly speed up queries!

Materialized views (snapshots) are Oracle-specific
(although the concept of "summary table" is more generic)

Materialized views and query rewrite

- Typical syntax for materialized views:
  - ```
    CREATE MATERIALIZED VIEW mv2
    BUILD IMMEDIATE
    REFRESH ON COMMIT
    ENABLE QUERY REWRITE
    AS (SELECT… FROM tab1)
    ```
- A query on tab1 will be rewritten using mv2 if the QUERY_REWRITE mechanism is switched on
  - `ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE`
  - `ALTER SESSION SET QUERY_REWRITE_INTEGRITY=TRUSTED;`

  Your DBA must grant you the QUERY REWRITE privilege

  ```
  GRANT QUERY REWRITE TO <user>
  ```

NB: Some m.v.'s are not allowed (e.g. join + group + refresh on commit)
  ```

Materialized views: examples

```
CREATE MATERIALIZED VIEW mv_sal_per_deptno
  BUILD IMMEDIATE
  REFRESH ON COMMIT
  ENABLE QUERY REWRITE
  AS (SELECT deptno "DEPARTMENT", count(empno) "# EMP",
             sum(sal) as "TOTAL SAL"
      FROM emp
      GROUP BY deptno);
```

Now

```
SELECT depto, count(empno) FROM emp GROUP BY deptno;
```

Would probably use mv_sal_per_deptno materialized view (the Oracle optimizer checks if worth)

Materialized views: examples

```
CREATE MATERIALIZED VIEW mv_all_emps
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY')) + 15/24
   AS SELECT * FROM fran.emp@dallas
         UNION
      SELECT * FROM marco.emp@balt
```

Select from remote databases (next slides)

Refresh every Monday at 15:00

Updatable views

- ***What about update v_emp?***
  (the view with employers, job and department name)

- Views can generally be used also to *insert, update or delete* base table rows
  - such views are referred to as *updatable views*

- Many restrictions (some are quite intuitive…)
  - views are not updatable if they contain GROUP/ORDER BY
  - Key preserved (base table row appears at most once)

- For extra consistency, specify "WITH CHECK OPTION"
  CREATE VIEW v1 AS … *WITH CHECK OPTION*
  - cannot insert or update in the base table if not possible to select by the view after that modification!

Grant / Revoke

- ***May I give read access to my tables/views to other user?***

- DBA's can grant/revoke any administrative privilege
- *Only you can grant/revoke privileges (select/insert/update/delete) on the objects you own*
  - *Not even the DBA!*
- Access can be granted on tables or columns
  - Check in **`USER_TAB_PRIVS`** and **`USER_COL_PRIVS`** the privileges you have granted or have been granted (data dictionary tables, wait a few slides more)
  - *Use views to give access to a subset of the data only*
- Accessing a table in another user's schema:

```
SELECT * FROM oradb02.emp;
```

- It is good practice to create synonyms to hide the fact that objects are outside of the schema (manageability)

Synonyms

```
SELECT * FROM oradb31.emp, oradb33.emp, emp@devdb9
  WHERE oradb31.emp.empno = oradb33.emp.empno
    AND oradb31.emp.empno = emp@devdb9.empno;
```

- ***Can it be simpler?***

- Synonyms are alias for tables, views, sequences

```
CREATE SYNONYM emp31 FOR oradb31.emp;

SELECT * FROM emp31; = SELECT * FROM oradb31.emp;
```

- Used to give manageability, hide underlying tables, simplify queries.

Database links

- ***And if I've data in other machine?***

- A database link is an object in the *local* database that allows you to access objects on a *remote* database

```
CREATE DATABASE LINK devdb
    CONNECT TO  scott  IDENTIFIED BY tiger
    USING 'devdb';
```

- Access to tables over a database link:

```
SELECT * FROM emp@devdb;
```

- Use synonyms to hide the fact that a table is remote: `CREATE SYNONYM dev_emp for emp@devdb;`

- Try avoid joins between local and remote tables
  - Push the join as much as possible into the remote tables
  - Oracle 9i→ optimizer tries it best

Sequences

- ***Is there a number generator for unique integers?***

- A "*sequence*" is a database object that *generates* (in/de)creasing *unique integer numbers*
- Can be used as *Primary Key* for the rows of a table
  - In the absence of a more "natural" choice for row ID
- Better than generating ID in application code
  - Very efficient thanks to caching
  - Uniqueness over multiple sessions, transaction safe, no locks
- No guarantee that ID will be continuous
  - rollback, use in >1 tables, concurrent sessions
  - Gaps less likely if caching switched off

Creating and using sequences

- Sequence creation (with many options)

```
CREATE SEQUENCE seq_deptno
INCREMENT BY 10        (default is 1)
MAXVALUE 1000  (default is 10^27)
NOCACHE;       (default is `CACHE 20' values)
```

- Get values:

```
SELECT seq_deptno.NEXTVAL FROM DUAL; -- 1
SELECT seq_deptno.CURRVAL FROM DUAL; -- 1

INSERT INTO dept VALUES
  (seq_dept.NEXTVAL,'HR','ATALANTA'); -- 11
```

Data dictionary views

Schema information:

user_ts_quotas	lists all of the tablespaces + how much can be used, how much is used
user_objects, user_tables, user_views…	objects created in the user's schema
user_sys_privs, user_role_privs, user_tab_privs	system privileges roles granted to the user privileges granted on the user's objects
user_segments, user_extents	storage of the user's objects

- all_* tables with information about accessible objects

Data dictionary views

```
SELECT * FROM user_ts_quotas;

TABLESPACE_NAME     BYTES MAX_BYTES BLOCKS MAX_BLOCKS
--------------- --------- --------- ------ ----------
TRAINING_INDX       65536        -1     16         -1
TRAINING_DATA   869597184        -1 212304         -1
TEMP                    0        -1      0         -1
DATA                    0        -1      0         -1
INDX                    0        -1      0         -1
```

Partitioning

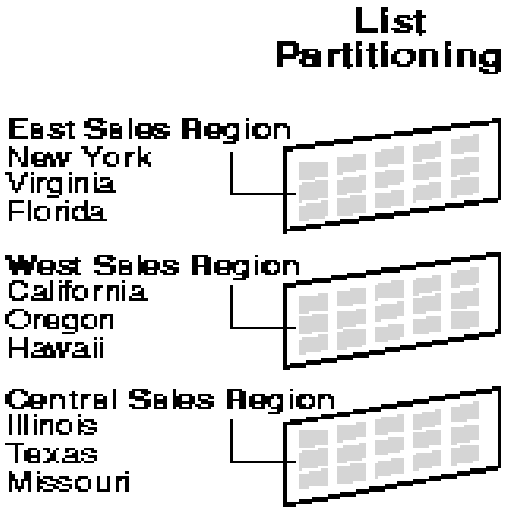
- ***My queries are getting slow as my table is enormous…***
- Partitioning is the key concept to ensure the *scalability* of a database to a very large size
  - data warehouses (large DBs loaded with data accumulated over many years, optimized for read only data analysis)
  - online systems (periodic data acquisition from many sources)

- Tables and indices can be decomposed into smaller and more manageable pieces called *partitions*
  - *Manageability:* data management operations at partition level
    - parallel backup, parallel data loading on independent partitions
  - *Query performance:* partition pruning
    - queries restricted only to the relevant partitions of the table
  - Partitioning is *transparent to user applications*
    - tables/indices logically unchanged even if physically partitioned!

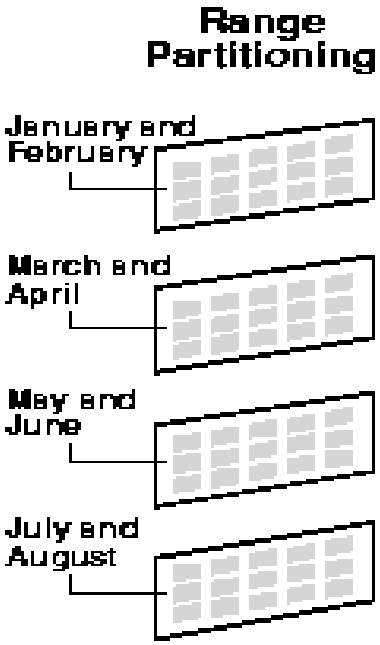
Types of partitioning

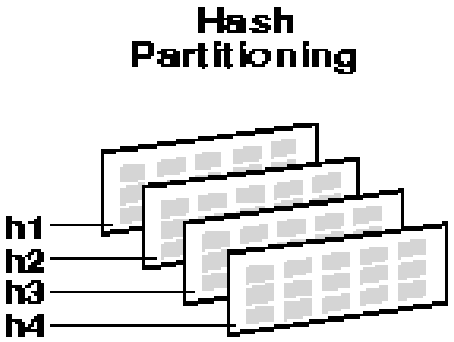
Partitioning according to values of one (or more) column(s)

- Range: partition by predefined ranges of continuous values
- Hash: partition according to hashing algorithm applied by Oracle
- Composite: e.g. range-partition by key1, hash-subpartition by key2



List
(Oracle9i)





Range Hash

(R+H) Composite
(L+H) Composite

Partitioning benefits: partition pruning

Loading data into a table partitioned by date range

```
INSERT INTO sales ( …, sale_date, … )
  VALUES ( …, TO_DATE('3-MARCH-2001','dd-mon-yyyy'), … );
```

Querying data from a table partitioned by date

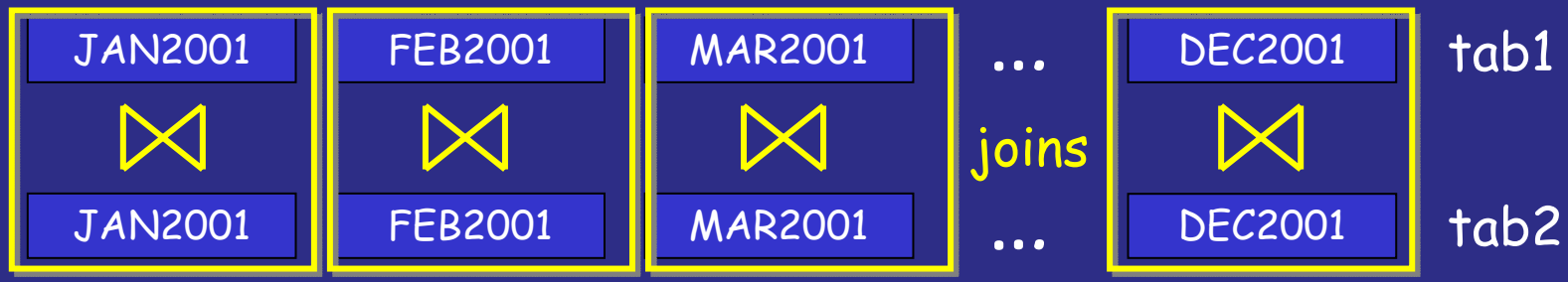

```
SELECT … FROM sales
  WHERE sales_date = TO_DATE ('14-DEC-2001','dd-mon-yyyy');
```

Partition benefits: partition-wise joins

```
SELECT … FROM tab1, tab2 WHERE tab1.key = tab2.key AND …
```

- Without partitioning: global join (query time ~ N x N)

- With partitioning: local joins (query time ~ N)



Partition examples: Range partitioning

```
CREATE TABLE  events
   (event_id    NUMBER(10),
    event_data BLOB)
PARTITION BY RANGE(event_id) (
  PARTITION evts_0_100k
     VALUES LESS THAN (100000)
     TABLESPACE tsa,
  PARTITION evts_100k_200k
     VALUES LESS THAN (200000)
     TABLESPACE tsb,
  PARTITION evts_200k_300k
     VALUES LESS THAN (300000)
     TABLESPACE tsc
);
```

Assigning different partitions to different tablespaces further simplifies data management operations (export/backup) and allows parallel I/O on different filesystems.
[For dedicated servers only! Standard users do not need this!]

Hash partitioning

- Hash partitioning is an alternative to range partitioning
  - When there is no a-priori criterion to group the data
  - When it is important to balance partition sizes
  - When all partitions are equally frequent accessed
    - *Use range partitioning for historical/ageing data!*
- Syntax example:

```
CREATE TABLE files (…, filename, …)
       PARTITION BY HASH (filename) PARTITIONS 5;
```

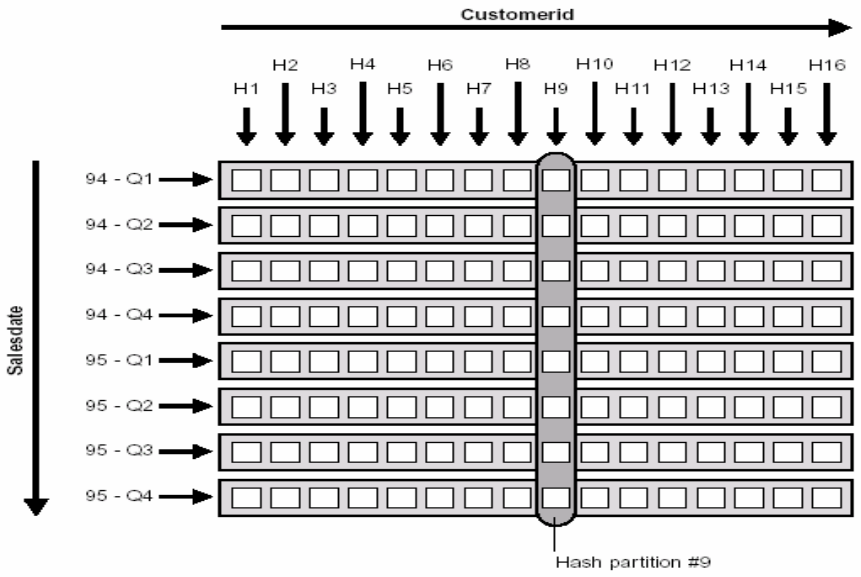
  - Specify the partitioning key(s) and the number of partitions
  - The hashing algorithm cannot be chosen or modified

Composite partitioning

- Use composite partitioning for *very large* tables:
  - First, partition by range (typically, by date ranges)
  - Further subpartition by hash each primary partition

```
CREATE TABLE sales (sale_id, sale_date, customer_id, …)
  PARTITION BY RANGE (sale_date) (
    PARTITION y94q1 VALUES
      LESS THAN TO_DATE(1994-03-01,'YYYY-MM-DD'),
    PARTITION …, PARTITION …)
  SUBPARTITION BY HASH (customer_id) PARTITIONS 16;
```

Example: a SALES table
-Range partitioning by date (quarters)
-Hash subpartitioning by customer ID



Partitioned (local) indexes

- Indexes for partitioned tables can be partitioned too
  - *Local indices: defined within the scope of a partition*

```
CREATE INDEX i_sale_date ON sales (sale_date) LOCAL
```

  - In contrast to *global indexes*: defined on the table as a whole
- Combine the advantages of partitioning and indexing:
  - Partitioning improves query performance by pruning
  - Local index improves performance on full scan of partition
- Prefer local indexes, but global indexes are also needed
  - A Primary Key constraint on a column automatically builds for it a global B*-tree index (PK is globally unique within the table)
- Bitmap indexes on partitioned tables are always local
  - The concept of global index only applies to B*-tree indexes

Index organized tables (IOT)

- *If a table is most often accessed via a PK*, it may be useful to build the table itself like a B*-tree index!
  - In contrast to standard "heap" tables
- Advantages and disadvantages:
  - Faster queries (no need to look up the real table)
  - Reduced size (no separate index, efficient compression)
  - *But performance may degrade if access is not via the PK*
- IOT syntax

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

Bitmap indexes

- Indexes with a bitmap of the column values
- When to use?
  - low cardinalities (columns with few discrete values/<1%)

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	

```
SELECT * FROM costumers
  WHERE mar_status='MARRIED'
    AND region ='CENTRAL'
     OR region ='WEST';
```

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

status = 'married'		region = 'central'		region = 'west'		
0		0		0		0
1		1		1		1
1	AND	0	OR	1	=	1
0		0		1		0
0		1		0		0
1		1		0		1

0		0		0		0
1		1		1		1
1	AND	1	=	1		1
0		1		0		0
0		1		0		0
1		1		1		1

```
CREATE BITMAP INDEX
i_costumers_region ON
costumers(region);
```

Function-based indexes

- Indexes created after applying function to column
  - They speed up queries that evaluate those functions to select data
  - Typical example, if customers are stored as “ROSS”, “Ross”, “ross”:

```
CREATE INDEX customer_name_index
  ON sales (UPPER(customer_name));
```

- Bitmap indices can also be function-based
  - Allowing to map continuous ranges to discrete cardinalities
  - For instance, map dates to quarters:

```
CREATE BITMAP  INDEX sale_date_index
  ON sales (UPPER TO_CHAR(sale_date, 'YYYY"Q"Q'));
```

  - Combining bitmap indices separately built on different columns speeds up multidimensional queries (“AND” of conditions along different axes)

Reverse key indexes

- Index with key reversed (last characters first)
- When to use?
  - Most of keys share first characters (filenames with path)
  - No use of range SELECTs (BETWEEN, <, >, ...)
  - 123, 124, 125 will be indexed as 321, 421, 521
- How to create?

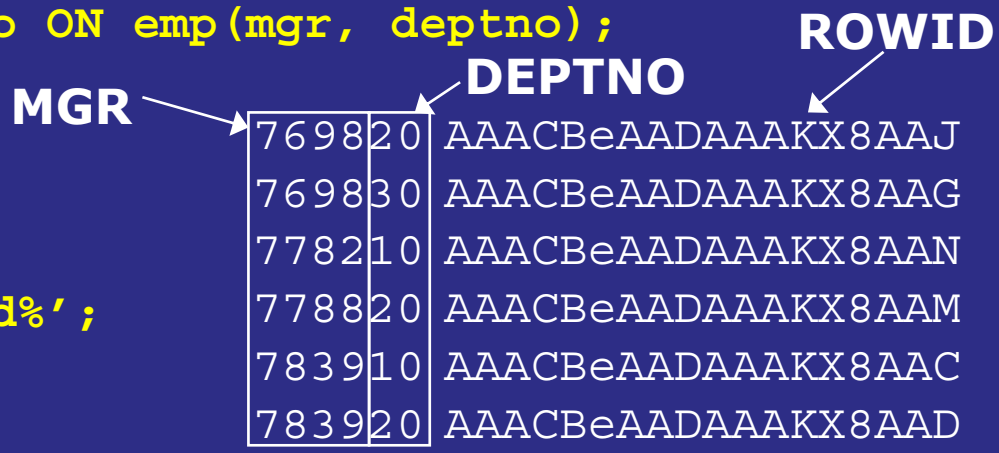
```
CREATE INDEX i_ename ON emp (ename) REVERSE;
```

Composite indexes

- 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
- How to create?
  - Columns with higher selectivity first
  - Columns that can be alone in WHERE clause first

```
CREATE INDEX i_mgr_deptno ON emp(mgr, deptno);

SELECT * FROM emp
  WHERE mgr = 7698
  AND deptno = 30
  AND ename LIKE 'Richard%';
```



Sneak preview of Flashback queries

- **Query all data at point in time**

```
SELECT * FROM emp AS OF '2:00 P.M.' WHERE …
```

- **Flashback Versions Query**
  - **See all versions of a row between two times**
  - **See transactions that changed the row**

```
SELECT * FROM emp
  VERSIONS BETWEEN '2:00 PM' and '3:00 PM' WHERE …
```

- **Flashback Transaction Query**
  - **See all changes made by a transaction**

```
SELECT * FROM dba_transaction_query
WHERE xid = '00020003000002D';
```

Multi-dimensional aggregation

- We saw how to group table rows by values of N columns
- Oracle *data-warehousing* features offer ways to also display integrated totals for the rows in these *slices* :
  - Group first by column x, then (within x-groups) by column y
    SELECT x, y, count(*), … FROM… *GROUP BY ROLLUP (x,y)*
    e.g. display daily sales, as well as monthly and yearly subtotals
  - Group by column x and column y at the same time
    SELECT x, y, count(*), … FROM… *GROUP BY CUBE (x,y)*
    e.g. display sales by product and region, as well as subtotals by product for all regions and subtotals by region for all products

CUBE and ROLLUP in practice

```
SELECT x, y, count(*)
FROM t GROUP
   BY...
```

x	y
A	1
B	2
A	2
B	2
A	1
C	2

x	y	coun
A	1	2
A	2	1
B	2	2
C	2	1

GROUP BY x, y

= GROUP BY x,y
+ y-subtotals ∀x

GROUP BY
ROLLUP (x,y)

x	y	count
A	1	2
A	2	1
A	NULL	*3*
B	2	2
B	NULL	*2*
C	2	1
C	NULL	*1*
NULL	NULL	*6*

= GROUP BY ROLLUP (x,y)
+ x-subtotals ∀y

GROUP BY
CUBE (x,y)

x	y	count
A	1	2
A	2	1
A	NULL	3
B	2	2
B	NULL	2
C	2	1
C	NULL	1
NULL	*1*	*2*
NULL	*2*	*4*
NULL	NULL	6

The rows generated by CUBE/ROLLUP can be found by GROUPING(x) =

- *1 if x is a "fake" NULL from CUBE or ROLLUP*
- *0 otherwise (x is a "true" NULL or is not NULL)*

References

- **"Oracle SQL: The Essential Reference"**
  David Kreines & Ken Jacobs (O'Reilly, 2000)
- **"Mastering Oracle SQL"**
  S. Mishra & A. Beaulieu (O'Reilly, 2002)
- **"Beginning Oracle Programming"**
  S. Dillon, C. Beck & T. Kyte (Wrox, 2002)
- *http://www.ss64.com/ora (Oracle commands)*
- **Oracle online documentation**
  http://otn.oracle.com or *http://oradoc.cern.ch*

Hands-on exercises: SQL

- At 14:00 in Bld: 572 rooms 23-25
- Exercises on this morning SQL tutorials
- Using SQL*Plus connected to training database. More info this afternoon.

Bon appetit!

THE END

Thank you!

Questions?

Contact:
Physics-Database.Support@cern.ch