Adatkezelés tárgy

SQL alapok

Kerepes Tamás
WEBváltó Kft.
tamas.kerepes@webvalto.hu

Bevezető az SQL-be

Az SQL (Structured Query Language) történelme

- 1970: Edgar Frank „Ted” Codd javasolja a relációs adatbázisokat, mint koncepciót
- 1974: IBM System R - az első prototípus
- 1979: Oracle – az első üzletileg elérhető „production” rendszer
- 1973-1974: Chamberlin és Boyce (IBM) vezetik be. Először SEQUEL a név, de az le volt már védve. Így lett SQL
- 1986 óta van SQL szabvány. Amelyik relációs adatbáziskezelő rendszer nem az SQL nyelvet vette át, lényegében eltűnt a süllyesztőben a következő évtizedekben
- Noha mindegyik gyártó terméke SQL nyelvet beszél, sajnos különféle „nyelvjárások” nehezítik a helyzetet
- Ma milliók „beszélik” ezt a nyelvet. Informatikusoknak alapelvárás az ismerete.
- Lehet ismerni felületesen vagy virtuózként

Az SQL alapvető jellemzői

- Erősen hasonlít angol nyelvű mondatokra. Pl.

  SELECT name FROM employees

- Egy nem procedurális nyelv: pl. a fenti „lekérdezésben” nem azt mondjuk el, hogy **hogyan** kell az eredményt megtalálni (tehát a procedúrát), hanem azt, hogy **mit** szeretnénk látni eredményként
- Ezt az SQL nyelvet használjuk az adatok „feldolgozására” akkor is, ha az programunk egyébként Java, C#, C, COBOL, Fortran vagy akármilyen más nyelven van megírva

Az SQL nyelv utasításai (csoportosítás)

- Data Definition Language (DDL): az adamodell kialakítására:

  CREATE TABLE diak (nev VARCHAR(30), eletkor NUMBER)

- Data Manipulation Language (DML) az adatok módosítására:

  INSERT INTO diak VALUES (’Jancsi’, 18)

- Query Language: az adatok lekérdezésére. Pl.:

  SELECT eletkor FROM diak

- „Egyéb” parancsok: tranzakciókezelés, munkamenet kontroll, stb.

Milyen módon kell tudni az SQL nyelvet

- Az SQL szintaxisának az alapjait fejből kell tudni („fújni kell”), különösen a SELECT utasítás alapjait.
- A DML parancsokat is illik fejből tudni.
- DDL parancsból rengeteg létezik: az alapvetőeket illik fejből tudni, a többit meg ‚megérteni’.
- A lekérdezések (SELECT) igen bonyolultak lehetnek. Komoly fejtörést okozhat néha a megírások. Nem ritka, hogy feketeöves szakemberek is több mint egy órán át fogalmaznak meg egy ilyen bonyolult lekérdezést.
- Megesik, hogy a gyakorlatban a SELECT parancs többszáz, esetleg többezer sor hosszúságú.
- Az SQL mélységeiben már eltérnek egymástól a különböző gyártók „nyelvjárásai” - az SQL szabvány igencsak erodálódott

Using DDL Statements to Create Tables

Overview of Tables

- A table is the basic unit of data organization in a relational database.
- A table describes an entity, which is something of significance about which information must be recorded.
- You can create a relational table with the following organizational characteristics (this is heavily vendor specific):
  - A heap-organized table does not store rows in any particular order. The CREATE TABLE statement creates a heap-organized table by default.
  - An index-organized table orders rows according to the primary key values. For some applications, index-organized tables enhance performance and use disk space more efficiently.
  - An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
  - Possibly some others, like partitioned tables and so on

Naming Rules

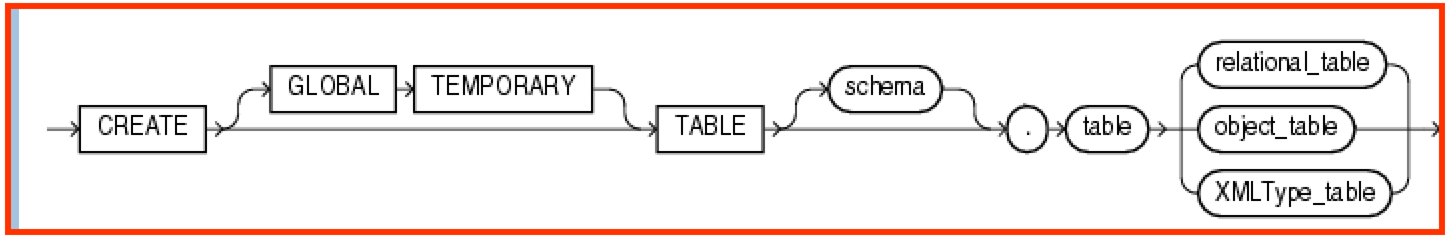
Table names and column names:

- Must begin with a letter
- Must often be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be a reserved word of that vendor. For example the reserved words in Oracle are:

```
SELECT * FROM v$reserved_words
ORDER BY keyword;
```

CREATE TABLE Statement

The general syntax (in case of Oracle):



- You must have:
  - `Appropriate` privilege
  - A storage area
- You specify:
  - Table name
  - Column name, column data type, and column size or number of valuable characters/bytes

The basic Syntax:

```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

Some of the accepted data types (e.g. in Oracle):

Data Type	**Description**
`VARCHAR2(`*`size`*`)`	Variable-length character data
`CHAR(`*`size`*`)`	Fixed-length character data
`NUMBER(p, s)`	Variable-length numeric data
`DATE`	Date and time values
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Maximum size is (4 gigabytes - 1) * (`DB_BLOCK_SIZE`).
`RAW and LONG RAW`	Raw binary data
`BLOB`	Maximum size is (4 gigabytes - 1) * (`DB_BLOCK_SIZE` initialization parameter (8 TB to 128 TB)).
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	Oracle specific: a unique address of a row in the database

Some further data types in Oracle

Data Type	Description
TIMESTAMP	Date with fractional seconds
INTERVAL YEAR TO MONTH	Stored as an interval of years and months
INTERVAL DAY TO SECOND	Stored as an interval of days, hours, minutes, and seconds
BINARY_FLOAT	32-bit floating point number. This data type requires 4 bytes.
BINARY_DOUBLE	64-bit floating point number. This data type requires 8 bytes.
BFILE	Binary data stored in an external file (up to 4 GB)
TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE	All values of TIMESTAMP as well as time zone displacement value

ANSI Data Types

- SQL statements that create tables and clusters can also use ANSI data types .
- Oracle recognizes the ANSI data type name that differs from the Oracle Database data type name

ANSI Data Types and converted (equivalent) Oracle Data Types

ANSI SQL Data Type	Oracle Data Type
CHARACTER(n) CHAR(n)	CHAR(n)
CHARACTER VARYING(n) CHAR VARYING(n)	VARCHAR2(n)
NATIONAL CHARACTER(n) NATIONAL CHAR(n) NCHAR(n)	NCHAR(n)
NATIONAL CHARACTER VARYING(n) NATIONAL CHAR VARYING(n) NCHAR VARYING(n)	NVARCHAR2(n)
NUMERIC[(p,s)] DECIMAL[(p,s)] **(Note 1)**	NUMBER(p,s)
INTEGER INT SMALLINT	NUMBER(p,0)
FLOAT **(Note 2)** DOUBLE PRECISION **(Note 3)** REAL **(Note 4)**	FLOAT(126) FLOAT(126) FLOAT(63)

Creating Tables

- Create the table.

```
CREATE TABLE countries2
   (country_id   CHAR(2) ,
    country_name VARCHAR2(40),
    area         BINARY_FLOAT,
    inhabitants  INTEGER,
    map          BLOB,
    history      CLOB);
table COUNTRIES2 created.
```

- Confirm table creation (in case of Oracle):

```
DESCRIBE countries2
```

```
DESC countries2
Name         Null Type
------------ ---- --------------
COUNTRY_ID        CHAR(2)
COUNTRY_NAME      VARCHAR2(40)
AREA              BINARY_FLOAT()
INHABITANTS       NUMBER(38)
MAP               BLOB
HISTORY           CLOB
```

Retrieving Data Using the SQL `SELECT` Statement

Selecting All and specific Columns

```
SELECT * FROM   divisions;
```

	DIVISION_ID	DIVISION_NAME	MANAGER_ID	CITY	COUNTRY_ID	PARENT_ID
1	1	Head Quarters		San Francisco	US	
2	10	Administration	200	Paris	FR	1
3	20	Marketing	201	Bucharest	RO	1
4	30	Purchasing	114	San Francisco	US	1
5	40	Human Resources	203	Budapest	HU	10
6	50	Shipping	192	Brussels	BE	1
7	90	Executive	100	San Francisco	US	1
8	60	IT Department	103	Canada	CA	90
9	70	Public Relations	204	London	UK	20
10	80	Sales	145	Washington D.C.	US	1

```
SELECT division_id,division_name,city FROM divisions;
```

	DIVISION_ID	DIVISION_NAME	CITY
1	1	Head Quarters	San Francisco
2	10	Administration	Paris
3	20	Marketing	Bucharest
4	30	Purchasing	San Francisco
5	40	Human Resources	Budapest
6	50	Shipping	Brussels
7	90	Executive	San Francisco
8	60	IT Department	Canada
9	70	Public Relations	London
10	80	Sales	Washington D.C.

Writing SQL Statements

- SQL statements are not case-sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- Depending on the tool used, you may end each SQL statement with a semicolon (;) or similar marker.

Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Operator Precedence

```
SELECT worker_id , last_name, salary, 12*salary+100
FROM workers;
```

	WORKER_ID	LAST_NAME	SALARY	12*SALARY+100
1	207	HEMINGWAY	8300	99700
2	100	GAUSS	24000	288100
3	101	EULER	17000	204100
4	102	BERNOULLI	17000	204100
5	103	BERNOULLI	9000	108100
6	104	WILLIS	6000	72100

```
SELECT worker_id,last_name, salary, 12*(salary+100)
FROM workers;
```

	WORKER_ID	LAST_NAME	SALARY	12*(SALARY+100)
1	207	HEMINGWAY	8300	100800
2	100	GAUSS	24000	289200
3	101	EULER	17000	205200
4	102	BERNOULLI	17000	205200
5	103	BERNOULLI	9000	109200
6	104	WILLIS	6000	73200

Defining a Null Value

- Null is a value that is unavailable, unassigned, unknown, or inapplicable.
- Null is not the same as zero or a blank space.

```
SELECT last_name, position_id, salary, commission
FROM  workers;
```

	LAST_NAME	POSITION_ID	SALARY	COMMISSION
1	HEMINGWAY	SALES_REP	8300	0.5
2	GAUSS	ADMIN_PRES	24000	
3	EULER	ADMIN_VP	17000	
4	BERNOULLI	ADMIN_VP	17000	
5	BERNOULLI	IT_MGR	9000	
6	WILLIS	IT_PROG	6000	
7	VERDI	IT_PROG	4800	
8	LORENTZ	IT_PROG	4200	
9	GRIEG	FINANCE_MGR	12000	

...

Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null. Use the NVL function to avoid this!

```
SELECT last_name, salary,commission,12*salary*(1+commission)
FROM   workers;
```

	LAST_NAME	SALARY	COMMISSION	12*SALARY*(1+COMMISSION)
25	STILES	5000		
26	NKOMO	2500		
27	GANDHI	2600		
28	REMBRANDT	2500		
29	RUSSELL	14000	0.4	235200
30	NERUDA	12000	0.3	187200
31	GROSICS	10500	0.2	151200

```
SELECT last_name,salary,commission,
12*salary*(1+NVL(commission,0)) FROM   workers;
```

	LAST_NAME	SALARY	COMMISSION	12*SALARY*(1+NVL(COMMISSION,0))
25	STILES	5000		60000
26	NKOMO	2500		30000
27	GANDHI	2600		31200
28	REMBRANDT	2500		30000
29	RUSSELL	14000	0.4	235200
30	NERUDA	12000	0.3	187200
31	GROSICS	10500	0.2	151200

Defining a Column Alias

A column alias:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name (there can also be the optional `AS` keyword between the column name and the alias)
- Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

Column Aliases

```
SELECT last_name name, 12*salary*(1+commission) ann_sal
FROM   workers;
```

	NAME	ANN_SAL
34	MONET	95040
35	DVORAK	81840
36	FEUERSTEIN	172500
37	SHAKESPEARE	102120
38	ABEL	171600
39	TAYLOR	123840
40	GAUDI	96600
41	WATT	81840
42	BELL	

```
SELECT last_name "Name",
12*salary*(1+commission) "Annual salary" FROM   workers;
```

	Name	Annual salary
34	MONET	95040
35	DVORAK	81840
36	FEUERSTEIN	172500
37	SHAKESPEARE	102120
38	ABEL	171600
39	TAYLOR	123840
40	GAUDI	96600
41	WATT	81840
42	BELL	

Literal Character Strings

- A literal is a character, a number, or a date that is included in the `SELECT` statement.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

Using Literal Character Strings

```
SELECT
first_name||' '||last_name ||' is a '||position_id
        AS "Worker Details"
FROM  workers;
```

	Worker Details
1	Vitus Jonassen BERING is a ACCOUNT_MGR
2	Carl Friedrich GAUSS is a ADMIN_PRES
3	Leonard EULER is a ADMIN_VP
4	Johann BERNOULLI is a ADMIN_VP
5	Gustave FLAUBERT is a FINANCE_ACCOUNT
6	John LENNON is a FINANCE_ACCOUNT
7	Isaac STERN is a FINANCE_ACCOUNT
8	Jose Manuel BAROSO is a FINANCE_ACCOUNT
9	Edvard GRIEG is a FINANCE_MGR
10	Carlos SANTANA is a HR_REP
11	Béla BARTÓK is a HR_REP

Using Literal Character Strings
Alternative Quote (q) operator

```
SELECT 'Position of ', last_name, ' is ',position_id
FROM   workers;
```

	'POSITIONOF'	LAST_NAME	'IS'	POSITION_ID
1	Position of	HEMINGWAY	is	SALES_REP
2	Position of	GAUSS	is	ADMIN_PRES
3	Position of	EULER	is	ADMIN_VP
4	Position of	BERNOULLI	is	ADMIN_VP

```
SELECT division_name,q'#, it's manager id:#',manager_id
FROM   divisions;
```

```
DIVISION_NAME                  Q'#,IT'SMANAGERID:#' MANAGER_ID
------------------------------ -------------------- ----------
Head Quarters                  , it's manager id:
Administration                 , it's manager id:          200
Marketing                      , it's manager id:          201
```

Duplicate Rows

The default display of queries is all rows, including duplicate rows.

```
SELECT division_id
FROM   workers;
```

	DIVISION_ID
1	210
2	90
3	90
4	90
5	60
6	60

```
SELECT DISTINCT division_id
FROM   workers;
```

	DIVISION_ID
1	100
2	30
3	
4	120
5	210

Restricting and Sorting Data

Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the `WHERE` clause:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- The `WHERE` clause follows the `FROM` clause.

```
SELECT worker_id, last_name, position_id, division_id
FROM   workers
WHERE  division_id = 90;
```

	WORKER_ID	LAST_NAME	POSITION_ID	DIVISION_ID
1	100	GAUSS	ADMIN_PRES	90
2	101	EULER	ADMIN_VP	90
3	102	BERNOULLI	ADMIN_VP	90

Character Strings and Dates

- Character strings and date values are enclosed by single quotation marks.
- Character values are case-sensitive, and date values are format-sensitive.
- The default date format is DD-MON-RR.
- Use DATE casting operator

```
SELECT first_name, last_name, salary
FROM   workers
WHERE  last_name = 'GAUSS';
```

```
SELECT first_name, last_name, start_date
FROM   workers
WHERE  start_date>'30-JUN-99';
```

```
SELECT first_name, last_name, start_date
FROM   workers
WHERE  start_date> DATE '1999-06-30';
```

Comparison operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
`BETWEEN ...AND...`	Between two values (inclusive)
`IN(set)`	Match any of a list of values
`LIKE`	Match a character pattern
`IS NULL`	Is a null value

Using Comparison operators

```
SELECT first_name,last_name, position_id,salary
FROM   workers
WHERE  salary >= 13000 ;
```

	FIRST_NAME	LAST_NAME	POSITION_ID	SALARY
1	Carl Friedrich	GAUSS	ADMIN_PRES	23501
2	Leonard	EULER	ADMIN_VP	16501
3	Johann	BERNOULLI	ADMIN_VP	16501
4	Bertrand	RUSSELL	SALES_MGR	13501

```
SELECT first_name,last_name, position_id,salary
FROM   workers
WHERE  salary BETWEEN 12000 and 17000;
```

Lower limit Upper limit

	FIRST_NAME	LAST_NAME	POSITION_ID	SALARY
1	Leonard	EULER	ADMIN_VP	16501
2	Johann	BERNOULLI	ADMIN_VP	16501
3	Bertrand	RUSSELL	SALES_MGR	13501
4	Henri	COANDA	MARKETING_MGR	12501

Using the `IN` operator I.

- Use the `IN` membership operator to test for values in a list
- IN operator implemented with FULL TABLE SCAN and OR operator (in this case)

```
SELECT worker_id, last_name, salary, manager_id
FROM   workers
WHERE  manager_id IN (100, 101, 103) ;
```

	WORKER_ID	LAST_NAME	SALARY	MANAGER_ID
1	101	EULER	17000	100
2	102	BERNOULLI	17000	100
3	104	WILLIS	6000	103
4	106	VERDI	4800	103
5	107	LORENTZ	4200	103
6	108	GRIEG	12000	101

Using the `IN` operator II.

- Use the `IN` membership operator to test for values in a list
- IN operator implemented with `INDEX RANGE SCAN` and `OR` operator (in this case)

```
SELECT worker_id, last_name, salary, division_id
FROM   workers
WHERE  division_id IN (10,90) ;
```

	WORKER_ID	LAST_NAME	SALARY	DIVISION_ID
1	200	JOPLIN	4400	10
2	100	GAUSS	24000	90
3	101	EULER	17000	90
4	102	BERNOULLI	17000	90

Using the `LIKE` operator

- You can combine pattern-matching characters:
- Search operators can contain either literal characters or numbers:
  - `%` denotes zero or many characters.
  - `_` denotes one character.

```
SELECT last_name,first_name,salary
FROM   workers
WHERE  last_name LIKE '_U%' ;
```

	LAST_NAME	FIRST_NAME	SALARY
1	EULER	Leonard	17000
2	GURION	Ben	11000
3	EUSEBIO	Silva Ferreira	3080
4	RUSSELL	Bertrand	14000
5	PUSKIN	Alekszandr Szergejevics	3000

- You can use the `ESCAPE` identifier to search for the actual `%` and `_` symbols.

Examples for LIKE operator

- Use literal in prefix position

```
SELECT last_name,first_name,salary
FROM   workers WHERE  last_name LIKE 'GA%';
```

	LAST_NAME	FIRST_NAME	SALARY
1	GANDHI	Indira	2600
2	GAUSS	Carl Friedrich	24000

- Literals not in prefix position

```
SELECT last_name,first_name,salary
FROM   workers WHERE  last_name LIKE '%A%E%';
```

	LAST_NAME	FIRST_NAME	SALARY
1	FLAUBERT	Gustave	9000
2	SHAKESPEARE	William	7400
3	ABEL	Niels Henrik	11000
4	MICHELANGELO	Buonarroti Simoni	2600

Using the NULL operators

Test for nulls with the `IS NULL` operator.

```
SELECT first_name,last_name, manager_id
FROM   workers
WHERE  manager_id IS NULL;
```

	FIRST_NAME	LAST_NAME	MANAGER_ID
1	Carl Friedrich	GAUSS	

```
SELECT last_name, manager_id,commission
FROM   workers
WHERE  commission IS NOT NULL;
```

	LAST_NAME	MANAGER_ID	COMMISSION
1	HEMINGWAY	205	0.5
2	RUSSELL	100	0.4
3	NERUDA	100	0.3
4	GROSICS	100	0.2
5	BERNSTEIN	145	0.25

Logical operators

Operator	Meaning
`AND`	Returns `TRUE` if *both* component conditions are true
`OR`	Returns `TRUE` if *either* component condition is true
`NOT`	Returns `TRUE` if the following condition is false

Using the AND operator

AND requires both conditions to be true:

```
SELECT worker_id, last_name, position_id, salary
FROM   workers
WHERE  salary >=10000
AND    position_id LIKE '%MGR%' ;
```

	WORKER_ID	LAST_NAME	POSITION_ID	SALARY
1	108	GRIEG	FINANCE_MGR	12000
2	114	GURION	PURCHASE_MGR	11000
3	145	RUSSELL	SALES_MGR	14000
4	147	NERUDA	SALES_MGR	12000
5	149	GROSICS	SALES_MGR	10500
6	201	COANDA	MARKETING_MGR	13000
7	205	BERING	ACCOUNT_MGR	12000

Using the OR operator

OR requires either condition to be true:

```
SELECT worker_id, last_name, position_id, salary
FROM   workers
WHERE  salary >=10000
OR     position_id LIKE '%MGR%' ;
```

	WORKER_ID	LAST_NAME	POSITION_ID	SALARY
1	100	GAUSS	ADMIN_PRES	24000
2	101	EULER	ADMIN_VP	17000
3	102	BERNOULLI	ADMIN_VP	17000
4	103	BERNOULLI	IT_MGR	9000
5	108	GRIEG	FINANCE_MGR	12000
6	114	GURION	PURCHASE_MGR	11000
7	120	MOZART	STOCK_MGR	8000
8	124	COSTNER	STOCK_MGR	5800
9	145	RUSSELL	SALES_MGR	14000
10	147	NERUDA	SALES_MGR	12000
11	149	GROSICS	SALES_MGR	10500
12	168	FEUERSTEIN	SALES_REP	11500
13	174	ABEL	SALES_REP	11000
14	201	COANDA	MARKETING_MGR	13000
15	204	BACH	PR_REP	10000
16	205	BERING	ACCOUNT_MGR	12000

Using the NOT operator

```
SELECT first_name,last_name, position_id
FROM  workers
WHERE  position_id
       NOT IN ('IT_PROG', 'STOCK_CLERK', 'SALES_REP');
```

	FIRST_NAME	LAST_NAME	POSITION_ID
1	Carl Friedrich	GAUSS	ADMIN_PRES
2	Leonard	EULER	ADMIN_VP
3	Johann	BERNOULLI	ADMIN_VP
4	Daniel	BERNOULLI	IT_MGR
5	Edvard	GRIEG	FINANCE_MGR
6	Gustave	FLAUBERT	FINANCE_ACCOUNT
7	John	LENNON	FINANCE_ACCOUNT
8	Isaac	STERN	FINANCE_ACCOUNT
9	Jose Manuel	BAROSO	FINANCE_ACCOUNT
10	Ben	GURION	PURCHASE_MGR

Rules of Precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	`IS [NOT] NULL`, `LIKE`, `[NOT] IN`
5	`[NOT] BETWEEN`
6	Not equal to
7	`NOT` logical condition
8	`AND` logical condition
9	`OR` logical condition

You can use parentheses to override rules of precedence.

Rules of Precedence

```
SELECT last_name, position_id, salary
FROM   workers
WHERE  position_id = 'SALES_MGR'
OR     position_id = 'ADMIN_PRES'
AND    salary > 11000;
```

	LAST_NAME	POSITION_ID	SALARY
1	GAUSS	ADMIN_PRES	24000
2	RUSSELL	SALES_MGR	14000
3	NERUDA	SALES_MGR	12000
4	GROSICS	SALES_MGR	10500

```
SELECT last_name, position_id, salary
FROM   workers
WHERE  (position_id = 'SALES_MGR'
OR     position_id = 'ADMIN_PRES')
AND    salary > 11000;
```

	LAST_NAME	POSITION_ID	SALARY
1	GAUSS	ADMIN_PRES	24000
2	RUSSELL	SALES_MGR	14000
3	NERUDA	SALES_MGR	12000

Using the `ORDER BY` Clause

- Sort retrieved rows with the `ORDER BY` clause:
  - `ASC`: ascending order, default
  - `DESC`: descending order
- The `ORDER BY` clause comes last in the `SELECT` statement:

```
SELECT   last_name, position_id, division_id, start_date
FROM     workers
ORDER BY start_date desc;
```

	LAST_NAME	POSITION_ID	DIVISION_ID	START_DATE
1	FERMI	SHIPPING_CLERK	50	17-MAR-2014
2	CHOPIN	PUBLIC_ACCOUNT	110	07-JUN-2011
3	GRIEG	FINANCE_MGR	100	17-AUG-2009
4	SANTANA	HR_REP	230	10-OCT-2007
5	HEMINGWAY	SALES_REP	210	07-JUN-2007
6	NEWTON	SHIPPING_CLERK	50	03-MAR-2007
7	BERING	ACCOUNT_MGR	110	07-JUN-2004

Examples I.

```
SELECT last_name, position_id, salary
FROM   workers ORDER BY position_id;
```

	LAST_NAME	POSITION_ID	SALARY
1	BERING	ACCOUNT_MGR	12000
2	GAUSS	ADMIN_PRES	24000
3	EULER	ADMIN_VP	17000
4	BERNOULLI	ADMIN_VP	17000
5	FLAUBERT	FINANCE_ACCOUNT	9000

```
SELECT last_name, position_id, salary
FROM   workers ORDER BY salary DESC;
```

	LAST_NAME	POSITION_ID	SALARY
1	GAUSS	ADMIN_PRES	24000
2	EULER	ADMIN_VP	17000
3	BERNOULLI	ADMIN_VP	17000
4	RUSSELL	SALES_MGR	14000
5	COANDA	MARKETING_MGR	13000
6	BERING	ACCOUNT_MGR	12000
7	NERUDA	SALES_MGR	12000
8	GRIEG	FINANCE_MGR	12000
9	FEUERSTEIN	SALES_REP	11500
10	ABEL	SALES_REP	11000
11	GURION	PURCHASE_MGR	11000
12	GROSICS	SALES_MGR	10500

Examples II.

- Sorting by column alias:

```
SELECT last_name, position_id, 12*salary "Annual salary"
FROM   workers
ORDER BY "Annual salary";
```

	LAST_NAME	POSITION_ID	Annual salary
1	SEAGAL	STOCK_CLERK	26400
2	REMBRANDT	STOCK_CLERK	30000
3	NKOMO	STOCK_CLERK	30000
4	MICHELANGELO	SHIPPING_CLERK	31200
5	GANDHI	STOCK_CLERK	31200

- Sorting by multiple columns:

```
SELECT last_name, division_id, 12*salary "Annual salary"
FROM   workers
ORDER BY division_id, salary DESC;
```

	LAST_NAME	DIVISION_ID	Annual salary
1	JOPLIN	10	52800
2	COANDA	20	156000
3	GURION	30	132000
4	KHAN	30	37200
5	PELE	30	34800
6	EUSEBIO	30	33600
7	BARTÓK	40	78000

Using Single-Row Functions to Customize Output

SQL Functions

- Function performs action
- SQL functions are built into the Database and are available for use in various appropriate SQL statements.
- Functions are similar to operators in that they manipulate data items and return a result.
- Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

- A function without any arguments is similar to a pseudocolumn
- If you call a SQL function with an argument of a data type other than the data type expected by the SQL function, then Oracle attempts to convert the argument to the expected data type before performing the SQL function.

Two Types of SQL Functions

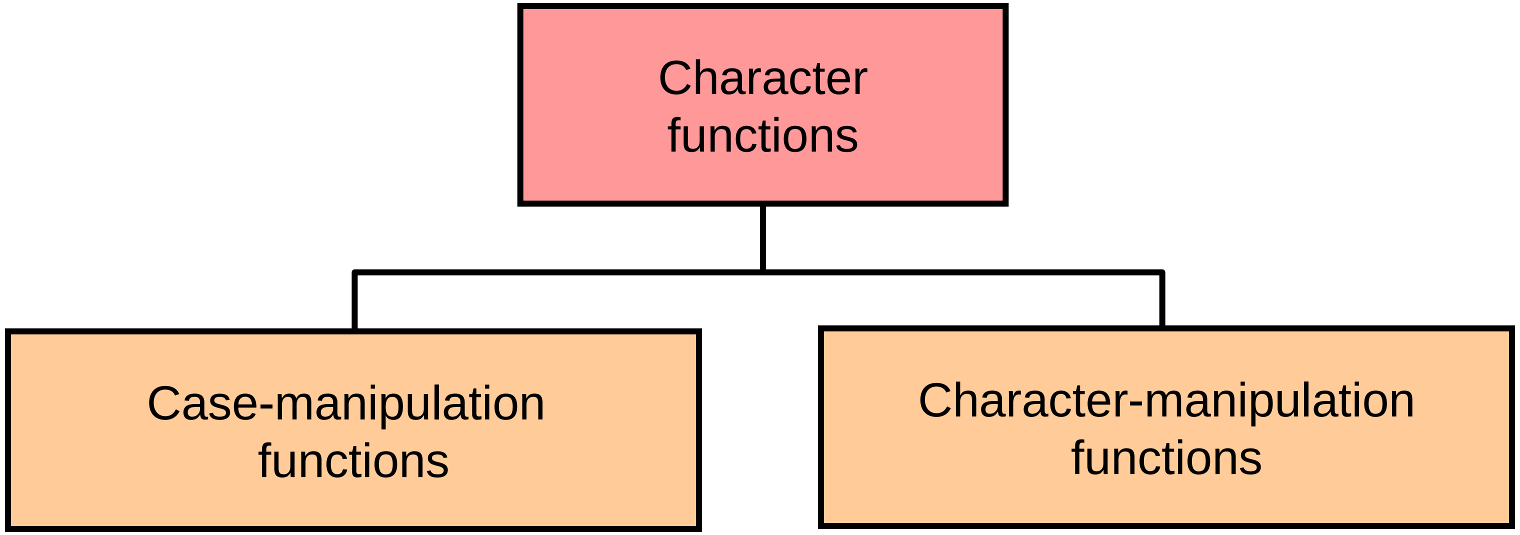
- **Single-Row Functions**
  Single-row functions return a single result row for every row of a queried table or view.
  These functions can appear in select lists, WHERE clauses, START WITH and CONNECT BY clauses, and HAVING clauses.
- **Multi-Row Functions (Aggregate Functions)**
  Aggregate functions return a single result row based on groups of rows, rather than on single rows.
  A group can be:
  - The whole table
  - The subset of the table (filtered by WHERE clause)
  - The set of rows that Oracle Database creates according to GROUP BY clause

Single Row Functions
(some examples only)

Categories of the Single-Row Functions

- Single-Row Functions
  - Numeric Functions
  - Character Functions Returning Character Values
  - Character Functions Returning Number Values
  - NLS Character Functions
  - Datetime Functions
  - General Comparison Functions
  - Conversion Functions
  - Large Object Functions
  - Collection Functions
  - Hierarchical Functions
  - Data Mining Functions
  - XML Functions
  - Encoding and Decoding Functions
  - NULL-Related Functions
  - Environment and Identifier Functions

Character Functions Returning Character Values




```
LOWER
UPPER
INITCAP
NLS_LOWER
NLS_UPPER
NLS_INITCAP
NLSSORT
```

```
CONCAT, CHR
SUBSTR
LENGTH
INSTR
LPAD | RPAD
TRIM |LTRIM | RTRIM
REPLACE, TRANSLATE
```

Case-Manipulation Functions

These functions convert case for character strings:

Function	Result
`LOWER('SQL Course')`	`sql course`
`UPPER('sql Course')`	`SQL COURSE`
`INITCAP('SQL Course')`	`Sql Course`

```
SELECT worker_id, last_name, division_id
FROM   workers
WHERE  last_name = 'gauss';
no rows selected
```

```
SELECT worker_id, last_name, division_id
FROM   workers
WHERE  lower(last_name) = 'gauss';
```

	WORKER_ID	LAST_NAME	DIVISION_ID
1	100	GAUSS	90

Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
`CONCAT('Buda', 'Pest')`	`BudaPest`
`SUBSTR('San Francisco',5,7)`	`Francis`
`LENGTH('SQL Language')`	`12`
`INSTR('SQL Language', 'L',4)`	`5`
`LPAD(salary,10,'*')`	`*****17000`
`RPAD(salary, 10, '*')`	`17000*****`
`REPLACE ('JACK and JUE','J','BL')`	`BLACK and BLUE`
`TRIM('H' FROM 'Hello world')`	`ello world`

Number Functions

Function	Result
POWER(10,0.3010)	2
SQRT(121)	11
ROUND(45.926, 2)	45.93
TRUNC(45.926, 2)	45.92
CEIL, FLOOR	
POWER(10,0.3010)	1.99986
MOD(1600, 300)	100
SIN,COS,TAN	
LN,SINH,COSH, TANH	

```
SELECT SIN(1.57),COS(0),TAN(3.14/4),POWER(10,0.301),
CEIL(1.1),FLOOR(1.9), LN(2.718282) FROM dual;
```

SIN(1.57)	COS(0)	TAN(3.14/4)	POWER(10,0.301)	CEIL(1.1)	FLOOR(1.9)	LN(2.718282)
.999999683	1	.99920399	1.99986187	2	1	1.00000006

Using the `ROUND` and `TRUNC` Function

```
SELECT ROUND(35.911,2), ROUND(35.911,0),
       ROUND(35.911,-1)
FROM   DUAL;
```

	ROUND(35.911,2)	ROUND(35.911,0)	ROUND(35.911,-1)
1	35.91	36	40

```
SELECT TRUNC(35.911,2), TRUNC(35.911,0),
       TRUNC(35.911,-1)
FROM   DUAL;
```

	TRUNC(35.911,2)	TRUNC(35.911,0)	TRUNC(35.911,-1)
1	35.91	35	30

Working with Dates

- SYSDATE is a function that returns:
- Date
- Time
- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

```
SELECT last_name, start_date, SYSDATE,SYSDATE+1 tomorow
FROM   workers
WHERE  start_date < '01-FEB-91';
```

	LAST_NAME	START_DATE	SYSDATE	TOMOROW
1	GAUSS	17-JUN-1987	10-JUN-2015	11-JUN-2015
2	EULER	21-SEP-1989	10-JUN-2015	11-JUN-2015
3	BERNOULLI	03-JAN-1990	10-JUN-2015	11-JUN-2015
4	JOPLIN	17-JUN-1987	10-JUN-2015	11-JUN-2015

Date Functions

Function	Result
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next day of the date specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

Function	Result
MONTHS_BETWEEN ('01-SEP-95','11-JAN-94')	**19.6774194**
ADD_MONTHS ('11-JAN-94',6)	**'11-JUL-94'**
NEXT_DAY ('01-SEP-95','FRIDAY')	**'08-SEP-95'**
LAST_DAY ('01-FEB-95')	**'28-FEB-95'**

Using Date Functions

Assume `SYSDATE = '25-JUL-03'`:

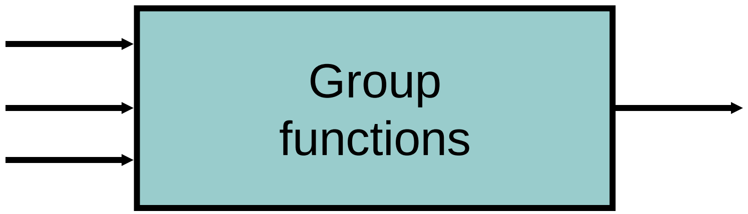
Function	Result
ROUND(SYSDATE,'MONTH')	01-AUG-03
ROUND(SYSDATE ,'YEAR')	01-JAN-04
TRUNC(SYSDATE ,'MONTH')	01-JUL-03
TRUNC(SYSDATE ,'YEAR')	01-JAN-03

Reporting Aggregated Data Using the Group Functions

Types of Group Functions

Group functions operate on sets of rows to give one result per group.

- `AVG`
- `COUNT`
- `MAX`
- `MIN`
- `STDDEV`
- `SUM`
- `VARIANCE`
- …



Using the AVG, SUM ,MIN,MAX and COUNT Functions

COUNT(*) returns the number of rows in a table

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary), COUNT(*)
FROM   workers
WHERE  position_id LIKE '%REP%';
```

```
AVG(SALARY) MAX(SALARY) MIN(SALARY) SUM(SALARY)   COUNT(*)
----------- ----------- ----------- ----------- ----------
 8135.71429       11500        6200      113900         14
```

```
SELECT MIN(last_name),MAX(last_name),
       MIN(start_date), MAX(start_date)
FROM   workers;
```

	MIN(LAST_NAME)	MAX(LAST_NAME)	MIN(START_DATE)	MAX(START_DATE)
1	ABEL	WILLIS	17-JUN-1987	17-MAR-2014

Using the `COUNT` Function and the `DISTINCT` Keyword

COUNT(*expr*) returns the number of rows with non-null values for the *expr*:

```
SELECT COUNT(commission)
FROM   workers
WHERE  division_id = 80;

COUNT(COMMISSION)
-----------------
                9
```

COUNT(DISTINCT expr) returns the number of distinct non-null values of the *expr*.

```
SELECT COUNT(DISTINCT commission)
FROM   workers
WHERE  division_id = 80;

COUNT(DISTINCTCOMMISSION)
-------------------------
                        6
```

Creating Groups of Data: GROUP BY Clause Syntax

```
SELECT     column, group_function(column)
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[ORDER BY  column];
```

You can divide rows in a table into smaller groups by using the `GROUP BY` clause.

Using the `GROUP BY` Clause

All columns in the `SELECT` list that are not in group functions must be in the `GROUP BY` clause.

```
SELECT    division_id,
          TO_CHAR(AVG(salary),'99,999.99') avg_sal
FROM      workers
GROUP BY  division_id ;
```

	DIVISION_ID	AVG_SAL
1	100	6,600.00
2	30	5,170.00
3		7,000.00
4	120	2,500.00
5	210	8,525.00
6	90	19,333.33
7	20	13,000.00
8	70	10,000.00
9	230	6,500.00

Illegal Queries
Using Group Functions

Any column or expression in the `SELECT` list that is not an aggregate function must be in the `GROUP BY` clause:

```
SELECT division_id, COUNT(last_name)
FROM   workers;
```

```
Error starting at line : 1 in command -
SELECT division_id, COUNT(last_name)
FROM   workers
Error at Command Line : 1 Column : 8
Error report -
SQL Error: ORA-00937: not a single-group group function
00937. 00000 -  "not a single-group group function"
```

Column missing in the `GROUP BY` clause

Illegal Queries
Using Group Functions

- You cannot use the `WHERE` clause to restrict groups.
- You use the `HAVING` clause to restrict groups.
- You cannot use group functions in the `WHERE` clause.

```
SELECT   division_id, AVG(salary)
FROM     workers
WHERE    AVG(salary) > 8000
GROUP BY division_id;
```

```
Error starting at line : 1 in command -
SELECT   division_id, AVG(salary)
FROM     workers
WHERE    AVG(salary) > 8000
GROUP BY division_id
Error at Command Line : 3 Column : 10
Error report -
SQL Error: ORA-00934: group function is not allowed here
00934. 00000 -  "group function is not allowed here"
```

Cannot use the `WHERE` clause to restrict groups

Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the database server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

Using the `HAVING` Clause

```
SELECT   position_id, SUM(salary) PAYROLL
FROM     workers
WHERE    position_id  LIKE '%S%'
GROUP BY position_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

```
SELECT T.qualified_salary, COUNT(*), SUM(SALARY), COUNT(*)
FROM (SELECT last_name,salary,
      CASE WHEN salary<5000  THEN 'Low'
           WHEN salary<10000 THEN 'Medium'
           WHEN salary<20000 THEN 'Good'
           ELSE 'Excellent'   END qualified_salary
      FROM WORKERS) T
GROUP BY T.qualified_salary
HAVING COUNT(*)>1;
```

	QUALIFIED_SALARY	COUNT(*)	SUM(SALARY)	COUNT(*)_1
1	Good	12	151000	12
2	Low	18	58680	18
3	Medium	22	162200	22

Where to filter?

```
SELECT   city, customer_name, sum(bonus)
FROM     customers
WHERE    city='Montreal'
GROUP BY city, customer_name;
```

	CITY	CUSTOMER_NAME	SUM(BONUS)
1	Montreal	Paulette's Coffee Shop	34
2	Montreal	Clifton Lunch	4052
3	Montreal	Bellucci's	4130
4	Montreal	Globus Office	621
5	Montreal	Velia's Cafe	7079
6	Montreal	Club 427	9110
7	Montreal	Cafe Sevilla	1958

OPERATION	OBJECT_NAME
SELECT STATEMENT	
HASH (GROUP BY)	
TABLE ACCESS (BY INDEX ROWID BATCHED)	CUSTOMERS
INDEX (RANGE SCAN)	CUSTOMERS_CITY_IX
Access Predicates	
CITY='Montreal'	

```
SELECT   city, customer_name, sum(bonus)
FROM     customers
GROUP BY city, customer_name
HAVING   city='Montreal';
```

	CITY	CUSTOMER_NAME	SUM(BONUS)
1	Montreal	Paulette's Coffee Shop	34
2	Montreal	Clifton Lunch	4052
3	Montreal	Bellucci's	4130
4	Montreal	Globus Office	621
5	Montreal	Velia's Cafe	7079
6	Montreal	Club 427	9110
7	Montreal	Cafe Sevilla	1958

OPERATION	OBJECT_NAME
SELECT STATEMENT	
FILTER	
Filter Predicates	
CITY='Montreal'	
HASH (GROUP BY)	
TABLE ACCESS (FULL)	CUSTOMERS

Nesting Group Functions

Display the maximum average salary:

```
SELECT   MAX(AVG(salary))
FROM     workers
GROUP BY division_id;
```

	MAX(AVG(SALARY))
1	18834.3333333333333333333333333333333333

Displaying Data
from Multiple Tables

Types of Joins

Different types of Join-s and different syntaxes:

- Equi-Joins
  - Natural joins
  - `USING` clause
  - Vendor-specific syntax variations
- Outer Joins
  - Left Outer Join
  - Right Outer Join
  - Full (or two-sided) outer join
- Non equi-joins
- Cross joins

Retrieving Records with Natural Joins

```
SELECT division_id, division_name, city,
       country_id, country_name
FROM         divisions
NATURAL JOIN countries ;
```

	DIVISION_ID	DIVISION_NAME	CITY	COUNTRY_ID	COUNTRY_NAME
1	210	Government Sales	Canberra	AU	Australia
2	50	Shipping	Brussels	BE	Belgium
3	110	Accounting	Brussels	BE	Belgium
4	60	IT Department	Canada	CA	Canada
5	140	IT Designers	Bern	CH	Switzerland
6	120	Treasury	Copenhagen	DK	Denmark

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		24
MERGE JOIN		24
TABLE ACCESS (BY INDEX ROWID)	DIVISIONS	24
INDEX (FULL SCAN)	DIVISIONS_COUNTRY_ID_IX	24
SORT (JOIN)		27
Access Predicates		
DIVISIONS.COUNTRY_ID=COUNTRIES.COUNTRY_ID		
Filter Predicates		
DIVISIONS.COUNTRY_ID=COUNTRIES.COUNTRY_ID		
TABLE ACCESS (FULL)	COUNTRIES	27

Retrieving Records with the `USING` Clause

```
SELECT e.worker_id, e.last_name,
       division_id, division_name
FROM   workers e JOIN divisions d
USING  (division_id) ;
```

```
SELECT e.worker_id, e.last_name,          division_id,division_name
FROM   workers e JOIN divisions d USING (division_id)

Plan hash value: 2327659369

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost  (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |       |       |     4 (100)|          |
|   1 |  MERGE JOIN                  |                |    52 |  1716 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DIVISIONS      |    24 |   408 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | DIVISION_ID_PK |    24 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |                |    53 |   848 |     2   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | WORKERS        |    53 |   848 |     2   (0)| 00:00:01 |
```

Traditional Join (Oracle Syntax)

```
SELECT e.worker_id, e.last_name,
       e.division_id, division_name
FROM   workers e , divisions d
WHERE  e.division_id = d.division_id ;
```

```
SELECT e.worker_id, e.last_name,          e.division_id,division_name
FROM   workers e , divisions d WHERE e.division_id=d.division_id

Plan hash value: 2327659369

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |       |       |     4 (100)|          |
|   1 |  MERGE JOIN                  |                |    52 |  1716 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DIVISIONS      |    24 |   408 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | DIVISION_ID_PK |    24 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |                |    53 |   848 |     2   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | WORKERS        |    53 |   848 |     2   (0)| 00:00:01 |
```

Retrieving Records with the ON Clause

```
SELECT e.worker_id, e.last_name,
       e.division_id, division_name
FROM   workers e JOIN divisions d
on     e.division_id=d.division_id ;
```

```
SELECT e.worker_id, e.last_name,          e.division_id,division_name
FROM   workers e JOIN divisions d on e.division_id=d.division_id

Plan hash value: 2327659369

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |       |       |     4 (100)|          |
|   1 |  MERGE JOIN                   |                |    52 |  1716 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DIVISIONS      |    24 |   408 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | DIVISION_ID_PK |    24 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |                |    53 |   848 |     2   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | WORKERS        |    53 |   848 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Self-Joins Using the ON Clause

WORKERS (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

WORKERS (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...

MANAGER_ID in the WORKER (WORKERS) table is equal to EMPLOYEE_ID in the MANAGER (WORKERS) table.

Self-Joins Using the `ON` Clause

```
SELECT e.worker_id, e.last_name emp, m.last_name mgr
FROM   workers e JOIN workers m
ON     e.manager_id = m.worker_id;
```

MANAGER_ID in the *WORKERS* table is equal to *WORKER_ID* in the *MANAGER* (*WORKERS*) table.

	WORKER_ID	EMP	MGR
1	201	COANDA	GAUSS
2	149	GROSICS	GAUSS
3	147	NERUDA	GAUSS
4	145	RUSSELL	GAUSS
5	124	COSTNER	GAUSS
6	123	SANTANA	GAUSS
7	120	MOZART	GAUSS

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		52	4
HASH JOIN		52	4
Access Predicates			
E.MANAGER_ID=M.WORKER_ID			
NESTED LOOPS			
NESTED LOOPS		52	4
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	WORKERS	54	2
INDEX (UNIQUE SCAN)	WORKERS_WORKER_ID_PK		
Access Predicates			
E.MANAGER_ID=M.WORKER_ID			
TABLE ACCESS (BY INDEX ROWID)	WORKERS	1	2
TABLE ACCESS (FULL)	WORKERS	54	2

Joining 3 tables

- Most database engines join two row sources at a time
- Subsequent joins can be used if necessary

```
SELECT e.worker_id, e.last_name, e.division_id,
       d.division_name, d.city, c.country_name
FROM   workers e JOIN divisions d
ON     e. division_id = d. division_id
JOIN   countries c
ON     c.country_id = d.country_id;
```

	WORKER_ID	LAST_NAME	DIVISION_ID	DIVISION_NAME	CITY	COUNTRY_NAME
1	207	HEMINGWAY	210	Government Sales	Canberra	Australia
2	100	GAUSS	90	Executive	San Francisco	United States Of America
3	101	EULER	90	Executive	San Francisco	United States Of America
4	102	BERNOULLI	90	Executive	San Francisco	United States Of America
5	103	BERNOULLI	60	IT Department	Canada	Canada
6	104	WILLIS	60	IT Department	Canada	Canada
7	106	VERDI	60	IT Department	Canada	Canada
8	107	LORENTZ	60	IT Department	Canada	Canada

Non-Equijoins

WORKERS

	FIRST_NAME	LAST_NAME	SALARY
1	Ernest Miller	HEMINGWAY	8300
2	Carl Friedrich	GAUSS	24000
3	Leonard	EULER	17000
4	Johann	BERNOULLI	17000
5	Daniel	BERNOULLI	9000
6	Bruce	WILLIS	6000
7	Giuseppe	VERDI	4800
8	Hendrik	LORENTZ	4200
9	Edvard	GRIEG	12000
10	Gustave	FLAUBERT	9000
11	John	LENNON	8200

JOB_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

Salary in the `WORKERS` table must be between lowest salary and highest salary in the `JOB_GRADES` table.

Example for Non-Equijoin

```
SELECT e.first_name,e.last_name, e.salary, j.grade_level
FROM   workers e JOIN job_grades j
ON     e.salary
       BETWEEN j.lowest_sal AND j.highest_sal
WHERE last_name LIKE 'S%' or last_name like 'G%'
```

	FIRST_NAME	LAST_NAME	SALARY	GRADE_LEVEL
1	Steven	SEAGAL	2200	A
2	Indira	GANDHI	2600	A
3	Nobby	STILES	5000	B
4	Carlos	SANTANA	6500	C
5	Antoni	GAUDI	7000	C
6	William	SHAKESPEARE	7400	C
7	Isaac	STERN	7700	C
8	Gyula	GROSICS	10500	D
9	Ben	GURION	11000	D
10	Edvard	GRIEG	12000	D
11	Carl Friedrich	GAUSS	24000	E

```
OPERATION                                              OBJECT_NAME
SELECT STATEMENT
  MERGE JOIN
    SORT (JOIN)
      TABLE ACCESS (FULL)                              JOB_GRADES
        Filter Predicates
          J.HIGHEST_SAL>0
    FILTER
      Filter Predicates
        E.SALARY<=J.HIGHEST_SAL
      SORT (JOIN)
        Access Predicates
          E.SALARY>=J.LOWEST_SAL
        Filter Predicates
          E.SALARY>=J.LOWEST_SAL
        TABLE ACCESS (FULL)                            WORKERS
```

LEFT OUTER JOIN

- WORKERS is the driving table (Left from the operator)
- All rows are retrieved from driving table

```
SELECT e.last_name, d.division_id, d.division_name
FROM   workers e LEFT OUTER JOIN divisions d
ON   e.division_id = d.division_id;
```

	LAST_NAME	DIVISION_ID	DIVISION_NAME
46	MOZART	160	Shareholder Services
47	BIZET	200	IT Helpdesk
48	WATT	210	Government Sales
49	TAYLOR	210	Government Sales
50	ABEL	210	Government Sales
51	HEMINGWAY	210	Government Sales
52	SANTANA	230	Recruiting
53	GAUDI		

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		54	5
HASH JOIN (OUTER)		54	5
Access Predicates			
E.DIVISION_ID=D.DIVISION_ID(+)			
NESTED LOOPS (OUTER)		54	5
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	WORKERS	54	2
TABLE ACCESS (BY INDEX ROWID)	DIVISIONS	1	3
INDEX (UNIQUE SCAN)	DIVISION_ID_PK		
Access Predicates			
E.DIVISION_ID=D.DIVISION_ID(			
TABLE ACCESS (FULL)	DIVISIONS	24	3

RIGHT OUTER JOIN

- DIVISIONS is the driving table (Right from the operator)
- All rows are retrieved from driving table

```
SELECT e.last_name, d.division_id, d.division_name
FROM   workers e RIGHT OUTER JOIN divisions d
ON     e.division_id = d.division_id;
```

	LAST_NAME	DIVISION_ID	DIVISION_NAME
42	NKOMO	120	Treasury
43	GANDHI	130	Corporate Tax
44		140	IT Designers
45		150	IT Programers
46	COSTNER	160	Shareholder Services
47	MOZART	160	Shareholder Services

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		60	4
MERGE JOIN (OUTER)		60	4
TABLE ACCESS (BY INDEX ROWID)	DIVISIONS	24	2
INDEX (FULL SCAN)	DIVISION_ID_PK	24	1
SORT (JOIN)		54	2
Access Predicates			
E.DIVISION_ID(+)=D.DIVISION_ID			
Filter Predicates			
E.DIVISION_ID(+)=D.DIVISION_ID			
TABLE ACCESS (FULL)	WORKERS	54	2

FULL OUTER JOIN

```
SELECT e.last_name, d.division_id, d.division_name
FROM   workers e FULL OUTER JOIN divisions d
ON   e.division_id = d.division_id;
```

	LAST_NAME	DIVISION_ID	DIVISION_NAME
40	GAUDI		
41	WATT	210	Government Sales
42	BELL	50	Shipping
43	NEWTON	50	Shipping
44	FERMI	50	Shipping
45	PUSKIN	50	Shipping
46	MICHELANGELO	50	Shipping
47	JOPLIN	10	Administration
48	COANDA	20	Marketing
49	BIZET	200	IT Helpdesk
50	BARTÓK	40	Human Resources
51	BACH	70	Public Relations
52	BERING	110	Accounting
53	CHOPIN	110	Accounting
54		220	Retail Sales

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		59
VIEW	VW_FOJ_0	59
HASH JOIN (FULL OUTER)		59
Access Predicates		
E.DIVISION_ID=D.DIVISION_ID		
TABLE ACCESS (FULL)	DIVISIONS	24
TABLE ACCESS (FULL)	WORKERS	53

Creating Cross Joins (Cartesian Product)

- The `CROSS JOIN` clause produces the cross-product of two tables.
- This is also called a Cartesian product between the two tables.

```
SELECT last_name, division_name
FROM   workers
CROSS JOIN divisions;
```

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		1272
MERGE JOIN (CARTESIAN)		1272
TABLE ACCESS (FULL)	DIVISIONS	24
BUFFER (SORT)		53
INDEX (FAST FULL SCAN)	WORKERS_NAME_IX	53

Using Subqueries to Solve Queries

Subquery Syntax

```
SELECT    select_list
FROM      table
WHERE     expr operator
                     (SELECT        select_list
                      FROM          table);
```

- Logically:
  - the subquery (inner query) executes at least once before the main query (outer query).
  - The result of the subquery is used by the main query.
- Physically:
  - The optimizer decides how to implement the problem
  - With or without Query Transformation

Using a Subquery

- The result of the subquery can't be displayed.

```
SELECT first_name, last_name, salary
FROM   workers
WHERE  salary > (SELECT AVG(salary)
                 FROM   workers);
```

7469.43

	FIRST_NAME	LAST_NAME	SALARY
1	Ernest Miller	HEMINGWAY	8300
2	Carl Friedrich	GAUSS	24000
3	Leonard	EULER	17000
4	Johann	BERNOULLI	17000
5	Daniel	BERNOULLI	9000
6	Edvard	GRIEG	12000

OPERATION	OBJECT_NAME
SELECT STATEMENT	
TABLE ACCESS (FULL)	WORKERS
Filter Predicates	
SALARY> (SELECT AVG(SALARY) FROM WORKERS WORKERS)	
SORT (AGGREGATE)	
TABLE ACCESS (FULL)	WORKERS

Using IN-LINE view

- With in-line view the result of the subquery can be displayed .

```
SELECT first_name, last_name, salary,
       (SELECT AVG(salary) FROM   workers) AVG_SALARY
FROM   workers
WHERE  salary >(SELECT AVG(salary) FROM   workers);
```

FIRST_NAME	LAST_NAME	SALARY	AVG_SALARY
Ernest Miller	HEMINGWAY	8300	7,469.43
Carl Friedrich	GAUSS	24000	7,469.43
Leonard	EULER	17000	7,469.43
Johann	BERNOULLI	17000	7,469.43
Daniel	BERNOULLI	9000	7,469.43

OPERATION	OBJECT_NAME
SELECT STATEMENT	
SORT (AGGREGATE)	
TABLE ACCESS (FULL)	WORKERS
TABLE ACCESS (FULL)	WORKERS
Filter Predicates	
SALARY> (SELECT AVG(SALARY) FROM WORKERS WORKERS)	
SORT (AGGREGATE)	
TABLE ACCESS (FULL)	WORKERS

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =                              ST_CLERK
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141)
AND    salary >                                 2600
                (SELECT salary
                 FROM   employees
                 WHERE  employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary =
                (SELECT MIN(salary)
                 FROM   employees);
```

2500

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

What Is Wrong with This Statement?

```
SELECT employee_id, last_name
FROM   employees
WHERE  salary =
                (SELECT   MIN(salary)
                 FROM     employees
                 GROUP BY department_id);
```

```
ERROR at line 4:
ORA-01427: single-row subquery returns more than
one row
```

Single-row operator with multiple-row subquery

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
`IN`	Equal to any member in the list
`ANY`	Compare value to each value returned by the subquery
`ALL`	Compare value to every value returned by the subquery

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT worker_id, last_name, position_id, salary
FROM   workers
WHERE  salary < ANY (SELECT salary FROM  workers
                     WHERE  position_id = 'PURCHASE_CLERK')
```

	POSITION_ID	SALARY
1	PURCHASE_CLERK	3080
2	PURCHASE_CLERK	3190
3	PURCHASE_CLERK	3410
4	PURCHASE_CLERK	3600

	WORKER_ID	LAST_NAME	POSITION_ID	SALARY
1	128	SEAGAL	STOCK_CLERK	2200
2	144	REMBRANDT	STOCK_CLERK	2500
3	140	NKOMO	STOCK_CLERK	2500
4	199	MICHELANGELO	SHIPPING_CLERK	2600
5	143	GANDHI	STOCK_CLERK	2600
6	126	TOLSZTOJ	STOCK_CLERK	2700
7	195	FERMI	SHIPPING_CLERK	2800
8	197	PUSKIN	SHIPPING_CLERK	3000
9	117	EUSEBIO	PURCHASE_CLERK	3080
10	116	PELE	PURCHASE_CLERK	3190
11	125	ROBERTS	STOCK_CLERK	3200
12	115	KHAN	PURCHASE_CLERK	3410

Using the `ALL` Operator in Multiple-Row Subqueries

Display workers whose salaries less than all IT_PROG's salary

```
SELECT worker_id, last_name, position_id, salary
FROM   workers
WHERE  salary < ALL (SELECT salary FROM  workers
                     WHERE  position_id = 'IT_PROG')
```

	POSITION_ID	SALARY
1	IT_PROG	4200
2	IT_PROG	4400
3	IT_PROG	4800
4	IT_PROG	6000

	WORKER_ID	LAST_NAME	POSITION_ID	SALARY
1	192	BELL	SHIPPING_CLERK	4000
2	193	NEWTON	SHIPPING_CLERK	3900
3	137	CRUYFF	PURCHASE_CLERK	3600
4	115	KHAN	PURCHASE_CLERK	3410
5	125	ROBERTS	STOCK_CLERK	3200
6	116	PELE	PURCHASE_CLERK	3190
7	117	EUSEBIO	PURCHASE_CLERK	3080
8	197	PUSKIN	SHIPPING_CLERK	3000
9	195	FERMI	SHIPPING_CLERK	2800
10	126	TOLSZTOJ	STOCK_CLERK	2700
11	143	GANDHI	STOCK_CLERK	2600
12	199	MICHELANGELO	SHIPPING_CLERK	2600
13	140	NKOMO	STOCK_CLERK	2500
14	144	REMBRANDT	STOCK_CLERK	2500
15	128	SEAGAL	STOCK_CLERK	2200

Null Values in a Subquery

- Never use NOT IN operator for subquery which can contain NULL values.

```
SELECT last_name
FROM   workers
WHERE  worker_id NOT IN (SELECT manager_id FROM  workers);
no rows selected
```

- Using NVL function you can avoid this situation.

```
SELECT first_name, last_name, position_id
FROM   workers
WHERE  worker_id NOT IN
                 (SELECT NVL(manager_id,-1) FROM  workers);
```

	FIRST_NAME	LAST_NAME	POSITION_ID
1	Leonard	BERNSTEIN	SALES_REP
2	Bruce	WILLIS	IT_PROG
3	Johann Sebastian	BACH	PR_REP
4	Béla	BARTÓK	HR_REP
5	Hendrik	LORENTZ	IT_PROG
6	Alekszandr Szergejevics	PUSKIN	SHIPPING_CLERK

Correlated Subquery vs. Join I.

```
SELECT   A.last_name, A.salary,
         A.division_id
FROM     workers A
WHERE    A.salary > (SELECT  ROUND(AVG(salary)) salavg
                     FROM    workers B
                     WHERE   A.division_id = B.division_id);
```

OPERATION	OBJECT_NAME	
SELECT STATEMENT		
HASH JOIN		
Access Predicates		
A.DIVISION_ID=ITEM_1		
Filter Predicates		
A.SALARY>SALAVG		
NESTED LOOPS		
NESTED LOOPS		4
STATISTICS COLLECTOR		
VIEW	VW_SQ_1	17
HASH (GROUP BY)		17
TABLE ACCESS (FULL)	WORKERS	53
INDEX (RANGE SCAN)	WORKERS_DIVISION_ID_IX	
Access Predicates		
A.DIVISION_ID=ITEM_1		
TABLE ACCESS (BY INDEX ROWID)	WORKERS	1
Filter Predicates		
A.SALARY>SALAVG		
TABLE ACCESS (FULL)	WORKERS	53

	LAST_NAME	SALARY	DIVISION_ID
1	GAUSS	24000	90
2	BERNOULLI	9000	60
3	GRIEG	12000	100
4	FLAUBERT	9000	100

Correlated Subquery vs. Join II.

```
SELECT   A.last_name, A.salary,
         A.division_id, B.salavg
FROM     workers A,
         (SELECT  division_id, ROUND(AVG(salary)) salavg
          FROM    workers GROUP BY division_id) B
WHERE    A.division_id = B.division_id
AND      A.salary > B.salavg;
```

OPERATION	OBJECT_NAME	
SELECT STATEMENT		4
HASH JOIN		4
Access Predicates		
A.DIVISION_ID=ITEM_1		
Filter Predicates		
A.SALARY>SALAVG		
NESTED LOOPS		
NESTED LOOPS		4
STATISTICS COLLECTOR		
VIEW	VW_SQ_1	17
HASH (GROUP BY)		17
TABLE ACCESS (FULL)	WORKERS	53
INDEX (RANGE SCAN)	WORKERS_DIVISION_ID_IX	
Access Predicates		
A.DIVISION_ID=ITEM_1		
TABLE ACCESS (BY INDEX ROWID)	WORKERS	1
Filter Predicates		
A.SALARY>SALAVG		
TABLE ACCESS (FULL)	WORKERS	53

	LAST_NAME	SALARY	DIVISION_ID
1	GAUSS	24000	90
2	BERNOULLI	9000	60
3	GRIEG	12000	100
4	FLAUBERT	9000	100

IN vs. Join I.

```
SELECT last_name, w.division_id, position_id
FROM   workers w, divisions d
WHERE  w.division_id = d.division_id
AND    d.country_id = 'UK';
```

	LAST_NAME	DIVISION_ID	POSITION_ID
1	BIZET	200	IT_DES
2	BACH	70	PR_REP

```
SQL_ID  dztzuyuxhyrva, child number 0
-------------------------------------
SELECT last_name, w.division_id, position_id FROM workers w, divisions
d WHERE w.division_id = d.division_id  and d.country_id = 'UK'

Plan hash value: 1066815395

--------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name                        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                             |       |       |     4 (100)|          |
|*  1 |  HASH JOIN                         |                             |    12 |   372 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| DIVISIONS                 |     4 |    28 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                | DIVISIONS_COUNTRY_ID_IX     |     4 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                | WORKERS                     |    53 |  1272 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------------------------
```

IN vs. Join II.

```
SELECT last_name, division_id, position_id
FROM   workers
WHERE  division_id IN (SELECT division_id
                       FROM   divisions
                       WHERE  country_id = 'UK');
```

	LAST_NAME	DIVISION_ID	POSITION_ID
1	BIZET	200	IT_DES
2	BACH	70	PR_REP

```
SQL_ID  dm9wfg27dvg28, child number 0
-------------------------------------
SELECT last_name, division_id, position_id FROM workers WHERE
division_id IN (SELECT division_id FROM divisions WHERE country_id =
'UK')

Plan hash value: 1066815395

----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                      |       |       |     4 (100)|          |
|*  1 |  HASH JOIN                          |                      |    12 |   372 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| DIVISIONS           |     4 |    28 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | DIVISIONS_COUNTRY_ID_IX |  4 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL                 | WORKERS              |    53 |  1272 |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------
```

Subquery in HAVING clause

What are the "best" and the "worst" positions in the company?

- The best position is where the average salary is the highest.
- The worst position is where the average salary is the lowest

```
SELECT   position_id, ROUND(AVG(salary))
FROM     workers
GROUP BY position_id
HAVING   ROUND(AVG(salary))=
             (SELECT   MAX(ROUND(AVG(salary)))
              FROM     workers
              GROUP BY position_id)
OR       ROUND(AVG(salary))=
             (SELECT MIN(ROUND(AVG(salary)))
              FROM workers
              GROUP BY position_id);
```

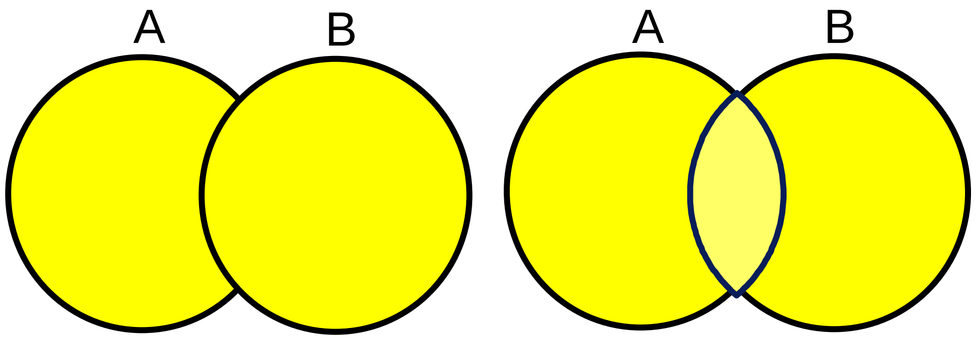
	POSITION_ID	ROUND(AVG(SALARY))
1	ADMIN_PRES	24000
2	STOCK_CLERK	2957

Using the Set Operators

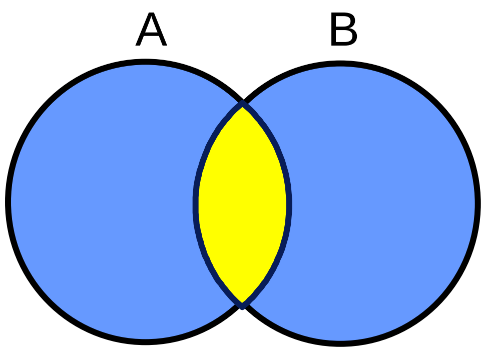
The SET Operators

- You can combine multiple queries using the set operators `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`.
- All set operators have equal precedence.
  If a SQL statement contains multiple set operators, then the database engine evaluates them from the left to right unless parentheses explicitly specify another order.
- The corresponding expressions in the select lists of the component queries of a compound query must match in number and must be in the same data type group (such as numeric or character).

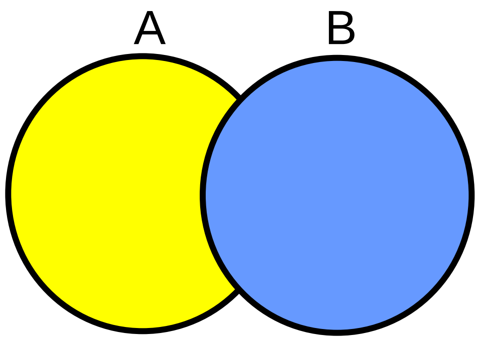
Set Operators



UNION/UNION ALL



INTERSECT



MINUS

Set Operator Guidelines

- The expressions in the `SELECT` lists must match in number.
- The data type of each column in the second query must match the data type of its corresponding column in the first query.
- Parentheses can be used to alter the sequence of execution.
- `ORDER BY` clause can appear only at the very end of the statement.

Using the `UNION ALL` Operator

Display the current and previous position details of all workers.

```
SELECT worker_id, position_id
FROM   workers
UNION ALL
SELECT worker_id, position_id
FROM   position history;
```

	WORKER_ID	POSITION_ID
1	101	ADMIN_VP
2	200	IT_PROG
3	176	SALES_REP
4	101	FINANCE_ACCOUNT
5	101	ACCOUNT_MGR
6	176	SALES_REP
7	176	SALES_MGR
8	200	ADMIN_ASST
9	200	FINANCE_ACCOUNT

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		63
UNION-ALL		
TABLE ACCESS (FULL)	WORKERS	53
TABLE ACCESS (FULL)	POSITION_HISTORY	10

Using the `UNION` Operator

Display the current and previous positions of all workers

Display each worker only once.

```
SELECT worker_id, position_id
FROM   workers
UNION
SELECT worker_id, position_id
FROM   position_history;
```

	WORKER_ID	POSITION_ID
1	101	ACCOUNT_MGR
2	101	ADMIN_VP
3	101	FINANCE_ACCOUNT
4	176	SALES_MGR
5	176	SALES_REP
6	200	ADMIN_ASST
7	200	FINANCE_ACCOUNT
8	200	IT_PROG

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		64	5
SORT (UNIQUE)		64	5
UNION-ALL			
TABLE ACCESS (FULL)	WORKERS	54	2
TABLE ACCESS (FULL)	POSITION_HISTORY	10	3

UNION = UNION ALL + SORT UNIQUE

Using the INTERSECT Operator

Display the worker_ids and position_ids of those workers who currently have the position that is the same as their position when they were initially hired (that is, they changed positions but have now gone back to doing their original position).

```
SELECT worker_id, position_id
FROM   workers
INTERSECT
SELECT worker_id, position_id
FROM   position_history;
```

	WORKER_ID	POSITION_ID
1	176	SALES_REP

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		10
INTERSECTION		
SORT (UNIQUE)		53
TABLE ACCESS (FULL)	WORKERS	53
SORT (UNIQUE)		10
TABLE ACCESS (FULL)	POSITION_HISTORY	10

MINUS Operator

Display the worker IDs of those workers who have never changed their positions

```
SELECT worker_id, position_id
FROM   workers
MINUS
SELECT worker_id, position_id
FROM   position_history;
```

	WORKER_ID	POSITION_ID
1	100	ADMIN_PRES
2	101	ADMIN_VP
3	102	ADMIN_VP
4	103	IT_MGR
5	104	IT_PROG

OPERATION	OBJECT_NAME	CARDINALITY
SELECT STATEMENT		53
MINUS		
SORT (UNIQUE)		53
TABLE ACCESS (FULL)	WORKERS	53
SORT (UNIQUE)		10
TABLE ACCESS (FULL)	POSITION_HISTORY	10

More realistic example

What kind of products were not sold before '06-jan-1998'?

```
SELECT product_id
FROM   products
MINUS
SELECT product_id
FROM   items i, orders o
WHERE  i.order_id = o.order_id
AND    date_ordered < '06-jan-1998';
```

	PRODUCT_ID
1	1052
2	1077
3	1114

Compare two tables

Having two tables with the same structures, what are the differences between them?

```
SELECT * FROM learning.workers l
MINUS
SELECT * FROM student1.workers s
UNION ALL
SELECT * FROM student1.workers
MINUS
SELECT * FROM learning.workers;
```

	WORKER_ID	FIRST_NAME	LAST_NAME	EMAIL	START_DATE	POSITION_ID	SALARY	COMMISSION	MANAGER_ID	DIVISION_ID	BORN
1	115	Oliver	KHAN	OKHAN	18-MAY-1995	PURCHASE_CLERK	3100		114	30	15-JUN-1969
2	116	Arantes	PELE	APELE	24-DEC-1997	PURCHASE_CLERK	2900		114	30	23-OCT-1940
3	117	Silva Ferreira	EUSEBIO	SEUSEBIO	24-JUL-1997	PURCHASE_CLERK	2800		114	30	25-JAN-1942
4	138	Nobby	STILES	NSTILES	26-OCT-1997	STOCK_CLERK	3200		120	160	18-MAY-1942
5	151	Leonard	BERNSTEIN	LBERNSTEIN	24-MAR-1997	SALES_REP	9500	0.25	145	80	25-AUG-1918
6	168	Steven	FEUERSTEIN	SFEUERSTEIN	11-MAR-1997	SALES_REP	11500	0.25	149	80	01-SEP-1958
7	200	Janis Lyn Joplin	JOPLIN	JJOPLIN	17-SEP-1987	ADMIN_ASST	4400		101	10	19-JAN-1943

Set Operator Guidelines

- The expressions in the `SELECT` lists must match in number.
- The data type of each column in the second query must match the data type of its corresponding column in the first query.
- Parentheses can be used to alter the sequence of execution.
- `ORDER BY` clause can appear only at the very end of the statement.

Using parentheses

Using parentheses to change the order of execution os SELECT statements.

```
SELECT salary,division_id  FROM workers WHERE division_id=80;
SELECT salary,division_id  FROM workers WHERE division_id=60;
SELECT salary,division_id   FROM workers WHERE division_id=20;
```

```
SELECT salary FROM workers WHERE division_id=80
UNION
SELECT salary FROM workers WHERE division_id=60
INTERSECT
SELECT salary  FROM workers WHERE division_id=20;
no rows selected
```

```
SELECT salary FROM workers WHERE division_id=80
UNION
(SELECT salary FROM workers WHERE division_id=60
INTERSECT
SELECT salary  FROM workers WHERE division_id=20);
 9 rows selected
```

DML Statements

Data Manipulation Language

- Data manipulation language (DML) statements query or manipulate data in existing schema objects.
- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
  - Merging rows into a table from source tables
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

`INSERT` Statement Syntax

- Use the `INSERT` statement to add rows to a table, the base table of a view, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or the base table of an object view.

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- With this syntax, only one row is inserted at a time.
- List values in the default order of the columns in the table.

```
INSERT INTO divisions (division_id,division_name,
manager_id,city,country_id,parent_id)
VALUES (2,'Head Quarters',
100,'San Francisco','US',NULL);

1 rows inserted.
```

Inserting Rows with Null Values

- Explicit method: specify the columns in the column list.

```
INSERT INTO divisions
(division_id,division_name,manager_id,city)
VALUES (2,'Head Quarters',100,'San Francisco');
1 row created.
```

- Implicit method: Specify the `NULL` keyword in the `VALUES` clause.

```
INSERT INTO   divisions
VALUES   (100, 'Finance', NULL, NULL, NULL, NULL);
1 row created.
```

The INSERT SELECT statement

- Write your `INSERT` statement with a subquery:

```
INSERT INTO preferred_customers
  SELECT *
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
105 row created.
```

- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause to those in the subquery.

```
INSERT INTO preferred_customers
          (customer_id ,customer_name,city)
  SELECT customer_id ,customer_name,city
  FROM   customers
  WHERE  credit_rating in ('GOOD','EXCELLENT');
105 row created.
```

`UPDATE` Statement Syntax

- Modify existing rows with the `UPDATE` statement:
- Use the UPDATE statement to change existing values in a table or in the base table of a view or the master table of a materialized view.

```
UPDATE        table
SET           column = value [, column = value, ...]
[WHERE        condition];
```

```
UPDATE workers
SET    division_id = 70
WHERE  worker_id = 100;
1 row updated.
```

```
UPDATE workers
SET   last_name = UPPER(last_name);
53 rows updated.
```

Updating Two Columns with a Subquery

Update worker 206's job and salary to match that of employee 205.

```
UPDATE   workers
SET    (position_id, salary) =
          (SELECT  position_id, salary
           FROM    workers
           WHERE   worker_id = 205)
WHERE worker_id     =  206;

1 rows updated.
```

OPERATION	OBJECT_NAME	CARDINALITY
UPDATE STATEMENT		1
UPDATE	WORKERS	
INDEX (UNIQUE SCAN)	WORKERS_WORKER_ID_PK	1
Access Predicates		
WORKER_ID=206		
TABLE ACCESS (BY INDEX ROWID)	WORKERS	1
INDEX (UNIQUE SCAN)	WORKERS_WORKER_ID_PK	1
Access Predicates		
WORKER_ID=205		

Correlated UPDATE statement

- A correlated subquery is a SELECT statement nested inside another SQL statement,which contains a reference to one or more columns in the outer query.
- The correlated subquery will be run once for each candidate row selected by the outer SQL statement.
- When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows you are interested in updating.

Set the correct bonus for each customer according their orders

```
UPDATE customers c SET bonus =
 (SELECT ROUND(sum(total_amount)/1000) FROM orders o
  WHERE c.customer_id = o.customer_id);
135 rows updated.
```

OPERATION	OBJECT_NAME	CARDINALITY
UPDATE STATEMENT		135
UPDATE	CUSTOMERS	
TABLE ACCESS (FULL)	CUSTOMERS	135
SORT (AGGREGATE)		1
TABLE ACCESS (BY INDEX ROWID BATCHED)	ORDERS	2603
INDEX (RANGE SCAN)	ORDERS_CUSTOMER_IX	2603
Access Predicates		
O.CUSTOMER_ID=:B1		

SELECT … FOR UPDATE and UPDATE STATEMENTS

- The SELECT FOR UPDATE statement allows you to lock the rows in the result set.
- You are not required to make changes to the records in order to use this statement.
- The record locks are released when the next commit or rollback statement is issued.
- In the first session:

```
update newworker set last_name=last_name;
```

- In the second session:

```
SELECT * FROM newworker FOR UPDATE NOWAIT; -- WAIT <sec>
```

```
Error starting at line : 1 in command -
SELECT * FROM newworker FOR UPDATE NOWAIT
Error report -
SQL Error: ORA-00054: resource busy and acquire with NOWAIT specified or timeout expired
00054. 00000 -  "resource busy and acquire with NOWAIT specified"
```

DELETE Statement

- You can remove existing rows from a table by using the `DELETE` statement:

```
DELETE [FROM]   table
[WHERE          condition];
```

- This DELETE will remove one row

```
DELETE FROM divisions
 WHERE  division_name = 'Contracting';
1 row deleted.
```

- This DELETE will cause an integrity error

```
DELETE FROM divisions
 WHERE  division_name = 'Recruiting';
```

DELETE with a subquery

- You can use a subquery to delete rows with values that also exist in another table.

```
DELETE FROM newworker
WHERE worker_id NOT IN
(SELECT NVL(manager_id,-1)  FROM   workers);
39 row deleted.
```

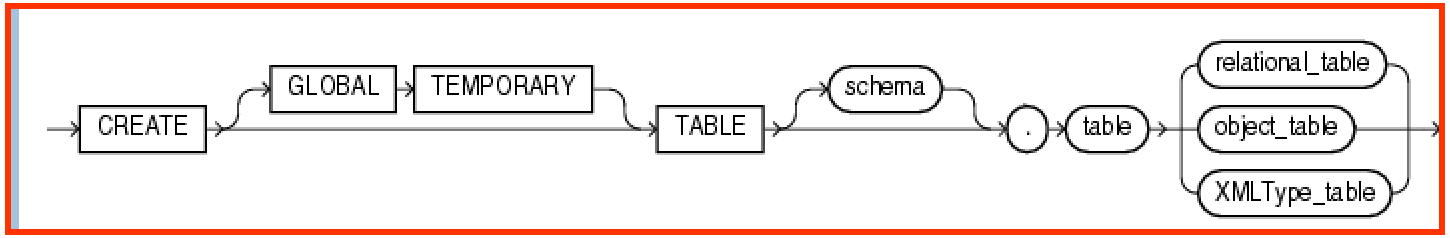
OPERATION	OBJECT_NAME	CARDINALITY	COST
DELETE STATEMENT		39	5
DELETE	NEWWORKER		
HASH JOIN (ANTI SNA)		39	5
Access Predicates			
WORKER_ID=NVL(MANAGER_ID,(-1))			
TABLE ACCESS (FULL)	NEWWORKER	53	3
TABLE ACCESS (FULL)	WORKERS	53	2

Managing Tables

Using DDL Statements to Create Tables

CREATE TABLE Statement

The general syntax:



- You must have:
  - CREATE TABLE privilege
  - A storage area
- You specify:
  - Table name
  - Column name, column data type, and column size or number of valuable characters/bytes

The basic syntax:

```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

Creating Tables

- Create the table.

```
CREATE TABLE countries2
  (country_id   CHAR(2) ,
   country_name VARCHAR2(40),
   area BINARY_FLOAT,
   inhabitants INTEGER,
   map BLOB, history CLOB);
table COUNTRIES2 created.
```

- Confirm table creation.

```
DESCRIBE countries2
```

```
DESC countries2
Name         Null Type
------------ ---- --------------
COUNTRY_ID        CHAR(2)
COUNTRY_NAME      VARCHAR2(40)
AREA              BINARY_FLOAT()
INHABITANTS       NUMBER(38)
MAP               BLOB
HISTORY           CLOB
```

Creating a Table
by Using a Subquery

- Create a table and insert rows by combining the `CREATE TABLE` statement and the AS *subquery* option.
- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

```
CREATE TABLE table [(column, column...)]
AS subquery;
```

```
CREATE TABLE rich_workers AS
SELECT  w.*, salary*12 ANNSAL
FROM  workers w
WHERE   salary>=10000;
```

```
desc rich_workers
Name        Null     Type
----------- -------- ------------
WORKER_ID            NUMBER(6)
FIRST_NAME           VARCHAR2(25)
LAST_NAME   NOT NULL VARCHAR2(25)
EMAIL       NOT NULL VARCHAR2(25)
START_DATE  NOT NULL DATE
POSITION_ID NOT NULL VARCHAR2(15)
SALARY               NUMBER(8,2)
COMMISSION           NUMBER(8,2)
MANAGER_ID           NUMBER(6)
DIVISION_ID          NUMBER(4)
BORN                 DATE
ANNSAL               NUMBER
```

Dropping a Table

- All data and structure in the table are deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- All constraints are dropped.
- You *cannot* roll back the `DROP TABLE` statement.
- Use the `DROP TABLE` statement to move a table or object table to the recycle bin or to remove the table and all its data from the database entirely.
- The basic syntax:

```
DROP TABLE <table_name> [CASCADE CONSTRAINTS] [PURGE];
```

The `ALTER TABLE` Statement

Use the `ALTER TABLE` statement to:

- Add a new column (traditional and virtual)
- Modify an existing column
- Define a default value for the new column
- Drop a column
- Rename a column
- Make table to read only/ read write
- …

The ALTER TABLE Statement

Use the `ALTER TABLE` statement to add, modify, or drop columns.

```
ALTER TABLE table
ADD          (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
MODIFY       (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
DROP         (column);
```

Add columns to the existing table

- Prerequisite:

```
DROP TABLE workers2;
CREATE TABLE workers2 AS SELECT * FROM workers;
```

- Add a new column to an existing table

```
ALTER TABLE workers2 ADD  bonus NUMBER(10) DEFAULT 5000;
table WORKERS2 altered.
```

- Test the result!

```
SELECT worker_id,first_name,last_name,salary,bonus
FROM workers2;
```

Modify column attributes

- Use the `MODIFY` clause to modify table columns
- You can change a column's size, and default value.
- A change to the default value affects only subsequent insertions to the table.
- Generally, you can not modify the base type of the column if the column is not empty (with some exceptions)

```
ALTER TABLE workers2
MODIFY salary VARCHAR2(20);
table WORKERS2 altered.
```

```
Error starting at line : 1 in command -
ALTER TABLE workers2
MODIFY salary VARCHAR2(20)
Error report -
SQL Error: ORA-54033: column to be modified is used in a virtual column expression
```

```
ALTER TABLE workers2
MODIFY (last_name VARCHAR2(30),start_date TIMESTAMP);
table WORKERS2 altered.
```

Removing columns from the table I.

You can mark a column as unused (logical drop) or delete it completely (physical drop).

Logical drop

- On large tables the process of physically removing a column can be very time and resource consuming.
- First, you may decide to logically drop it.

```
ALTER TABLE table_name SET UNUSED(col_name);
ALTER TABLE table_name SET UNUSED(col_name1,col_name2);
```

- Once this is done the columns will no longer be visible to the user.

```
ALTER TABLE workers2 SET UNUSED(ann_sal,bonus);
```

- Later, you can physically remove all unused columns

```
ALTER TABLE table_name DROP UNUSED COLUMNS;
```

Removing columns from the table II.

Physical drop

There two syntaxes for this purpose:

```
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE table_name DROP (column_name1, column_name2);
```

Example:

```
ALTER TABLE workers2 DROP COLUMN born;
ALTER TABLE workers2 DROP (bonus,email);
```

```
desc workers2
Name        Null     Type
----------- -------- ------------
WORKER_ID            NUMBER(6)
FIRST_NAME           VARCHAR2(25)
LAST_NAME   NOT NULL VARCHAR2(30)
START_DATE  NOT NULL TIMESTAMP(6)
POSITION_ID NOT NULL VARCHAR2(15)
SALARY               NUMBER(8,2)
COMMISSION           NUMBER(8,2)
MANAGER_ID           NUMBER(6)
DIVISION_ID          NUMBER(4)
```

Rename tables and columns

Use the RENAME statement to rename a table, view, sequence, or private synonym.

- Databases automatically transfer integrity constraints, indexes, and grants on the old object to the new object.
- The database engines invalidate all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

```
RENAME <old_table_name> TO <new_table_name>;
```

```
RENAME  workers2 TO old_workers2;
```

You can rename a column with following syntax:

```
ALTER TABLE <table_name>
RENAME COLUMN <old_col_name> TO <new_col_name>;
```

```
ALTER TABLE  old_workers2 RENAME COLUMN commission TO comm;
```

Managing Constraints

Constraint Guidelines

- The integrity constraints enforce business rules and prevent the entry of invalid information into tables.
- An integrity constraint is a schema object that is created and dropped using SQL.
- Constraints enforce rules at the table level.
- Advantages of integrity constraints:
  - Declarative ease: Created with SQL statements, no additional programming is required
  - Centralized rules: Integrity constraints are defined for tables and are stored in the data dictionary, all applications must adhere to the same integrity constraints.
  - Flexibility when loading data:
    You can disable integrity constraints temporarily to avoid performance overhead when loading large amounts of data.
- Define a constraint at the column or table level when :
  - Table is created
  - After the table has been created

Types of Constraints

- Constraints may be defined both at the table and column level.
- A constraint specified as part of the definition of a column or attribute is an inline specification.
- A key is the column or set of columns included in the definition of certain types of integrity constraints.
- Keys describe the relationships between the tables and columns of a relational database. Individual values in a key are called key values.

Constraint Type	Description	See Also
NOT NULL	Allows or disallows inserts or updates of rows containing a **null** in a specified column.	"NOT NULL Integrity Constraints"
Unique key	Prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.	"Unique Constraints"
Primary key	Combines a NOT NULL constraint and a unique constraint. It prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.	"Primary Key Constraints"
Foreign key	Designates a column as the foreign key and establishes a relationship between the foreign key and a primary or unique key, called the **referenced key**.	"Foreign Key Constraints"
Check	Requires a database value to obey a specified condition.	"Check Constraints"
REF	Dictates types of data manipulation allowed on values in a REF column and how these actions affect dependent values. In an object-relational database, a built-in data type called a REF encapsulates a reference to a row object of a specified object type. Referential integrity constraints on REF columns ensure that there is a row object for the REF.	*Oracle Database Object-Relational Developer's Guide* to learn about REF constraints

Defining Constraints

- Column-level constraint:

```
CREATE TABLE col_level_constraints(
  worker_id  NUMBER(6)
        constraint workers_worker_id_pk PRIMARY KEY,
  last_name   VARCHAR2(20) NOT NULL);
 table COL_LEVEL_CONSTRAINTS created.
```

- Table-level constraint:

```
CREATE TABLE table_level_constraints(
 worker_id    NUMBER(6),
 last_name    VARCHAR2(20),
 position_id  VARCHAR2(10) NOT NULL,-- column level only
 CONSTRAINT   table_level_cons_worker_id_pk
              PRIMARY KEY (worker_id));
 table TABLE_LEVEL_CONSTRAINTS created.
```

NOT NULL Constraint

- Ensures that null values are not permitted for the column
- A null is the absence of a value. By default, all columns in a table allow nulls.

```
SELECT worker_id,last_name,email,position_id,
start_date,salary,commission
FROM workers;
```

	WORKER_ID	LAST_NAME	EMAIL	POSITION_ID	START_DATE	SALARY	COMMISSION
1	207	HEMINGWAY	EHEMINGWAY	SALES_REP	07-JUN-2007	8300	0.5
2	100	GAUSS	CGAUSS	ADMIN_PRES	17-JUN-1987	24000	
3	101	EULER	LEULER	ADMIN_VP	21-SEP-1989	17000	
4	102	BERNOULLI	JBERNOULLI	ADMIN_VP	13-JAN-1993	17000	
5	103	BERNOULLI	DBERNOULLI	IT_MGR	03-JAN-1990	9000	
6	104	WILLIS	BWILLIS	IT_PROG	21-MAY-1991	6000	

NOT NULL constraint (No row can contain a null value for this column.)

NOT NULL constraint

Absence of NOT NULL constraint (Any row can contain a null value for this column.)

UNIQUE Constraint

- A unique key constraint requires that every value in a column or set of columns be unique or NULL.
- No rows of a table may have duplicate values in a single column (the unique key) or set of columns (the composite unique key) with a unique key constraint.
- Defined at either the table level or the column level:

UNIQUE constraint

WORKERS

	WORKER_ID	LAST_NAME	EMAIL	POSITION_ID	START_DATE	SALARY	COMMISSION
1	207	HEMINGWAY	EHEMINGWAY	SALES_REP	07-JUN-2007	8300	0.5
2	100	GAUSS	CGAUSS	ADMIN_PRES	17-JUN-1987	24000	
3	101	EULER	LEULER	ADMIN_VP	21-SEP-1989	17000	
4	102	BERNOULLI	JBERNOULLI	ADMIN_VP	13-JAN-1993	17000	
5	103	BERNOULLI	DBERNOULLI	IT_MGR	03-JAN-1990	9000	
6	104	WILLIS	BWILLIS	IT_PROG	21-MAY-1991	6000	

```
INSERT INTO  workers(worker_id,last_name,    start_date)
              VALUES(999,       'Impossible', sysdate);
```

UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE LEARNING.WORKERS
  ( WORKER_ID    NUMBER(6,0),   FIRST_NAME   VARCHAR2(25),
    LAST_NAME    VARCHAR2(25),  EMAIL        VARCHAR2(25),
    START_DATE   DATE ,         POSITION_ID  VARCHAR2(15),
    SALARY       NUMBER(8,2),   COMMISSION   NUMBER(8,2),
    MANAGER_ID   NUMBER(6,0),   DIVISION_ID  NUMBER(4,0),
    BORN         DATE,
    CONSTRAINT WORKER_EMAIL_UK UNIQUE (EMAIL)
…
```

PRIMARY KEY Constraint

- In a PRIMARY KEY constraint, the values in the group of one or more columns subject to the constraint uniquely identify the row.
- Each table can have one PRIMARY KEY that can not be NULL
- Defined at either the table level or the column level:

PRIMARY KEY

	WORKER_ID	LAST_NAME	EMAIL	POSITION_ID	START_DATE	SALARY	COMMISSION
1	207	HEMINGWAY	EHEMINGWAY	SALES_REP	07-JUN-2007	8300	0.5
2	100	GAUSS	CGAUSS	ADMIN_PRES	17-JUN-1987	24000	
3	101	EULER	LEULER	ADMIN_VP	21-SEP-1989	17000	
4	102	BERNOULLI	JBERNOULLI	ADMIN_VP	13-JAN-1993	17000	
5	103	BERNOULLI	DBERNOULLI	IT_MGR	03-JAN-1990	9000	
6	104	WILLIS	BWILLIS	IT_PROG	21-MAY-1991	6000	

```
INSERT INTO workers (worker_id, last_name,start_date,
                     email,position_id)
   VALUES(105, 'Impossible',SYSDATE,'Something','IT_PROG');
```

```
INSERT INTO  workers (worker_id,last_name,start_date,
                      email,position_id)
   VALUES(106,'Impossible',SYSDATE,'Something','IT');
```

```
SQL Error: ORA-00001: unique constraint (LEARNING.WORKERS_WORKER_ID_PK) violated
00001. 00000 -  "unique constraint (%s.%s) violated"
```

FOREIGN KEY Constraint

- Whenever two tables contain one or more common columns, the database can enforce the relationship between the two tables through a FOREIGN KEY constraint, also called a referential integrity constraint.
- The constraint requires that for each value in the column on which the constraint is defined, the value in the other specified other table and column must match.

FOREIGN KEY

	WORKER_ID	LAST_NAME	EMAIL	POSITION_ID	START_DATE	SALARY	COMMISSION
1	207	HEMINGWAY	EHEMINGWAY	SALES_REP	07-JUN-2007	8300	0.5
2	100	GAUSS	CGAUSS	ADMIN_PRES	17-JUN-1987	24000	
3	101	EULER	LEULER	ADMIN_VP	21-SEP-1989	17000	
4	102	BERNOULLI	JBERNOULLI	ADMIN_VP	13-JAN-1993	17000	
5	103	BERNOULLI	DBERNOULLI	IT_MGR	03-JAN-1990	9000	
6	104	WILLIS	BWILLIS	IT_PROG	21-MAY-1991	6000	

REFERENCES

	POSITION_ID	POSITION_TITLE	LOWEST_SALARY	HIGHEST_SALARY
1	IT_MGR	IT Manager	8000	15000
2	ADMIN_PRES	President	20000	35000
3	ADMIN_VP	Administration Vice President	15000	25000
4	ADMIN_ASST	Administration Assistant	3000	6000
5	FINANCE_MGR	Finance Manager	8200	16000
6	FINANCE_ACCOUNT	Accountant	4200	9000
7	ACCOUNT_MGR	Accounting Manager	8200	16000
8	PUBLIC_ACCOUNT	Public Accountant	4200	9000

`FOREIGN KEY` Constraint

Defined at either the table level or the column level:

```
CREATE TABLE LEARNING.WORKERS
  ( worker_id   NUMBER(6,0) , first_name  VARCHAR2(25),
    last_name   VARCHAR2(25), email       VARCHAR2(25) ,
    start_date  DATE        , position_id VARCHAR2(15),
    salary      NUMBER(8,2) , commission  NUMBER(8,2),
    manager_id  NUMBER(6,0) , division_id NUMBER(4,0),
    born        DATE,
 CONSTRAINT workers_worker_id_pk PRIMARY KEY (worker_id),
 CONSTRAINT workers_position_fk  FOREIGN KEY  (position_id)
 REFERENCES learning.positions   (position_id) ,
 CONSTRAINT workers_manager_fk   FOREIGN KEY (manager_id)
 REFERENCES learning.workers     (worker_id) ,
 CONSTRAINT workers_division_fk  FOREIGN KEY (division_id)
 REFERENCES learning.divisions   (division_id)
…
```

`FOREIGN KEY` Constraint: Keywords

- `FOREIGN KEY`: Defines the column in the child table at the table-constraint level
- `REFERENCES`: Identifies the table and column in the parent table
- `ON DELETE CASCADE`: Deletes the dependent rows in the child table when a row in the parent table is deleted
- `ON DELETE SET NULL`: Converts dependent FOREIGN KEY values to null

`CHECK` Constraint

- Defines a condition that each row must satisfy

```
...,  salary   NUMBER(2)
      CONSTRAINT WORKER_salary_min
             CHECK (salary > 0),...
```

`CREATE TABLE`: Example

```
CREATE TABLE workers  (  worker_id   NUMBER(6,0),
  first_name  VARCHAR2(25),
  last_name   VARCHAR2(25)CONSTRAINT workers_last_name_nn not,
  email       VARCHAR2(25)CONSTRAINT workers_email_nn NOT NULL,
  start_date  DATE CONSTRAINT workers_start_date_nn NOT NULL,
  position_id VARCHAR2(15)CONSTRAINT workers_position_nn NOT NULL,
  salary      NUMBER(8,2),
  commission  NUMBER(8,2),
  manager_id  NUMBER(6,0),
  division_id NUMBER(4,0),
  born        DATE,
   CONSTRAINT worker_salary_min CHECK (salary > 0 ENABLE,
   CONSTRAINT worker_email_uk UNIQUE (email),
   CONSTRAINT workers_worker_id_pk PRIMARY KEY (worker_id),
   CONSTRAINT workers_position_fk FOREIGN KEY (position_id)
    REFERNCES positions (position_id) ENABLE,
   CONSTRAINT workers_manager_fk FOREIGN KEY (manager_id)
    REFERNCES workers (worker_id) ENABLE,
   CONSTRAINT workers_division_fk FOREIGN KEY (division_id)
    REFERNCES divisions (division_id) ENABLE );
```

Adding a Constraint Syntax

Use the `ALTER TABLE` statement to:

- Add or drop a constraint
- Enable or disable constraints
- Add a `NOT NULL` constraint by using the `MODIFY` clause

```
ALTER TABLE  <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```

Adding a Constraint

There are two types of syntax:

- Adding a new column with suitable constraints following the column level syntax

```
ALTER TABLE newworker ADD  division_id NUMBER(4)
CONSTRAINT newworkers_division_fk
REFERENCES DIVISIONS (division_id);
Table altered.
```

- Adding suitable constraints following the table level syntax:

```
ALTER TABLE newworker ADD CONSTRAINT
newworkers_worker_id_pk PRIMARY KEY (worker_id) ;
Table altered.
ALTER TABLE newworker ADD
CONSTRAINT newworker_manager_fk FOREIGN KEY(manager_id)
    REFERENCES newworker (worker_id);
Table altered.
```

More complex CHECK constraints

- You can implement more complex business logic with `CHECK` constraint
- Example: You can assign commission for sales people only.

```
ALTER TABLE workers
ADD CONSTRAINT worker_comm_ck
CHECK(DECODE(
SUBSTR(position_id,1,5),'SALES',NULL,commission) IS NULL);
Table altered.
```

```
UPDATE workers SET commission = 0.2
WHERE position_id like 'SALES%';
14 rows updated.
```

```
UPDATE workers SET commission = 0.2
WHERE position_id like 'ADMIN%';
```

Ez a kép most nem jeleníthető meg.

Removing Constraint

- By default you can not drop a `constraint` that is referred by an other one.
- You must use the `CASCADE` clause

```
ALTER TABLE newworker DROP CONSTRAINT
newworkers_worker_id_pk;
```

Ez a kép most nem jeleníthető meg.

```
ALTER TABLE newworker DROP CONSTRAINT
newworkers_worker_id_pk cascade;
table NEWWORKER altered.
```

```
ALTER TABLE newworker ADD
CONSTRAINT NEWWORKER_MANAGER_FK FOREIGN KEY (MANAGER_ID)
REFERENCES newWORKER (WORKER_ID);
```

```
SQL Error: ORA-02270: no matching unique or primary key for this column-list
02270. 00000 -  "no matching unique or primary key for this column-list"
*Cause:    A REFERENCES clause in a CREATE/ALTER TABLE statement
           gives a column-list for which there is no matching unique or primary
           key constraint in the referenced table.
```

ON DELETE CASCADE

Delete child rows when a parent key is deleted.

```
ALTER TABLE newworker ADD
CONSTRAINT newworker_position_fk
FOREIGN KEY (position_id)
REFERENCES positions (position_id)
ON DELETE CASCADE;
Table altered.
```

Disabling Constraints

- Execute the `DISABLE` clause of the `ALTER TABLE` statement to deactivate an integrity constraint.
- Apply the `CASCADE` option to disable dependent integrity constraints.

```
ALTER TABLE workers
DISABLE CONSTRAINT worker_comm_ck;
Table altered.
```

Cascading Constraints

Example:

A `UNIQUE` index is automatically created if you enable a `UNIQUE` key or `PRIMARY KEY` constraint.

```
ALTER TABLE workers
DROP COLUMN worker_id CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (primary_keyk, foregin_key, col1)
CASCADE CONSTRAINTS;
Table altered.
```

Violating Constraints

```
DELETE FROM customers WHERE country_id='DE';
```

```
DELETE FROM customers
WHERE country_id='DE'
Error report -
SQL Error: ORA-02292: integrity constraint (LEARNING.ORDERS_CUSTOMER_ID_FK) violated - child record found
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
```

```
UPDATE customers SET credit_rating='VERY GOOD'
WHERE country_id='PL';
```

```
UPDATE customers SET credit_rating='VERY GOOD'
WHERE country_id='PL'
Error report -
SQL Error: ORA-02290: check constraint (LEARNING.CUSTOMER_CREDIT_RATING_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

Views and Indexes

Some Database Object Types

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of data retrieval queries
Synonym	Gives alternative names to objects

What Is a View?

`workers` table

	WORKER_ID	FIRST_NAME	LAST_NAME	POSITION_ID	SALARY	COMMISSION	DIVISION_ID
1	207	Ernest Miller	HEMINGWAY	SALES_REP	8300	0.5	210
2	100	Carl Friedrich	GAUSS	ADMIN_PRES	24000	0.2	90
3	101	Leonard	EULER	ADMIN_VP	17000	0.2	90
4	102	Johann	BERNOULLI	ADMIN_VP	17000	0.2	90
5	103	Daniel	BERNOULLI	IT_MGR	9000		60
6	104	Bruce	WILLIS	IT_PROG	6000		60
7	106	Giuseppe	VERDI	IT_PROG	4800		60
8	107	Hendrik	LORENTZ	IT_PROG	4200		60
9	108	Edvard	[illegible]	[illegible]	[illegible]		100
10	109	Gustave					100
11	110	John					100
12	111	[illegible]					100
13	112						100
14	[illegible]						30
15							30
16							30
17					[illegible]		30
18					8000		160

	WORKER_ID	LAST_NAME	SALARY	DIVISION_ID
1	103	BERNOULLI	9000	60
2	104	WILLIS	6000	60
3	106	VERDI	4800	60
4	107	LORENTZ	4200	60

Overview of Views

- A view is a logical representation of one or more tables. In essence, a view is a stored query.
- A view derives its data from the tables on which it is based, called base tables.
- Base tables can be tables or other views.
- All operations performed on a view actually affect the base tables.
- You can use views in most places where tables are used, but not everywhere!

Benefits of Views

Views enable you to tailor the presentation of data to different types of users.

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table
- Hide data complexity
- Present the data in a different perspective from that of the base table
- Isolate applications from changes in definitions of base tables

Basic syntax:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

- Specify OR REPLACE to re-create the view if it already exists.
- You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.

Creating a View

- Create the `worker_view` view, which contains details of the workers in division 60:

```
CREATE OR REPLACE VIEW worker_view
 AS
 SELECT WORKER_ID,FIRST_NAME ,LAST_NAME,POSITION_ID,SALARY,
       COMMISSION ,DIVISION_ID
 FROM workers ;
view WORKER_VIEW created.
SELECT * FROM worker_view WHERE salary>=6000;
```

	WORKER_ID	FIRST_NAME	LAST_NAME	POSITION_ID	SALARY	COMMISSION	DIVISION_ID
1	103	Daniel	BERNOULLI	IT_MGR	9000		60
2	104	Bruce	WILLIS	IT_PROG	6000		60

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2	2
TABLE ACCESS (BY INDEX ROWID BATCHED)	WORKERS	2	2
Filter Predicates			
SALARY>=6000			
INDEX (RANGE SCAN)	WORKERS_DIVISION_ID_IX	4	1
Access Predicates			
DIVISION_ID=60			

Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

Creating a Complex View

Create a complex view that contains group functions to display values from two tables:

```
CREATE OR REPLACE VIEW cust_orders
     (name, city, credit_rating,
      total_amount, average, count_of_orders)
AS SELECT   c.customer_name, c.city,c.credit_rating,
        TO_CHAR(SUM(o.total_amount),'99,999,999.99'),
        ROUND(AVG(o.total_amount),2),count(*)
   FROM     customers c JOIN orders o
   USING    (customer_id)
   GROUP BY c.customer_name,city,c.credit_rating;

view CUST_ORDERS created.
```

Rules for Performing DML Operations on a View

- You can usually perform DML operations on simple views.

- You cannot delete a row from the view if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - Some other elements

Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- Expressions
- …

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- Columns defined by expressions
- `NOT NULL` columns without default value in the base tables that are not selected by the view

Example: DML via view I.

Let’s create the following simple view:

```
CREATE OR REPLACE VIEW rich_people
        (worker_no,worker_name,salary)
 AS
 SELECT worker_id,last_name,salary
 FROM   workers  WHERE salary>10000;
```

Display the data that is behind the view:

```
SELECT * FROM  rich_people;
```

	WORKER_NO	WORKER_NAME	SALARY
1	100	GAUSS	24000
2	101	EULER	17000
3	102	BERNOULLI	17000
4	108	GRIEG	12000
5	114	GURION	11000
6	145	RUSSELL	14000
7	147	NERUDA	12000
8	149	GROSICS	10500
9	168	FEUERSTEIN	11500
10	174	ABEL	11000
11	201	COANDA	13000
12	205	BERING	12000

Example: DML via view II.

Modify the worker's salary via view:

```
UPDATE rich_people SET salary=salary/2;
12 rows updated.
```

```
SELECT * FROM  rich_people;
```

```
WORKER_NO WORKER_NAME                    SALARY
---------- ------------------------- ----------
       100 GAUSS                          12000
```

The UPDATE modified the data in the base table and can not be seen from view's point of view!

Issue a ROLLBACK!

```
ROLLBACK;
rollback complete.
```

Using the `WITH CHECK OPTION` Clause

- Specify WITH CHECK OPTION to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery.
- When used in the subquery of a DML statement, you can specify this clause in a subquery in the FROM clause but not in subquery in the WHERE clause.
- Any attempt to `INSERT` a row or t`UPDATE` rows will fail that violates the rules which are implemented in the `WITH CHECK OPTION`.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

Using the WITH CHECK OPTION

Reorganize the view using WITH CHECK OPTION

```
 CREATE OR REPLACE VIEW rich_people
(worker_no,worker_name,salary)
  AS
 SELECT worker_id,last_name,salary
 FROM workers
 WHERE salary>10000
 WITH CHECK OPTION CONSTRAINT  rich_emp_ck;
```

```
UPDATE rich_people SET salary=salary/2;
```

```
UPDATE rich_people SET salary=salary/2
Error report -
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 -  "view WITH CHECK OPTION where-clause violation"
```

But!

```
UPDATE rich_people SET salary=salary-499;
12 rows updated.
```

Case study:Using DML via complex view I.

Create a complex view that contains two tables:

```
CREATE OR REPLACE VIEW worker_view AS
SELECT worker_id,last_name,salary,d.division_id,division_name
FROM workers w, divisions d
WHERE w.division_id=d.division_id;
view WORKER_VIEW created.
```

Modify the worker's name via view:

```
UPDATE worker_view SET LAST_NAME=INITCAP(last_name);
52 rows updated.
SELECT * FROM worker_view;
```

	WORKER_ID	LAST_NAME	SALARY	DIVISION_ID	DIVISION_NAME
1	200	Joplin	4400	10	Administration
2	201	Coanda	12501	20	Marketing
3	116	Pele	3190	30	Purchasing
4	117	Eusebio	3080	30	Purchasing
5	115	Khan	3410	30	Purchasing
6	114	Gurion	10501	30	Purchasing
7	203	Bartók	6500	40	Human Resources

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW worker_view;
view WORKER_VIEW dropped.
```

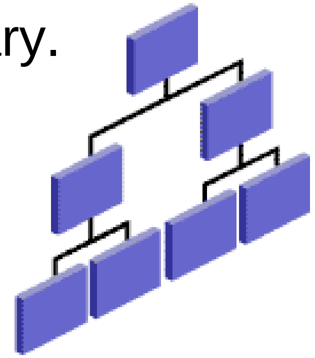
Indexes

Indexes

An index:

- Is a schema object that is an optional structure, associated with a table
- Can contain one or more columns of a table
- Can be used by the database server to speed up the retrieval of rows by using a pointer
- Can reduce disk input/output (I/O) by using a rapid path access method to locate data quickly
- Is dependent on the table that it indexes
- Is used and maintained automatically by the Oracle Server

You can reorgainize an index manualy, if necessary.

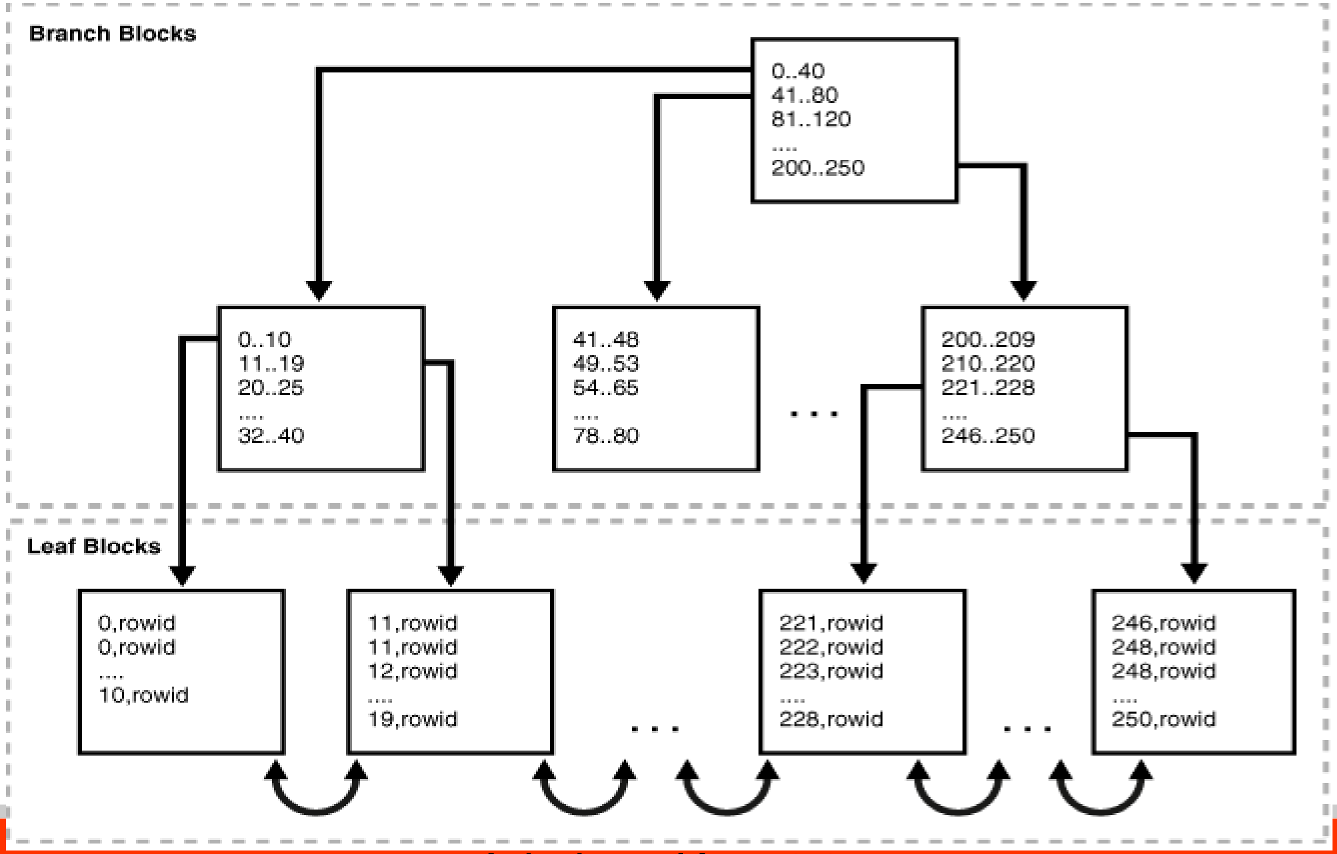


How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.
- Manually: You can create unique or nonunique index on columns to speed up access to the rows.
- Both of them can be non composite or composite index
- A composite index, also called a concatenated index, is an index on multiple columns in a table.

B-Tree Indexes

- A B-tree index is an ordered list of values divided into ranges.
- By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches
- Internal Structure of a B-tree Index



Creating an Index

- Create an index on one or more columns:

```
CREATE [UNIQUE]INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the `customer_name` column in the `customers` table:

```
CREATE INDEX customer_name_ix
ON customers (customer_name);
index CUSTOMER_NAME_IX created.
```

- Create a composite index on `workers table`

```
CREATE INDEX workers_name_ix
ON workers  (last_name,first_name);
index WORKERS_NAME_IX created.
```

Index usage

Using indexes in SELECT statements

```
SELECT * FROM customers
WHERE customer_name LIKE 'Bill%';
```

	CUSTOMER_ID	CUSTOMER_NAME	ADDRESS	CITY	COUNTRY_ID	CREDIT_RATING
1	1008	Bill Johnson's Big Apple	4411 Mercury St	New York	US	POOR
2	1089	Billy's Hickory-Pit Bar-B-Q	Czyzyny	Krakow	PL	GOOD
3	1075	Billy's On Clifton	Makowska	Warsaw	PL	GOOD

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	2
TABLE ACCESS (BY INDEX ROWID BATCHED)	CUSTOMERS	1	2
INDEX (RANGE SCAN)	CUSTOMER_NAME_IX	1	1
Access Predicates			
CUSTOMER_NAME LIKE 'Bill%'			
Filter Predicates			
CUSTOMER_NAME LIKE 'Bill%'			

CREATE INDEX with the CREATE TABLE Statement

```
CREATE TABLE worker_indexes
(worker_id NUMBER(6)
PRIMARY KEY USING INDEX
  (CREATE INDEX worker_id_idx ON
worker_indexes(worker_id)),
first_name  VARCHAR2(20),
last_name   VARCHAR2(25) );
table WORKER_INDEXES created.
```

```
CREATE TABLE cd(country_id INT,division_id INT,
CONSTRAINT country_division_uk
UNIQUE (country_id, division_id)
USING INDEX (CREATE UNIQUE INDEX country_div_ix
ON cd(country_id, division_id)),
CONSTRAINT division_country_uk
UNIQUE (division_id, country_id)
USING INDEX country_div_ix);
table CD created.
```

Function-Based Indexes

- A function-based index is based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.
- Any user-defined function referenced in column_expression must be declared as DETERMINISTIC.

```
CREATE INDEX city_name_ix
ON countries(UPPER(capital_city));
index CITY_NAME_IX created.
```

```
SELECT * FROM countries
WHERE UPPER(capital_city) LIKE 'BU%';
```

	COUNTRY_ID	COUNTRY_NAME	CAPITAL_CITY	CONTINENT_ID
1	RO	Romania	Bucharest	1
2	HU	Hungary	Budapest	1
3	AR	Argentina	Buenos Aires	2

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	2
TABLE ACCESS (BY INDEX ROWID BATCHED)	COUNTRIES	1	2
INDEX (RANGE SCAN)	CITY_NAME_IX	1	1
Access Predicates			
UPPER(CAPITAL_CITY) LIKE 'BU%'			
Filter Predicates			
UPPER(CAPITAL_CITY) LIKE 'BU%'			

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

- Remove the `emp_last_name_idx` index from the data dictionary:

```
DROP INDEX worker_id_ix;
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

Sequences

What are sequences

A sequence:

- Can automatically generate unique numbers
- Is a shareable object
- Can be used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

CREATE SEQUENCE Statement: Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE [ schema. ] sequence
  [ { START WITH|INCREMENT BY } integer
  | { MAXVALUE integer | NOMAXVALUE }
  | { MINVALUE integer | NOMINVALUE }
  | { CYCLE | NOCYCLE }
  | { CACHE integer | NOCACHE }
  | { ORDER | NOORDER }
];
```

Creating a Sequence

- Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.
- Do not use the `CYCLE` option.

```
CREATE SEQUENCE workers_seq MINVALUE 1 MAXVALUE 9999
   INCREMENT BY 1
   START WITH 10
   CACHE 20
   NOCYCLE;
sequence WORKERS_SEQ created.
```

`NEXTVAL` and `CURRVAL` Pseudocolumns

- `NEXTVAL` returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- `CURRVAL` obtains the current sequence value.
- `NEXTVAL` must be issued for that sequence before `CURRVAL` contains a value.

Using a Sequence

- Use the sequence that was previously created

```
INSERT INTO workers
  (worker_id,last_name,email,start_date,
   position_id,salary,manager_id,division_id,born)
VALUES (workers_seq.nextval,'gauss','cgauss2',
   to_date('17-jun-1987','dd-mon-rrrr'),'IT_PROG',24000,
   null,90,to_date('30-apr-1777','dd-mon-rrrr'));
1 rows inserted.
```

```
SELECT   workers_seq.CURRVAL FROM dual;
 CURRVAL
----------
        10
```

Caching Sequence Values

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table

Synonyms and other object types

Synonyms

A synonym:

- Is a database object
- Can be created to give an alternative name to a
  - Table,view or some other database object (e.g. procedure, …)
- Requires no storage other than its definition in the data dictionary
- Is useful for hiding the identity and location of an underlying schema object
- A synonym places a dependency on its target object and becomes invalid if the target object is changed or dropped.
- Synonyms are not a solution for data protection and security
- You can refer to synonyms in the following DML statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE` and `LOCK TABLE`.
- You can refer to synonyms in the following DDL statements: `AUDIT`, `NOAUDIT`, `GRANT`, `REVOKE`, `and COMMENT`.

Creating a Synonym for an Object

- Simplify access to objects by creating a synonym
- Create an easier reference to a table that is owned by another user
- Shorten lengthy object names
- To create a private synonym in your own schema, you must have the CREATE SYNONYM system privilege.

```
CREATE SYNONYM synonym FOR object;
```

```
CREATE SYNONYM cust FOR customers;
synonym CUST created.
SELECT * FROM cust;
```

	CUSTOMER_ID	CUSTOMER_NAME	ADDRESS	CITY
1	1064	Paulette's Coffee Shop	Pointe Saint-Charles	Montreal
2	1065	Bob's On Sheridan	Kloveniersburgwal	Amsterdam

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		135	3
TABLE ACCESS (FULL)	CUSTOMERS	135	3

Listing and Removing Synonyms

- To check the existence of the synonyms:

```
SELECT * FROM user_synonyms;
```

	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	ORIGIN_CON_ID
1	CUST	LEARNING	CUSTOMERS		0

- Drop a synonym:

```
DROP SYNONYM cust;
synonym CUST dropped.
```

```
SELECT * FROM user_synonyms;
no rows selected
```

Create Public synonym

- To create a PUBLIC synonym, you must have the CREATE PUBLIC SYNONYM system privilege.
- Public synonym are used mostly for public object,
- Name Resolution:
  - First the private synonym is used if it exists
  - Public synonym are used if private object (table, viwe, synonym) does not exist with this name

```
CREATE PUBLIC SYNONYM synonym
FOR    object;
```

```
GRANT SELECT ON  customers TO PUBLIC;
GRANT succeeded.
```

```
CREATE PUBLIC SYNONYM cust
FOR    customers;
public synonym CUST created.
```

Synonym Information

```
DESCRIBE user_synonyms
```

```
DESCRIBE user_synonyms
Name           Null     Type
-------------- -------- -------------
SYNONYM_NAME   NOT NULL VARCHAR2(128)
TABLE_OWNER             VARCHAR2(128)
TABLE_NAME     NOT NULL VARCHAR2(128)
DB_LINK                 VARCHAR2(128)
ORIGIN_CON_ID           NUMBER
```

```
SELECT * FROM   user_synonyms;
```

	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	ORIGIN_CON_ID
1	CUST	LEARNING	CUSTOMERS		0

```
SELECT * FROM all_synonyms WHERE table_owner='LEARNING';
```

	OWNER	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	ORIGIN_CON_ID
1	LEARNING	CUST	LEARNING	CUSTOMERS		0
2	PUBLIC	COUNTRIES	LEARNING	COUNTRIES		0
3	PUBLIC	CUST	LEARNING	CUSTOMERS		0
4	PUBLIC	ITEMS	LEARNING	ITEMS		0
5	PUBLIC	ORDERS	LEARNING	ORDERS		0
6	PUBLIC	WORKERS	LEARNING	NEWWORKERS		0

Many other objects…

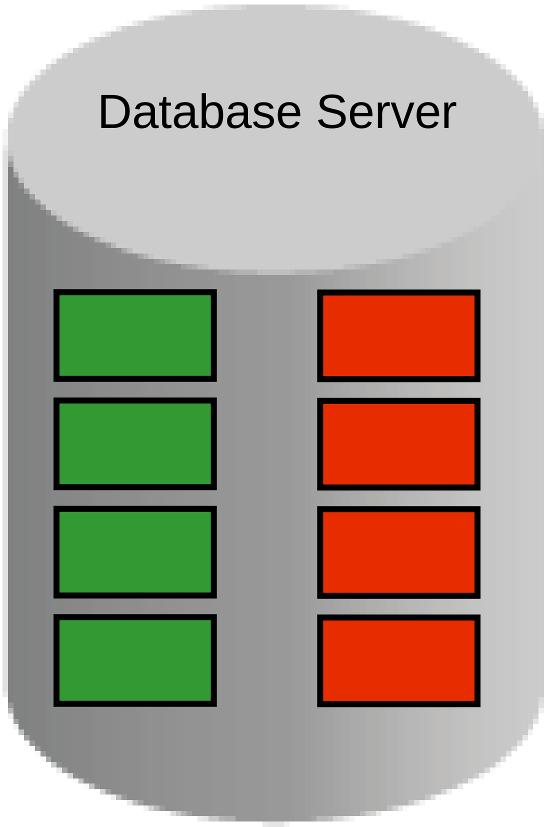
- Depending on the database system many other object types may exist:
  - Stored procedures
  - Functions
  - Packages
  - Packagy Bodies
  - Materialized Views
  - …
  - …

The Data Dictionary

Data Dictionary

Tables containing business data:

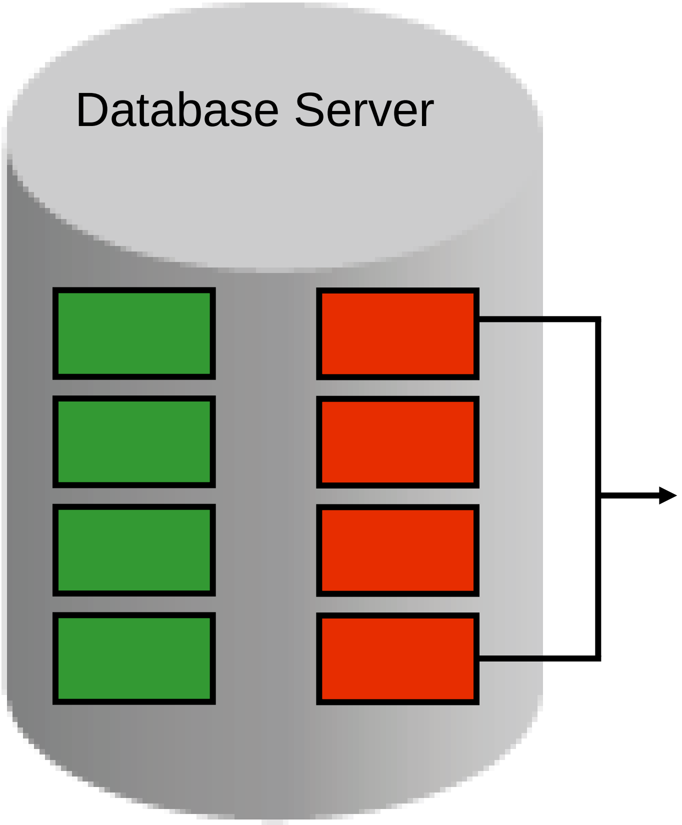
WORKERS
DIVISIONS
COUNTRIES
CUSRTOMERS
. . .



Data dictionary tables and views:

DICTIONARY
USER_OBJECTS
USER_TABLES
USER_TAB_COLUMNS
. . .

Data Dictionary Structure



Consists of:
- Base tables (in case or Oracle: TAB$, IND$ and so on)
- User-accessible views (in Oracle USER_TABLES, …

Data Dictionary Structure

View naming convention:

View Prefix	Purpose
USER	User’s view (what is in your schema; what you own)
ALL	Expanded user’s view (what you can access)
DBA	Database administrator’s view (what is in everyone’s schemas)
V$	Performance-related data – strictly speaking it is not Data Dictionary, but dynamic performance tables

How to Use the Dictionary Views

Start with `DICTIONARY` or similar. It contains the names and descriptions of the dictionary tables and views.

```
desc dictionary
```

```
desc dictionary
Name       Null Type
---------- ---- --------------
TABLE_NAME      VARCHAR2(128)
COMMENTS        VARCHAR2(4000)
```

```
SELECT *
FROM   DICTIONARY
WHERE  table_name LIKE '%SEQ%';
```

	TABLE_NAME	COMMENTS
1	DBA_SEQUENCES	Description of all SEQUENCEs in the database
2	USER_SEQUENCES	Description of the user's own SEQUENCEs
3	ALL_SEQUENCES	Description of SEQUENCEs accessible to the user
4	GV$REPLAY_CONTEXT_SEQUENCE	Synonym for GV_$REPLAY_CONTEXT_SEQUENCE
5	V$REPLAY_CONTEXT_SEQUENCE	Synonym for V_$REPLAY_CONTEXT_SEQUENCE
6	SEQ	Synonym for USER_SEQUENCES

`USER_OBJECTS` and `ALL_OBJECTS` Views in Oracle

`USER_OBJECTS`:

- Query `USER_OBJECTS` to see all the objects that you own.
- Using `USER_OBJECTS`, you can obtain a listing of all object names and types in your schema, plus the following information:
  - Date created
  - Date of last modification
  - Status (valid or invalid)

`ALL_OBJECTS`:

- Query `ALL_OBJECTS` to see all the objects to which you have access.

USER_OBJECTS View

```
SELECT object_id, object_name,
object_type, created, status
FROM   user_objects
ORDER BY object_type DESC;
```

	OBJECT_ID	OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
1	262067	RICH_PEOPLE	VIEW	29-JUN-2015	VALID
2	262064	CUST_ORDERS	VIEW	29-JUN-2015	VALID
3	261067	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
4	261068	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
5	261069	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
6	261070	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
7	261071	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
8	261066	PART_ORDERS	TABLE PARTITION	22-JUN-2015	VALID
9	103177	POSITIONS	TABLE	28-AUG-2014	VALID
10	105054	WORK	TABLE	27-OCT-2014	VALID
11	103264	WORKERS	TABLE	29-AUG-2014	VALID

Table Information

USER_TABLES:

```
DESCRIBE user_tables -- more columns
```

```
desc user_tables
Name                       Null     Type
-------------------------- -------- -------------
TABLE_NAME                 NOT NULL VARCHAR2(128)
TABLESPACE_NAME                     VARCHAR2(30)
CLUSTER_NAME                        VARCHAR2(128)
IOT_NAME                            VARCHAR2(128)
STATUS                              VARCHAR2(8)
```

```
SELECT table_name,tablespace_name,status,num_rows,
avg_row_len,blocks,LAST_ANALYZED
FROM   user_tables; -- more rows
```

	TABLE_NAME	TABLESPACE_NAME	STATUS	NUM_ROWS	AVG_ROW_LEN	BLOCKS	LAST_ANALYZED
1	CONTINENTS	USERS	VALID	4	10	5	27-AUG-2014
2	COUNTRIES	USERS	VALID	27	24	5	20-JUN-2015
3	DIVISIONS	USERS	VALID	24	47	5	22-JUN-2015
4	CUSTOMERS	USERS	VALID	135	93	5	22-JUN-2015
5	POSITIONS	USERS	VALID	21	35	5	21-JUN-2015
6	WORKERS	USERS	VALID	53	127	1	28-JUN-2015
7	PRODUCTS	USERS	VALID	100	49	4	01-SEP-2014
8	ITEMS	USERS	VALID	1758140	28	7930	17-JAN-2015
9	ORDERS	USERS	VALID	351441	41	2260	20-JUN-2015

Column Information

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

```
DESCRIBE user_tab_columns
Name                    Null      Type
--------------------    --------  --------------
TABLE_NAME              NOT NULL  VARCHAR2(128)
COLUMN_NAME             NOT NULL  VARCHAR2(128)
DATA_TYPE                         VARCHAR2(128)
DATA_TYPE_MOD                     VARCHAR2(3)
DATA_TYPE_OWNER                   VARCHAR2(128)
DATA_LENGTH             NOT NULL  NUMBER
DATA_PRECISION                    NUMBER
DATA_SCALE                        NUMBER
NULLABLE                          VARCHAR2(1)
COLUMN_ID                         NUMBER
DEFAULT_LENGTH                    NUMBER
DATA_DEFAULT                      LONG()
NUM_DISTINCT                      NUMBER
LOW_VALUE                         RAW(1000 BYTE)
HIGH_VALUE                        RAW(1000 BYTE)
DENSITY                           NUMBER
NUM_NULLS                         NUMBER
```

Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'WORKERS';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NULLABLE
1	BORN	DATE	7			Y
2	WORKER_ID	NUMBER	22	6	0	N
3	FIRST_NAME	VARCHAR2	25			Y
4	LAST_NAME	VARCHAR2	25			N
5	EMAIL	VARCHAR2	25			N
6	START_DATE	DATE	7			N
7	POSITION_ID	VARCHAR2	15			N
8	SALARY	NUMBER	22	8	2	Y
9	COMMISSION	NUMBER	22	8	2	Y
10	MANAGER_ID	NUMBER	22	6	0	Y
11	DIVISION_ID	NUMBER	22	4	0	Y

Constraint Information

- `USER_CONSTRAINTS` describes the constraint definitions on your tables.
- `USER_CONS_COLUMNS` describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

```
DESCRIBE user_constraints
Name                 Null     Type
-------------------- -------- --------------
OWNER                         VARCHAR2(128)
CONSTRAINT_NAME      NOT NULL VARCHAR2(128)
CONSTRAINT_TYPE               VARCHAR2(1)
TABLE_NAME           NOT NULL VARCHAR2(128)
SEARCH_CONDITION              LONG()
SEARCH_CONDITION_VC           VARCHAR2(4000)
R_OWNER                       VARCHAR2(128)
R_CONSTRAINT_NAME             VARCHAR2(128)
DELETE_RULE                   VARCHAR2(9)
STATUS                        VARCHAR2(8)
DEFERRABLE                    VARCHAR2(14)
DEFERRED                      VARCHAR2(9)
VALIDATED                     VARCHAR2(13)
GENERATED                     VARCHAR2(14)
BAD                           VARCHAR2(3)
RELY                          VARCHAR2(4)
LAST_CHANGE                   DATE
```

USER_CONSTRAINTS: Example

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'WORKERS';
```

	CONSTRAINT_NAME	CON_TYPE	SEARCH_CONDITION	R_CONSTRAINT_NAME
1	WORKERS_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	
2	WORKERS_EMAIL_NN	C	"EMAIL" IS NOT NULL	
3	WORKERS_START_DATE_NN	C	"START_DATE" IS NOT NULL	
4	WORKERS_POSITION_NN	C	"POSITION_ID" IS NOT NULL	
5	WORKER_SALARY_MIN	C	salary > 0	
6	WORKERS_DIVISION_FK	R		DIVISIONS_ID_PK
7	WORKERS_POSITION_FK	R		POSITION_ID_PK
8	WORKERS_MANAGER_FK	R		WORKERS_WORKER_ID_PK
9	WORKER_EMAIL_UK	U		
10	WORKERS_WORKER_ID_PK	P		
11	WORKER_COMM_CK	C	DECODE(SUBSTR(position_id,1,5),'SALES',NULL,COMMISSION) IS NULL	

Querying USER_CONS_COLUMNS

```
DESCRIBE user_cons_columns
```

```
SELECT constraint_name, table_name column_name,position
FROM   user_cons_columns
WHERE  table_name = 'WORKERS';
```

	CONSTRAINT_NAME	COLUMN_NAME	POSITION
1	WORKERS_LAST_NAME_NN	WORKERS	
2	WORKERS_EMAIL_NN	WORKERS	
3	WORKERS_START_DATE_NN	WORKERS	
4	WORKERS_POSITION_NN	WORKERS	
5	WORKER_SALARY_MIN	WORKERS	
6	WORKER_EMAIL_UK	WORKERS	1
7	WORKERS_WORKER_ID_PK	WORKERS	1
8	WORKERS_POSITION_FK	WORKERS	1
9	WORKERS_MANAGER_FK	WORKERS	1
10	WORKER_COMM_CK	WORKERS	
11	WORKER_COMM_CK	WORKERS	
12	WORKERS_DIVISION_FK	WORKERS	1

More realistic query

We often need a complex list of constraints

```
SELECT c.constraint_name, c.constraint_type con_type,
search_condition, cl.column_name,
cl.position, c.r_constraint_name
FROM   user_constraints c, user_cons_columns cl
WHERE  c.constraint_name = cl.constraint_name
AND c.table_name = 'WORKERS';
```

	CONSTRAINT_NAME	CON_TYPE	SEARCH_CONDITION	COLUMN_NAME	POS...	R_CONSTRAINT_NAME
1	WORKER_COMM_CK	C	DECODE(SUBSTR(position_id,1,5),'SALES',NULL,COMMISSION) IS NULL	COMMISSION		
2	WORKER_COMM_CK	C	DECODE(SUBSTR(position_id,1,5),'SALES',NULL,COMMISSION) IS NULL	POSITION_ID		
3	WORKERS_WORKER_ID_PK	P		WORKER_ID	1	
4	WORKER_EMAIL_UK	U		EMAIL	1	
5	WORKERS_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	LAST_NAME		
6	WORKER_SALARY_MIN	C	salary > 0	SALARY		
7	WORKERS_POSITION_NN	C	"POSITION_ID" IS NOT NULL	POSITION_ID		
8	WORKERS_START_DATE_NN	C	"START_DATE" IS NOT NULL	START_DATE		
9	WORKERS_EMAIL_NN	C	"EMAIL" IS NOT NULL	EMAIL		
10	WORKERS_DIVISION_FK	R		DIVISION_ID	1	DIVISIONS_ID_PK
11	WORKERS_MANAGER_FK	R		MANAGER_ID	1	WORKERS_WORKER_ID_PK
12	WORKERS_POSITION_FK	R		POSITION_ID	1	POSITION_ID_PK

Index Information

Often we need some info about indexes:

```
SELECT i.table_name, i.index_name,i.index_type,
i.clustering_factor ,i.blevel,i.num_rows,
       c.column_name ,c.column_position
FROM   user_indexes i, user_ind_columns c
WHERE  i.index_name= c.index_name
AND
c.table_name = 'WORKERS'
ORDER BY i.index_name,c.column_position;
```

	TABLE_NAME	INDEX_NAME	INDEX_TYPE	CLUSTERING_FACTOR	BLEVEL	NUM_ROWS	COLUMN_NAME	COLUMN_POSITION
1	WORKERS	WORKERS_DIVISION_ID_IX	NORMAL	1	0	52	DIVISION_ID	1
2	WORKERS	WORKERS_MANAGER_ID_IX	NORMAL	1	0	52	MANAGER_ID	1
3	WORKERS	WORKERS_NAME_IX	NORMAL	1	0	53	LAST_NAME	1
4	WORKERS	WORKERS_NAME_IX	NORMAL	1	0	53	FIRST_NAME	2
5	WORKERS	WORKERS_POSITION_ID_IX	NORMAL	1	0	53	POSITION_ID	1
6	WORKERS	WORKERS_WORKER_ID_PK	NORMAL	1	0	53	WORKER_ID	1
7	WORKERS	WORKER_EMAIL_UK	NORMAL	1	0	53	EMAIL	1

Monitoring the database by using the dynamic performance tables and views

Dynamic Performance Tables

- Most database engines collect masive amount of information about the internal activities of a system
- Mostly this info is a large set of numbers, that count different events (actions) inside the system.
- In case of an Oracle database this means hundreds of thousands of data
- For example there is a counter of executions of SQL statements. It’s initial value is 0 when the database software starts running, and every time a SQL statement gets executed, this number is incremented by the database management system’s code
- All these numbers are presented to users through some „fictive” tables. In case of Oracle these table names start with X$ and only the „SYS” (some kind of superuser) can access them. For example:
    - SELECT * FROM X$KCFIO   where the abbreviation stands for „Kernel„, „Cache” layer, File Input/Output

Dynamic Performance Views

- In Oracle a large set of well documented and more user friendly views are defined on top of the dynamic performance tables. These are the dynamic performance views. They are accessible to the database administrators.
- These views are called dynamic performance views because they are not read consistent, but rather dynamic. And they are mostly used for performance monitoring are tuning purposes.
- Although these views appear to be regular database views, they are not. These views provide data on internal disk structures and memory structures. You can select from these views, but you can never update or alter them.
- The actual dynamic performance views are identified by the prefix V_$.
- Public synonyms for these views have the prefix V$. Database administrators and other users should access only the V$ objects, not the V_$ objects and not the underlying X$-tables

The V$SESSION view

- V$SESSION displays session information for each current session.
- Session serial number. Used to uniquely identify a session's objects.
- Guarantees that session-level commands are applied to the correct session objects if the session ends and another session begins with the same session ID.

```
SELECT sid, serial#,username, taddr, lockwait, status,
schemaname, osuser,machine, terminal, program
FROM v$session
WHERE username NOT LIKE '%SYS%';
```

	SID	SERIAL#	USERNAME	TADDR	LOCKWAIT	STATUS	SCHEMANAME	OSUSER	MACHINE	TERMINAL	PROGRAM
1	256	719	LEARNING	00007FFAE2597440		INACTIVE	LEARNING	user	Lenovo2-PC	unknown	SQL Develo
2	386	1601	LEARNING	00007FFAE2597FD0	00007FFAE5479CA8	ACTIVE	LEARNING	Lenovo2-PC\user	WORKGROUP\LENOVO2-PC	LENOVO2-PC	sqlplus.ex

```
SELECT sid, serial#,username, taddr, lockwait,sql_id ,
row_wait_obj#, row_wait_file#, row_wait_block#, row_wait_row#
FROM v$session
WHERE username NOT LIKE '%SYS%';
```

	SID	SERIAL#	USERNAME	TADDR	LOCKWAIT	SQL_ID	ROW_WAIT_OBJ#	ROW_WAIT_FILE#	ROW_WAIT_BLOCK#	ROW_WAIT_ROW#
1	256	719	LEARNING	00007FFAE2597440			103074	6	67214	0
2	386	1601	LEARNING	00007FFAE2597FD0	00007FFAE5479CA8	ds5pd6qanbd7r	103074	6	67212	6

Some interesting V$ Views I.

- **V$SQL** lists statistics on shared SQL areas without the GROUP BY clause and contains one row for each child of the original SQL text entered.
- **V$SQLAREA** displays statistics on shared SQL areas and contains one row per SQL string. It provides statistics on SQL statements that are in memory, parsed, and ready for execution.
- **V$SQL_WORKAREA** displays information about work areas used by SQL cursors.
  Each SQL statement stored in the shared pool has one or more child cursors that are listed in the V$SQL view.
  V$SQL_WORKAREA lists all work areas needed by these child cursors;
- **V$PARAMETER** displays information about the contents of the server parameter file. If a server parameter file was not used to start the instance, then each row of the view will contain FALSE in the ISSPECIFIED column
- **V$PGASTAT** displays PGA memory usage statistics as well as statistics about the automatic PGA memory manager when it is enabled (that is, when PGA_AGGREGATE_TARGET is set).
  Cumulative values in V$PGASTAT are accumulated since instance startup.

Some interesting V$ Views II.

- **V$SQL_PLAN** contains the execution plan information for each child cursor loaded in the library cache.
- **V$SQL_PLAN_STATISTICS** provides execution statistics at the row source level for each child cursor.
- **V$SQLSTATS** displays basic performance statistics for SQL cursors and contains one row per SQL statement (that is, one row per unique value of SQL_ID)
- **V$TRANSACTION** lists the active transactions in the system.
- **V$LOCK** lists the locks currently held by the Oracle Database and outstanding requests for a lock or latch.
- **V$VERSION** displays version numbers of core library components in the Oracle Database. There is one row for each component.

Database transactions

Introduction to Transactions

- A transaction is a logical, atomic unit of work that contains one or more SQL statements. (DML, SELECT … FOR UPDATE)
- A transaction groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database.
- For example the Oracle Database assigns every transaction a unique identifier called a transaction ID. It is an identifier that is unique to a transaction and represents the undo segment number, slot, and sequence number.

```
UPDATE newworker SET last_name=last_name;

SELECT xid AS "txn id", xidusn AS "undo seg",
xidslot AS "slot", xidsqn AS "seq", status AS "txn status",
DBMS_TRANSACTION.LOCAL_TRANSACTION_ID
FROM V$TRANSACTION;
```

```
txn id                undo seg       slot        seq txn status       LOCAL_TRANSACTION_ID
---------------- ---------- ---------- ---------- ---------------- ----------------------
05000C00E5420000          5         12      17125 ACTIVE           5.12.17125
```

Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement

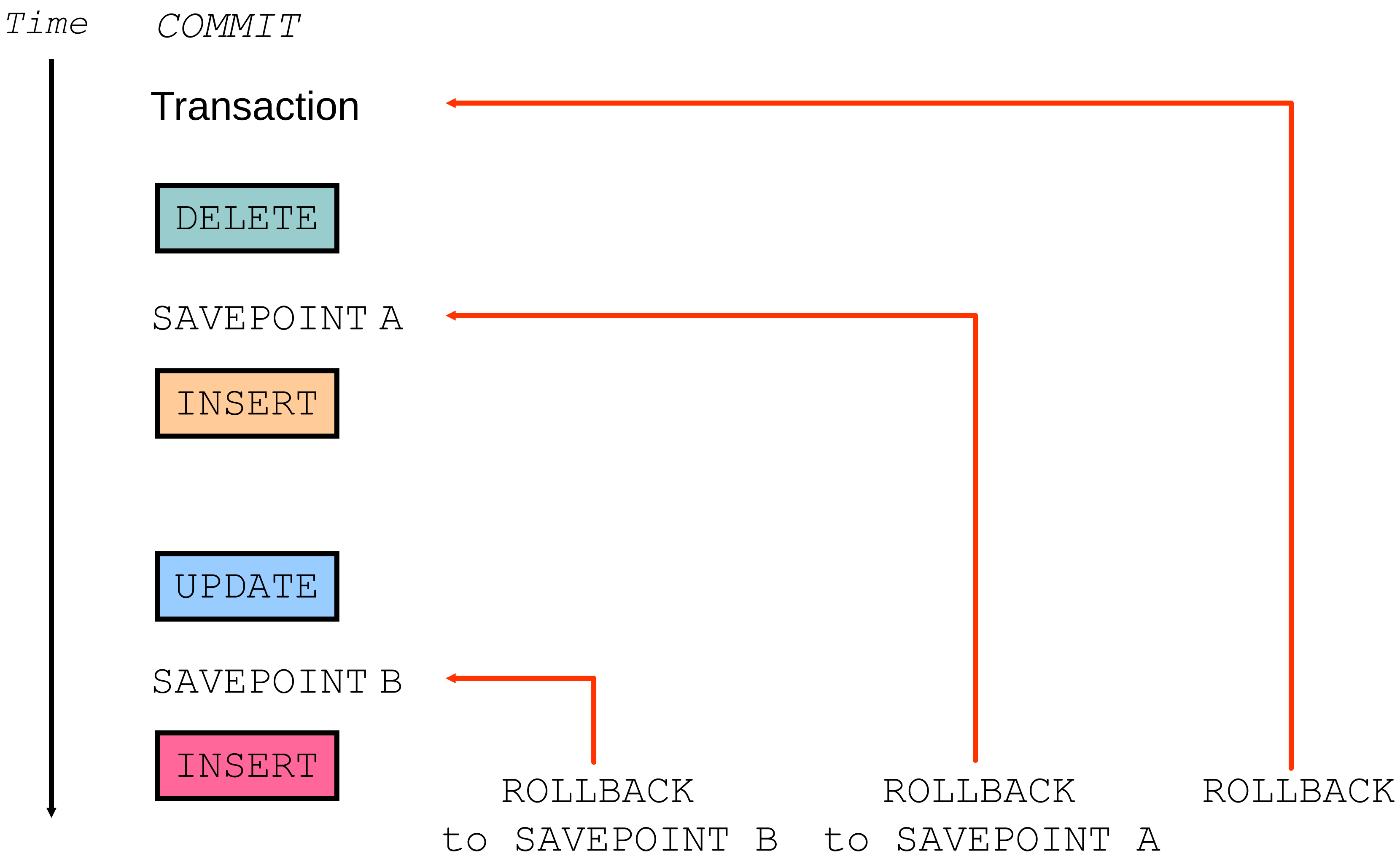
Database Transactions

- Begin when the first DML SQL statement is executed
- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits from the database session
  - An automatic rollback occurs under an abnormal termination of a user program, or at a system failure.
- With `COMMIT` and `ROLLBACK` statements, you can:
  - Ensure data consistency
  - Preview data changes before making changes permanent
  - Group logically related operations

Example for a simple transaction

```
SELECT COUNT(*) FROM newworker;
COUNT(*)
----------
        53
INSERT INTO newworker SELECT * FROM newworker;
53 rows inserted.
SELECT COUNT(*) FROM newworker;
COUNT(*)
----------
       106
DELETE FROM newworker WHERE position_id LIKE 'SALES%';
28 rows deleted.
SELECT COUNT(*) FROM newworker;
COUNT(*)
----------
        78
ROLLBACK;
rollback complete.
SELECT COUNT(*) FROM newworker;
COUNT(*)
----------
        53
```

Controlling Transactions



Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

State of the Data Before `COMMIT` or `ROLLBACK`

- The previous state of the data can be recovered.
- The current session can review the results of the DML operations by using the `SELECT` statement.
- Other sessions *cannot* view the results of the DML statements by the current session.
- The affected rows are *locked*; other sessions cannot change the data in the affected rows.

State of the Data After `COMMIT`

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All sessions can view the results.
- Locks on the affected rows are released;
  those rows are available for other sessions to manipulate.
- All savepoints are erased.

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
22 rows deleted.
ROLLBACK ;
Rollback complete.
```

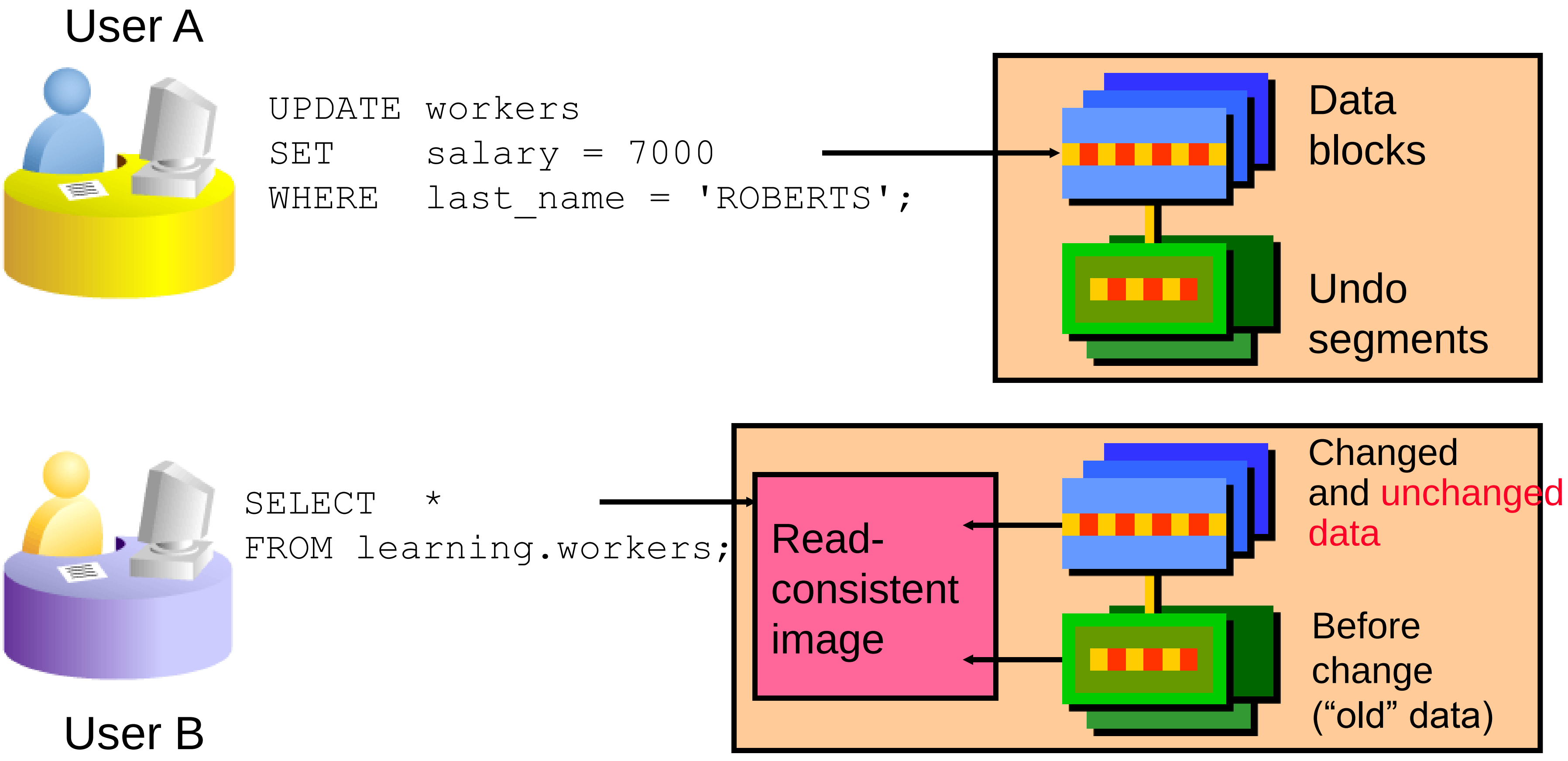
Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with changes made by another user.
- Read consistency ensures that on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers
- Database users access the database in two ways:
  - Read operations (`SELECT` statement)
  - Write operations (`INSERT, UPDATE, DELETE`)
- You need read consistency so that the following occur:
  - The database reader and writer are ensured a consistent view of the data.
  - Readers do not view data that is in the process of being changed.
  - Writers are ensured that the changes to the database are done in a consistent manner.
  - Changes made by one writer do not disrupt or conflict with the changes being made by another writer.

Implementation of Read Consistency



Simultaneous activity of more than one session: ACID properties, serializability, locking,

- ACID:
  - Atomicity: each transaction is a single unit
  - Consistency: bringing the data from one valid state to the other
  - Isolation: despite concurrent execution of transactions the result is the same as if it would be with sequential execution of transactions
  - Durability: once comitted it will stay there even in case of system failures
- Simultaneous activity is enabled by introducing a locking mechanism.
- Rows are the units that are locked most often: „row level locking”. Table level locking also takes place during the ongoing work of a database engine.

Locking mechanism and the deadlock

- If one session locks a row (Exclusively), and another session tries to lock the same row, it will wait until the lock is released.
- Locks are released when the transaction holding the lock is finished (either with COMMIT or with ROLLBACK)
- If many transaction run simultaneously, they may form a loop in which each of them is waiting upon the other transaction to release the lock
- In this case no transaction can release any lock because they are all blocked, this is called **deadlock**.
- Most database engines allow deadlocks to occur, but detect it’s occurence and then they roll back one of the statements that are in a deadlock situation.

Two-Phase Commit Mechanism

- Unlike a transaction on a local database, a distributed transaction involves altering data on multiple databases
- The database ensures the integrity of data in a distributed transaction using the two-phase commit mechanism.
- In the prepare phase, the initiating node in the transaction asks the other participating nodes to promise to commit or roll back the transaction.
- During the commit phase, the initiating node asks all participating nodes to commit the transaction.
- If this outcome is not possible, then all nodes are asked to roll back.

Phase	Description
Prepare phase	The initiating node, called the **global coordinator**, asks participating nodes other than the commit point site to promise to commit or roll back the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back.
Commit phase	If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction.
Forget phase	The global coordinator forgets about the transaction.

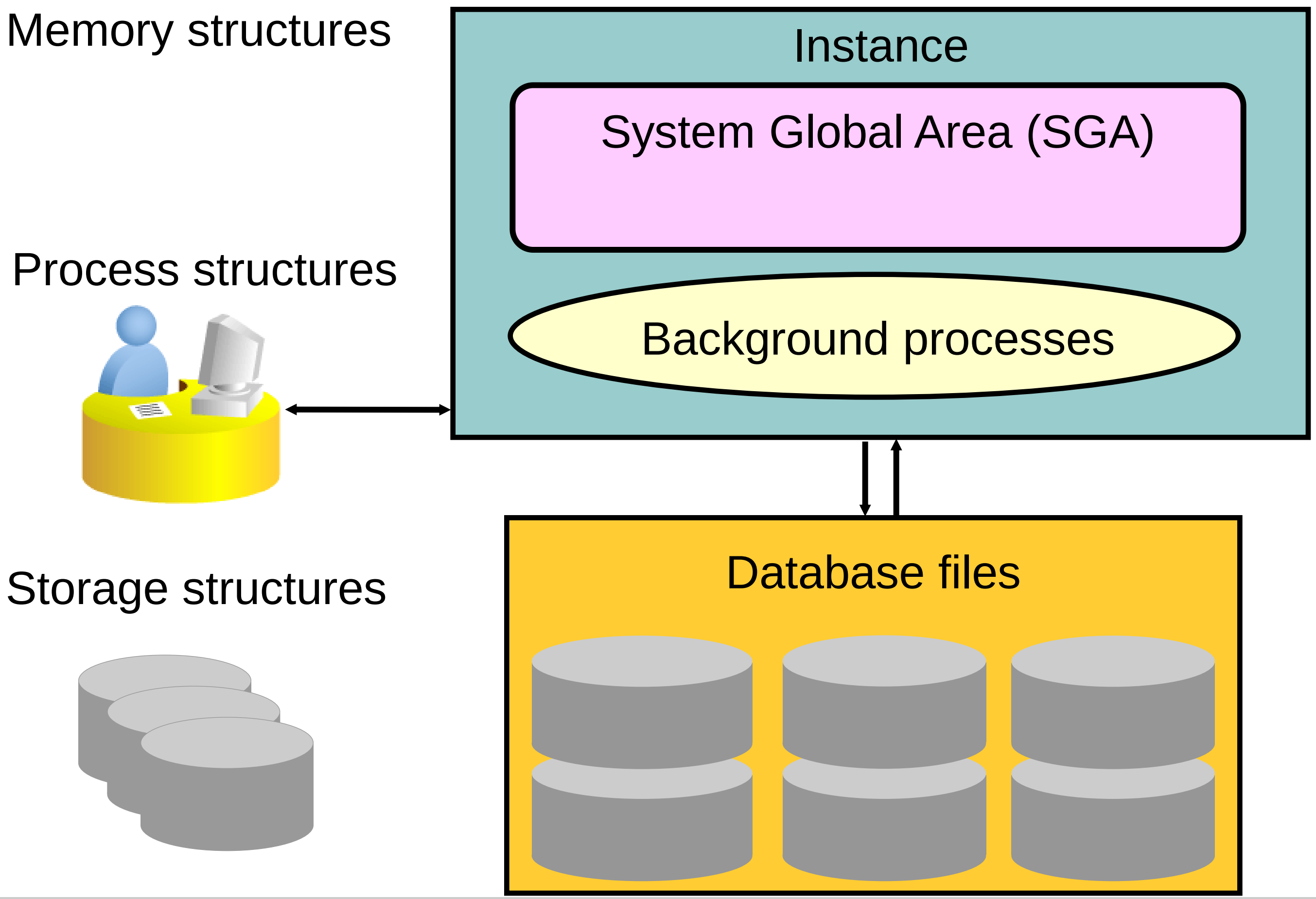
One solution of transaction management and recoverability of a database: the Oracle Database Architecture

An Oracle server:

- Is a database management system that provides an open, comprehensive, integrated approach to information management
- Consists of an Oracle instance and an Oracle database

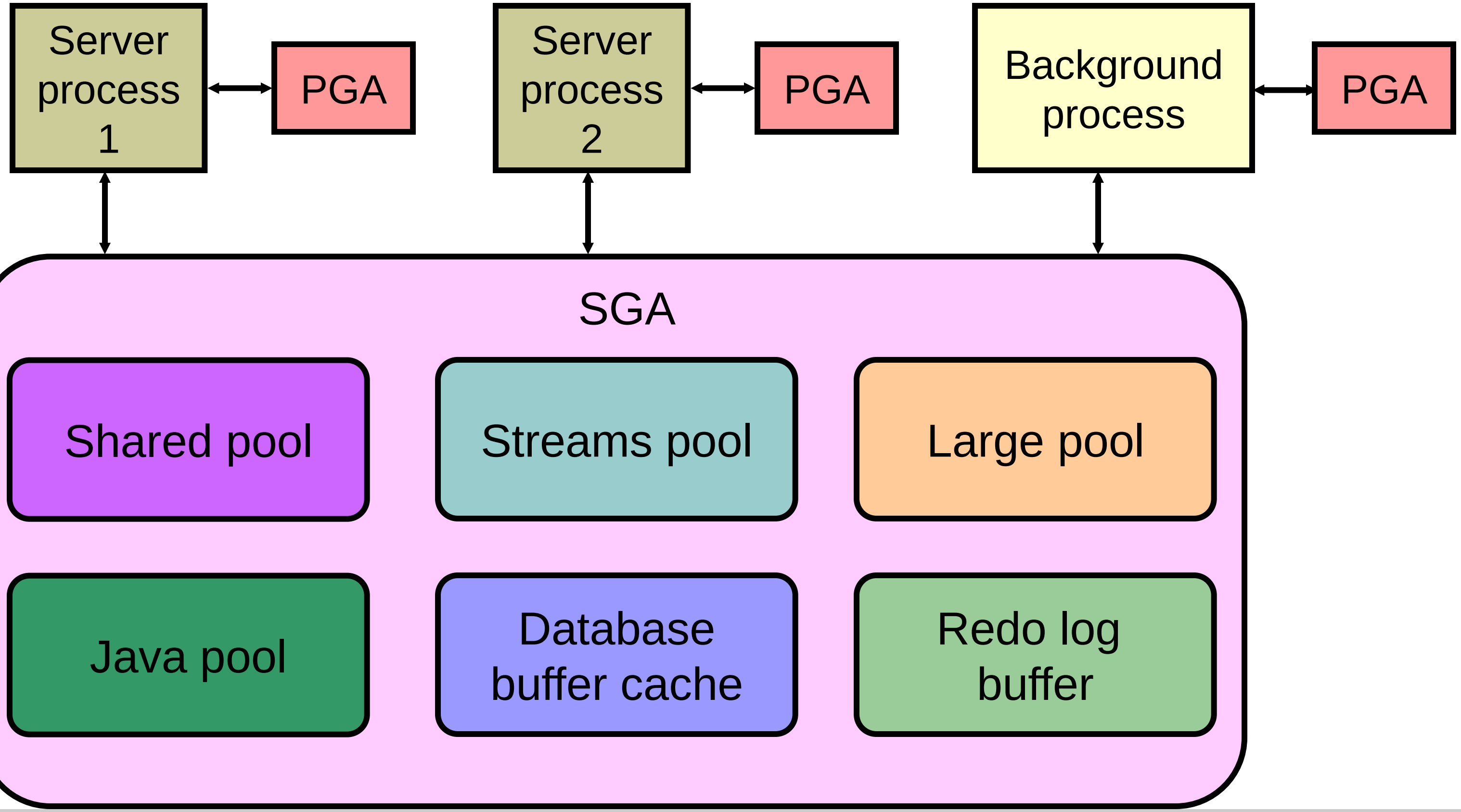
Database Structures

DB structures
- Memory
- Process
- Storage



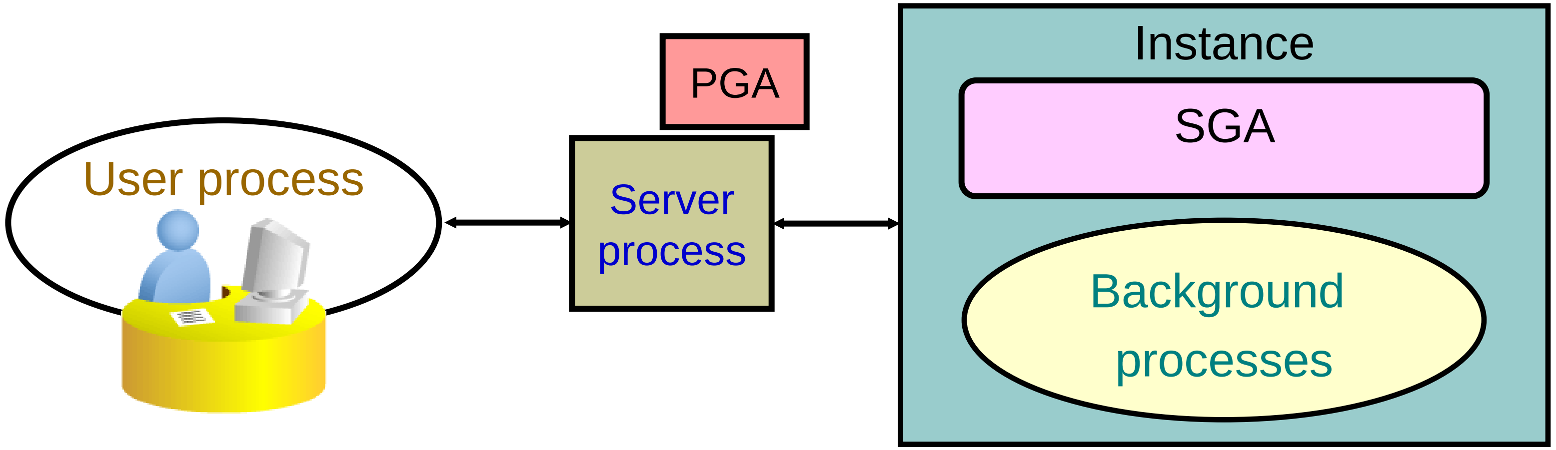
Oracle Memory Structures

DB structures
> Memory
Process
Storage



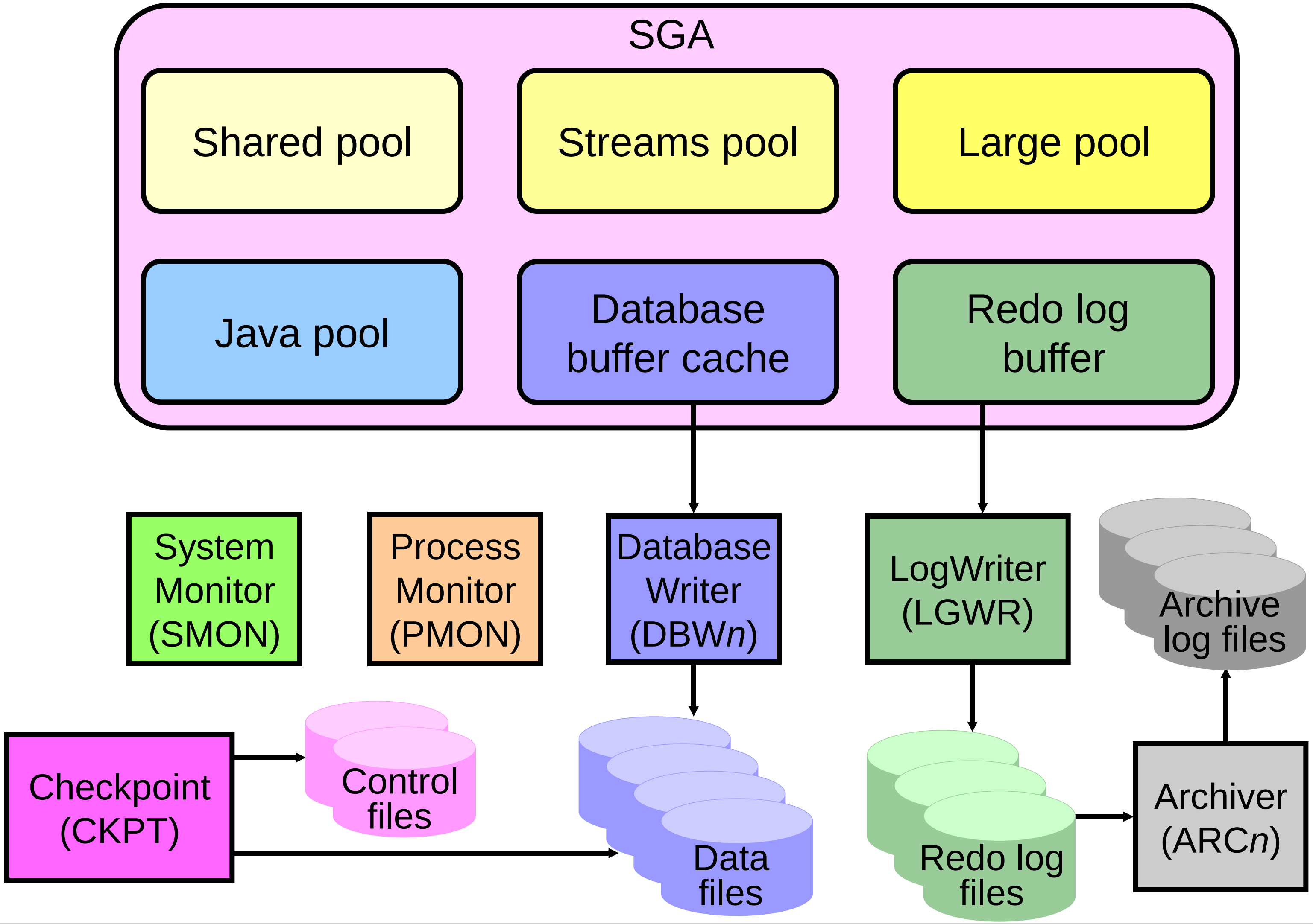
Process Structures

DB structures
Memory
> Process
Storage

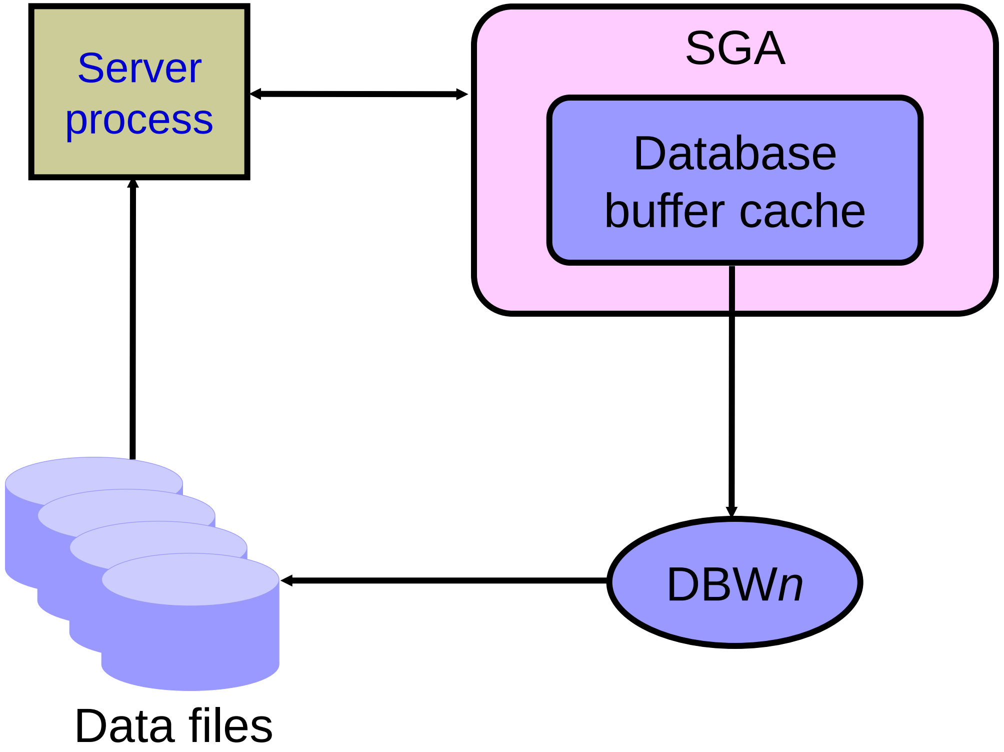


- User process: Is started at the time a database user requests a connection to the Oracle server
- Server process: Connects to the Oracle instance and is started when a user establishes a session
- Background processes: Are started when an Oracle instance is started

Oracle Instance Management



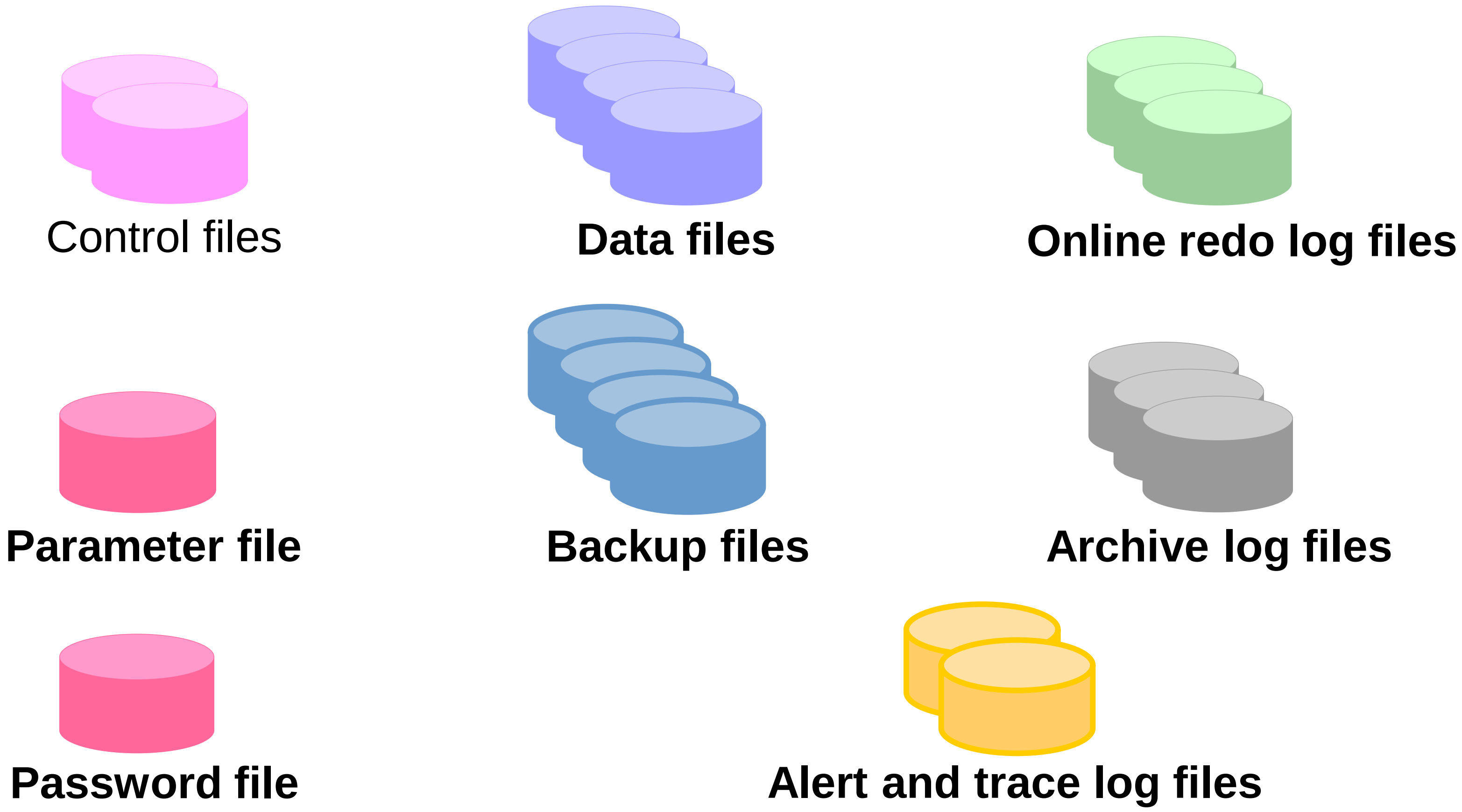
Server Process and Database Buffer Cache



Buffers:
- Pinned
- Clean
- Free or unused
- Dirty

Physical Database Structure

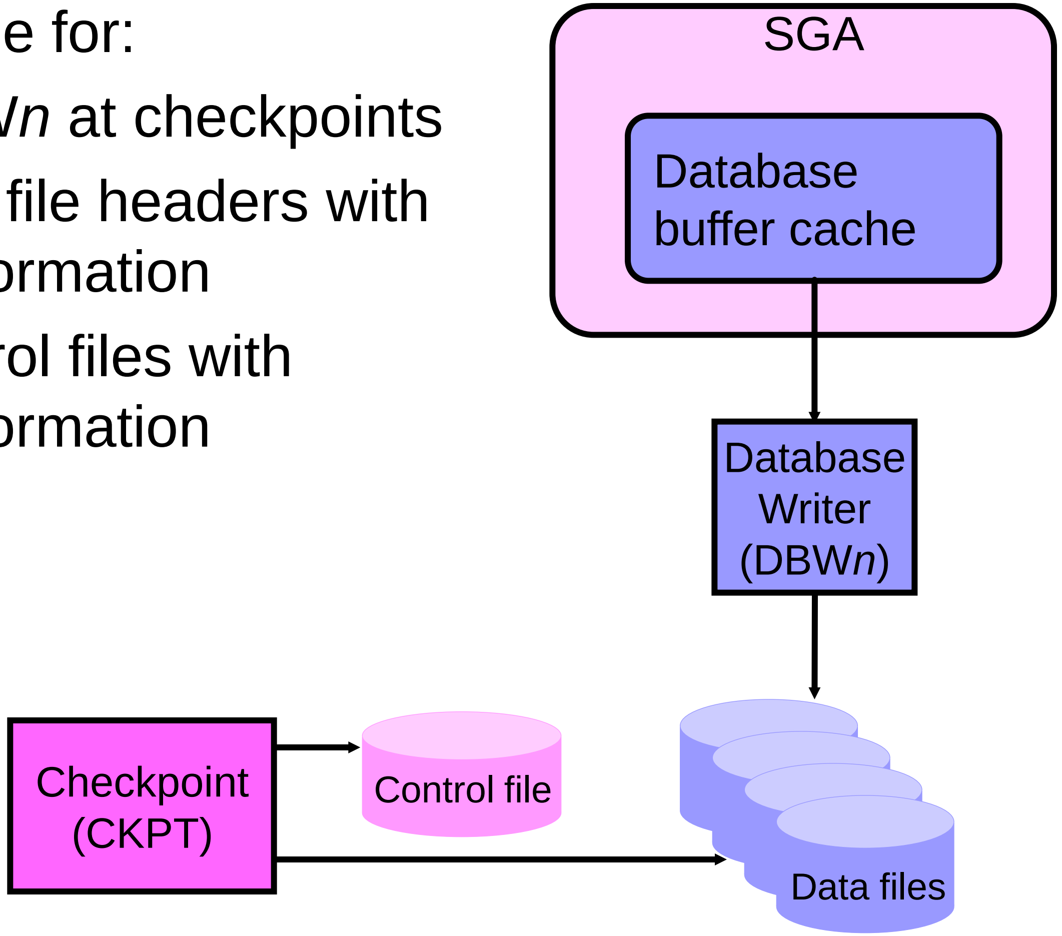
DB structures
Memory
Process
> Storage



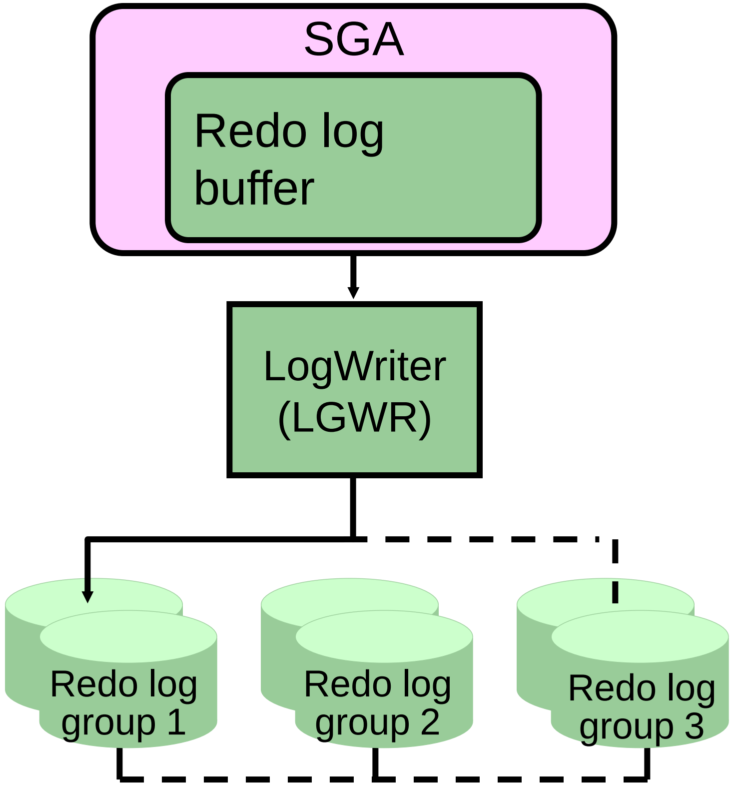
Background Processes and Recovery: Checkpoint (CKPT)

CKPT is responsible for:

- Signaling DBW*n* at checkpoints
- Updating data file headers with checkpoint information
- Updating control files with checkpoint information



Background Processes and Recovery: Redo Log Files and LogWriter



Redo log files:

- Record changes to the database
- Should be multiplexed to protect against loss

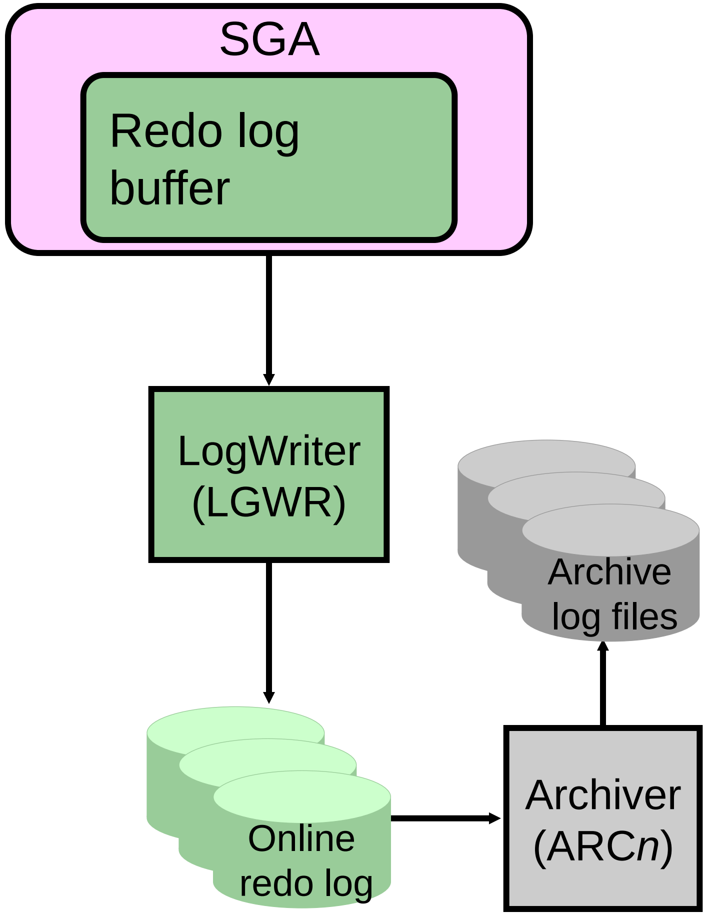
LogWriter writes:

- At commit
- When one-third full
- Every three seconds
- Before DBW*n* writes

Background Processes and Recovery: Archiver (ARC*n*)

Archiver (ARC*n*):

- Is an optional background process
- Automatically archives online redo log files when `ARCHIVELOG` mode is set for the database
- Preserves the record of all changes made to the database



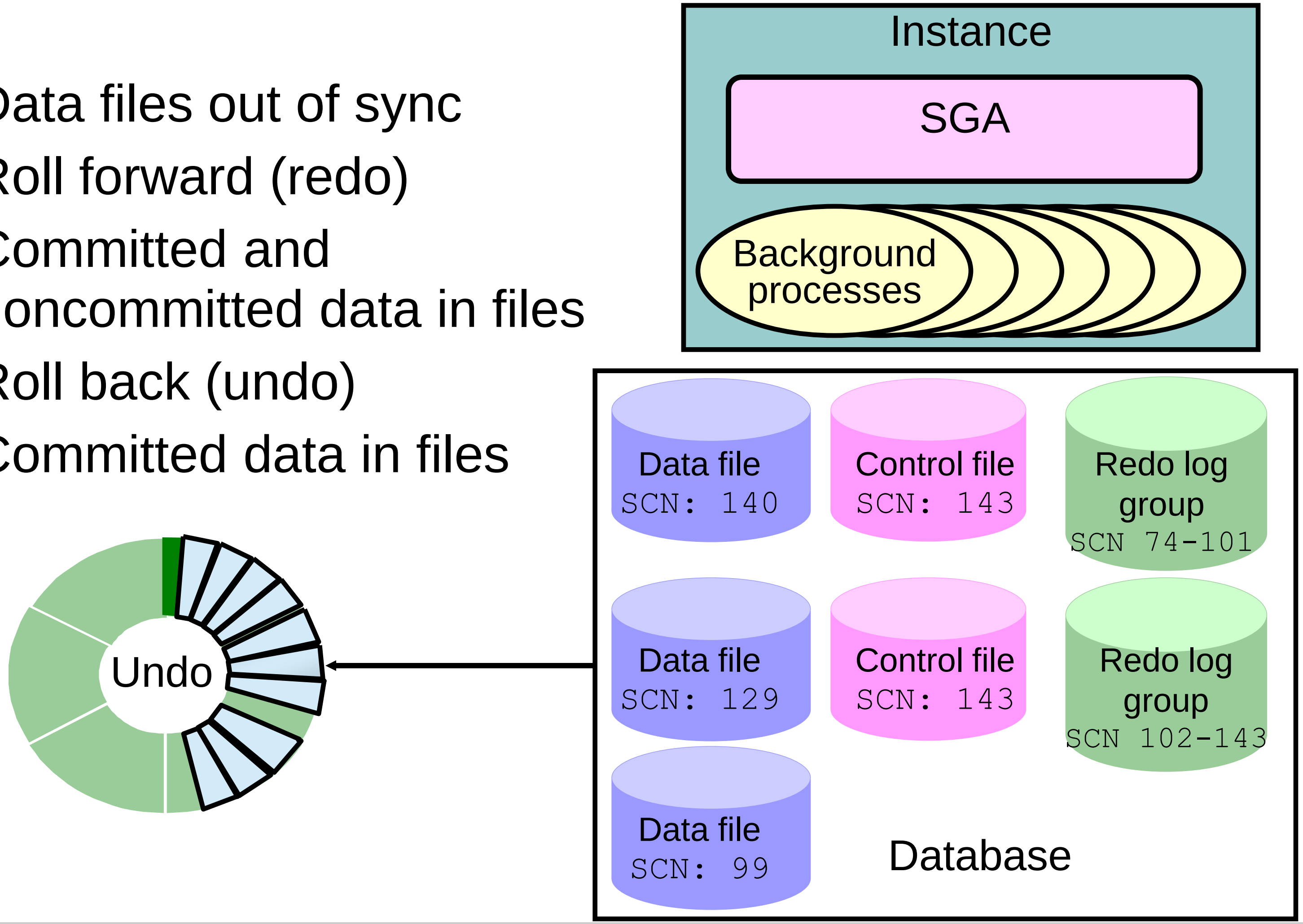
Instance Recovery

Instance or crash recovery:

- Is caused by attempts to open a database whose files are not synchronized on shutdown
- Is automatic
- Uses information stored in redo log groups to synchronize files
- Involves two distinct operations:
  - Rolling forward: Data files are restored to their state before the instance failed.
  - Rolling back: Changes made but not committed are returned to their original state.

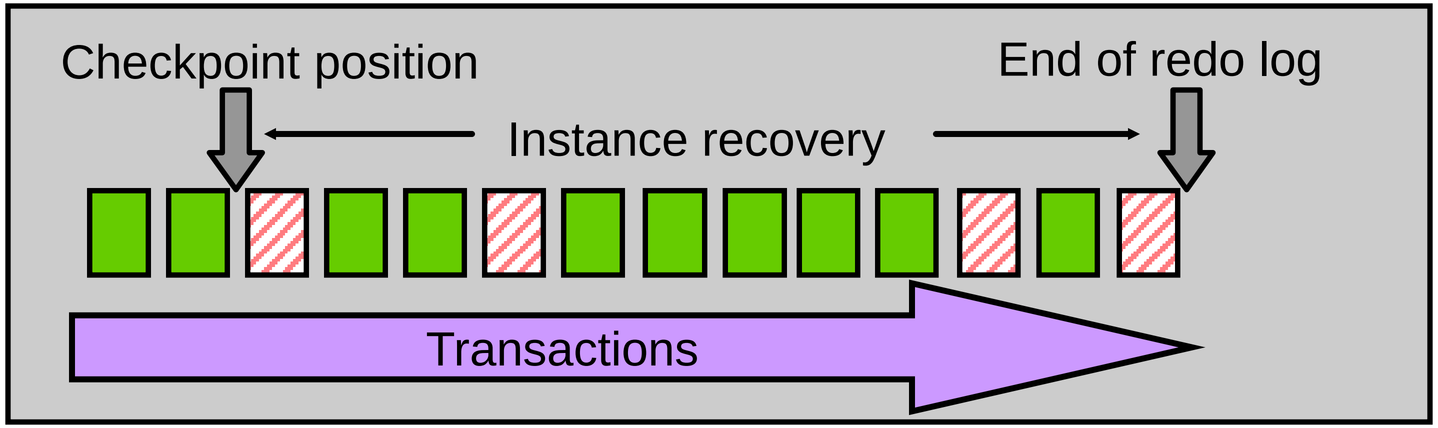
Phases of Instance Recovery

1. Data files out of sync
2. Roll forward (redo)
3. Committed and noncommitted data in files
4. Roll back (undo)
5. Committed data in files



Tuning Instance Recovery

- During instance recovery, the transactions between the checkpoint position and the end of redo log must be applied to data files.
- You tune instance recovery by controlling the „distance” between the checkpoint position and the end of redo log.



Media Failure

Typical Causes	Possible Solutions
Failure of disk drive	1. Restore the affected file from backup. 2. If necessary, inform the database about a new file location. 3. If necessary, recover the file by applying redo information.
Failure of disk controller	
Deletion or corruption of database file	

Configuring for Recoverability

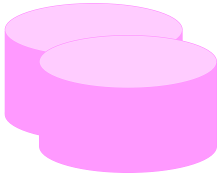
To configure your database for maximum recoverability, you must:

- Schedule regular backups
- Multiplex control files
- Multiplex redo log groups
- Retain archived copies of redo logs

Control Files

Protect against database failure by multiplexing control files. It is suggested that your database has:

- At least two copies (Oracle recommends three) of the control file
- Each copy on a separate disk
- At least one copy on a separate disk controller



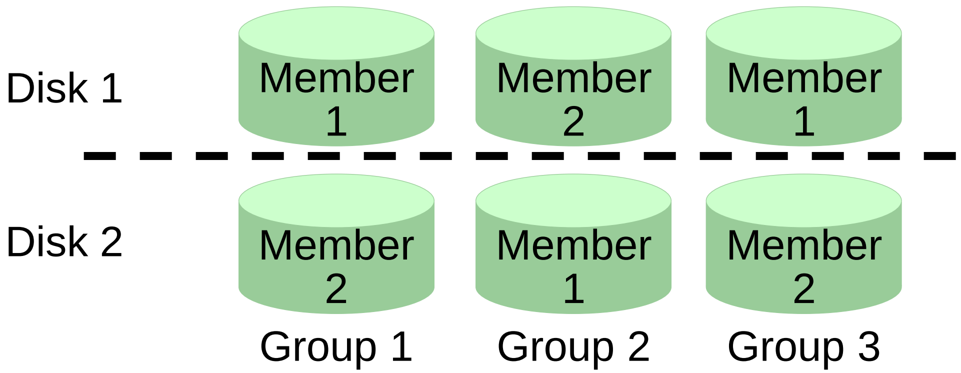
Control files

Redo Log Files

Multiplex redo log groups to protect against media failure and loss of data. It is suggested that redo log groups have:

- At least two members (files) per group
- Each member on a separate disk drive
- Each member on a separate disk controller

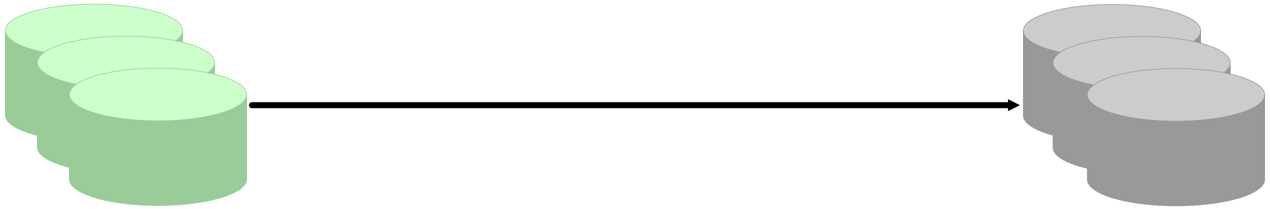
Note: Performance is heavily influenced by writing to redo logs.



Archive Log Files

To preserve redo information, create archived copies of redo log files by performing the following steps.

1. Specify archive log file naming convention.
2. Specify one or more archive log file locations.
3. Switch the database to `ARCHIVELOG` mode.



Online redo log files **Archive log files**

Database Security
Controlling User Access

Introduction to Database Security

Database security entails allowing or disallowing user actions on the database and the objects within it.

- A **user** (sometimes called a **username**) is a name defined in the database that can connect to and access objects.
- A **schema** is a named collection of objects, such as tables, views, clusters, procedures, and packages.
- **User Authentication**
  To prevent unauthorized use of a database username, database engines provide user validation through several different methods for normal database users.
- **Database Administrators**
  Each database requires at least one database administrator (DBA) to administer it.
- Schemas and users help database administrators manage database security.

Database Administrators

A database administrator's responsibilities can include the following tasks:

- Installing and upgrading the database server software and application tools
- Allocating system storage and planning future storage requirements for the database system
- Creating primary database storage structures (in case of Oracle they are called tablespaces) after application developers have designed an application
- Creating primary objects (tables, views, indexes) once application developers have designed an application
- Modifying the database structure, as necessary, based on information given by application developers
- Enrolling users and maintaining system security
- Controlling and monitoring user access to the database
- Monitoring and optimizing the performance of the database

Privileges

- A **privilege** is a right to execute a particular type of a SQL statement or to access another user's object
- Typically there are two distinct categories of privileges:
  - System privileges
  - Schema object privileges
- **System privileges**:
  - Allow an action type within the database (`CREATE TABLE, ALTER ANY USER,…`)
- **Object privileges**: A privilege or right to perform a particular action on a specific schema object
  (`GRANT SELECT ON` workers `TO` student;)
- Some schema objects, such as indexes, clusters, triggers and database links do not have associated object privileges.
  Their use is controlled with system privileges

System Privileges

- Depending on the database system a different set of privileges are available. (In Oracle more than 200!)
- The database administrator has high-level system privileges for tasks such as:
  - creating new users
  - creating a database
  - creating tablespaces or equivalent storage structures
  - creating tables and views
  - altering the definition of objects
  - removing users
  - removing tables
  - backing up tables
  - Mmnage resources

  and many more

Creating User Accounts

- The DBA creates users with the `CREATE USER` statement
- The details and options of this command are not part of the SQL standard
- In case of an Oracle database the command looks:

```
CREATE USER <user> IDENTIFIED BY <password>
[DEFAULT TABLESPACE <data_tablespace_name>]
[QUOTA <Kbytes or Mbytes> ON <data_tablespace_name>]
[TEMPORARY TABLESPACE <temporary_tablespace_nameg>]
[QUOTA <Kbytes or Mbytes> ON <temporary_tablespace_name>]
[PROFILE <profile_name>]
```

- For example:

```
CREATE USER  student1
IDENTIFIED BY oracle;
```

Grant and Revoke System Privileges

- After a user is created, specific system privileges must be granted to that user by the administrators:

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...]
[WITH ADMIN OPTION];
```

- System privileges for users and roles can or revoked using the following:

```
REVOKE {privilege [, privilege...]|ALL}
FROM   {user[, user...]|role|PUBLIC};
```

Examples for Granting System Privileges

The privileged user can grant specific system privileges to a user.

```
GRANT   create session, create table,
        create sequence, create view
TO      student1;
GRANT succeeded.
```

Using the WITH ADMIN OPTION, the grantee becomes administrator of that privige(s). Issue as a DBA user:

```
GRANT   create procedure
TO    student1
WITH ADMIN OPTION;
GRANT succeeded.
```

Now issue the following GRANT statement as *student1* user:

```
GRANT   create procedure
TO    scott;
GRANT succeeded.
```

REVOKE System Privileges

The privileges can be removed with REVOKE statement from specific user or role.

- Revoke system privileges from users and roles
- Revoke roles from users, roles, and program units.

Prerequisites

- To revoke a system privilege, you must have been granted the privilege with the ADMIN OPTION.
- You can revoke any privilege if you have the GRANT ANY PRIVILEGE system privilege.

```
REVOKE  create sequence, create view
FROM    student1;
REVOKE succeeded.
```

What Is a Role?

- A **role** is a set or group of privileges that can be granted together to users or to another role.
- Roles can be granted to and revoked from users simultaneously.
- Having the CREATE ROLE system privilege you can create a role with the following syntax

```
CREATE ROLE role_name
[ NOT IDENTIFIED] | IDENTIFIED {BY password |
USING [schema.] package | EXTERNALLY | GLOBALLY } ;
```

Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE developer;
```

- Grant privileges to a role:

```
GRANT create table, create view, create procedure
TO developer;
```

- Grant a role to users:

```
GRANT developer TO student1;
```

About Object privileges

- A **schema object privilege** is a privilege or right to perform a particular action on a specific schema object:
  - Table, View, Sequence
  - Procedure , Function ,Package
- Different object privileges are available for different types of schema objects.
- Some schema objects, such as clusters, indexes, triggers, and database links, do not have associated object privileges.
- Schema object privileges can be granted to and revoked from users and roles.
- **A user automatically has all object privileges for schema objects contained in his or her schema.**
- Object privileges for users and roles can be granted or revoked using the following:
  - `GRANT`
  - `REVOKE`

Example of some object privileges in Oracle (Not all!)

1	ALTER	12	DEBUG
2	CREATE	13	FLASHBACK
3	COMMENT	14	REFERENCES
4	DELETE	15	EXECUTE
5	GRANT	16	MERGE VIEW
6	INDEX	17	READ
7	INSERT	18	WRITE
8	LOCK	20	ON COMMIT REFRESH
9	RENAME	21	QUERY REWRITE
10	SELECT	22	FLASHBACK ARCHIVE
11	UPDATE	23	USE

Granting Object Privileges

- Grant query privileges on the *workers* table:

```
GRANT   select ON workers
TO      student1;
GRANT succeeded.
```

- Grant privileges to update specific columns to users and roles:

```
GRANT   update (division_name, city) ON divisions
TO      student1, developer;
GRANT succeeded.
```

- Grant all privileges on *customers* table to scott user

```
GRANT   ALL privileges ON  customers
TO      scott;
GRANT succeeded.
```

Revoking Object Privileges

- You use the **`REVOKE`** statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM    {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

Revoking Object Privileges

Revoke the `SELECT` and `INSERT` privileges given to the `STUDENT1` user on the `DIVISIONS` table:

```
REVOKE  SELECT, INSERT ON divisions
FROM    student1;
```

Mi alapján válasszunk a relációs adatbázisok közül?

avagy:

A relációs adatbázisok kötelezően elvárható és opcionális képességei

Mi alapján választunk a relációs adatbázisok közül?

- A szoftver ára és a működési sebesség (performancia) messze nem az egyedüli kritériumok.
- És még ezek a kritériumok is igen összetettek – sok különböző aspektusuk és összetevőjük van.
- 1-1 ilyen választásnak jellemzően évtizedes igen széleskörű következményei vannak egy szervezet (pl. egy cég) működésére.
- Befolyásolja majd ez a személyzeti kérdéseket is, meg a szóba jöhető (kiválasztható) szoftver-rendszerek halmazát is.
- A kérdés azért is rendkívüli fontosságú, mert gyakran az informatikára elköltött pénzek legnagyobb részét is ez képezi.

1. Azok a képességek, amelyekkel mindegyik relációs adatábiskezelőnek rendelkeznie kell

- Táblák, nézetek
- SQL nyelv: DDL, DML, Query
- Tranzakciókezelés
- ACID képességek:
  - Atomi tranzakciók (Atomicity)
  - Konzisztencia (Consistency)
  - Izoláció (Isolation)
  - Tartósság (Durability)
- Kényszerek
- Adatszótár
- Optimalizáló (automatikus végrehajtási terv generátor)
- …

2. Az adatbáziskezelő rendszer költsége

- Licenszköltség: jellemzően a felhasználók számától vagy a hardver méretétől függ.
- A licenszben különböző szoftverkiszerelések létezhetnek és különböző opcionális elemek.
- Szoftveres támogatás (support) éves díja
- Saját gépeken való használat licenszdíja, vagy felhőből bérelt szolgáltatás bérleti díja
- Esetleges adatvesztésből fakadó károk: kicsiny az esélye (de nem nulla), viszont hatalmas az üzleti kár
- Tervezett és tervezetlen állásidőből fakadó üzleti károk
- A szükséges hardver költsége
- Az üzemeltető személyzet költségei (bérek, képzés...)

3. Kiforottság, megbízhatóság

- Sajnos minden szoftver bug-os
- Kockázatos olyan adatbáziskezelőt választani, amelyben mi fogjuk először megtapasztalni a hibát.
- A jó szoftver az, amelyet már mások is alkalmaztak ugyanolyam célokra, ugyanolyam körülmények között.
- Ha bármi kételyünk van, az interneten találunk róla valami leírást.
- Olyan az adatbáziskezelő, mint a jó bor: minél öregebb, annál jobb.

4. Elterjedtség

- A jó adatbáziskezelőt rajtunk kívül sok száz másik munkahelyen használják (a szomszédban is ez van).
- Ha szükség van egy új, de tapasztalt munkatársra, könnyen találunk.
- Ha szükség van egy konzultánsra, megtaláljuk.
- Könnyű, szinte folyamatos a tapasztalatcsere.
- Ha szakmai segítségre, esetleg csak tanácsra van szükségünk, azt több helyről is megkaphatjuk.

5. Szabványok betartása

- A relációs adatbázisok de-facto szabványa az SQL.
- Ezen belül is előny, ha minél inkább betartja a gyártó az SQL szabvány részleteit is.
- A szabványtól eltérő (azon túlmutató) utasítások ugyan hasznosak lehetnek, de kockázatosak is:
  - Később nehezebb lesz adatbáziskezelőt váltani
  - Később megszűnhet a támogatásuk

6. A támogatott programozási nyelvek

- A jó adatbáziskezelő rendszernek minél több nyelvet kell támogatnia
- Java nyelvhez: JDBC
- C, C#, .NET támogatás szinte kötelező.
- PHP támogatás is előny.
- Cobol, Fortran, egyéb klasszikus programozási nyelvek támogatása: beágyazott SQL („embedded SQL”) technológiával.

7. Hardver és operációs rendszer támogatása

- Sok architektúrát támogat: Intel, SUN SPARC, HP PA-RISC, és különböző IBM hardverarchitektúrákon is működik.
- Támogatja a virtualizációt elméletben és árazásban is.
- Előny, ha működik Linux, Windows, Solaris, AIX, HP-UX, esetleg VMS operációs rendszereken is.
- A későbbi platformváltások legyenek könnyűek.
- „Embedded Database”: egybeolvad a felhasználói program és az adatbáziskezelő rendszer. Jellemzően nincs külön adatbázis, amelyet felügyelnünk kellene.
- „Mobile Database”: okostelefonokon, PDA-kon futó rendszerek.

8. Saját gépterem, vagy felhő

- Idővel egyre jelentősebb előnnyé válik majd az, hogy nemcsak a saját géptermünkben működtethetjük, hanem bérelhetünk a felhőben is ilyen szolgáltatást: „Platform as a Service”, vagyis PaaS.
- A PaaS nemcsak elméleti lehetőség kell hogy legyen, hanem a gyakorlatban is olajozottan kell működnie.

9. Adattípusok széleskörű támogatása

- Az adatbázisokban manapság már nemcsak szövegeket, számokat és dátumokat tárolunk.
- Képek, tetszőleges szöveg, térinformatikai adatok, stb.
- XMLType, JSON
- User Defined Data Type
- Új objektumtípusok és objektumok tárolása (attribútumokkal és metódusokkal).

10. Karakterkészlet

- A Unicode támogatása szinte kötelező.
- A Unicode legújabb szabványa 10.0. Ebben 136.000 jelből áll a „repertoár” (régen ezt „Character Set”-nek nevezték volna, de ez most nem polkorrekt).
- A kódolást vagy UTF-8, vagy UTF-16 (ennek a régebbi, lebutított változata az UCS-2).
- A jó adatbáziskezelő rendszerben lehetséges a kódkonverzió az egyéb (régi) karakterkészletekről Unicode-ra.
- A jó adatbáziskezelőben ez a konverzió állásidő nélkül vagy rövid állásidővel elvégezhető.
- A jó adatbáziskezelő rendszerben nem nő számottevően a helyigény a Unicode miatt (tehát UTF-8 választható).

11. Elosztott tranzakciókezelés

- Nemcsak egy adatbázison belül kell megbízhatóan kezelnie a tranzakciókat, hanem két adatbáziskezelő rendszer között is.
- A két adatbázis közötti kommunikáció rendszerint két külön kategóriát is jelent:
  - Két azonos adatbáziskezelő rendszer közötti megbízható adatátvitel (pl. Oracle esetén az adatbázis link és a kétfázisú jóváhagyási mechanizmus teszi ezt lehetővé)
  - Két különböző gyártó rendszere között: XA tranzakciók támogatása
- Szükséges a megbízható tranzakciókezelés egy adatbáziskezelő rendszer és egy másik szoftver – mondjuk egy üzenetküldő rendszer („Messaging System”) – között is: rendszerint XA protokoll.

12. Procedurális lehetőségek

- Az SQL nyelv nagyszerű, de mégis akadnak esetek, amikor más kéne.
- Szükségünk lehet algoritmusokra is, és ezek adatbázison belüli tárolására.
- Így a feldolgozás az adatok „közelében” történhet.
- Így kiterjeszthető az adatbáziskezelő rendszer funkcionalitása:
  - Maga a gyártó is fejleszthet ilyen kiterjesztéseket
  - A felhasználó kifejlesztheti a saját kiterjesztéseit
  - A kényszerek halmaza is kibővíthető így
  - Az objektumorientált metódusok is ebben a nyelvben íródnak
- Pl. Oracle esetén két ilyen nyelv is rendelkezésre áll: PL/SQL és Java.

13. Adatbázis triggerek

- Igen hasznos egy olyan képesség, hogy bizonyos programok maguktól végrehajtódnak (elsülnek) egyes események bekövetkeztekor:
  - Adatok beszúrása előtt (Pre-INSERT triggerek). Ezeket jellemzően a beszúrás ellenőrzéseként használjuk.
  - Adatok beszúrása után (Post-INSERT triggerek). Ezeket gyakran a művelet naplózása érdekében alkalmazzuk.
  - INSERT mellett DELETE és UPDATE esetére is hasznosak a triggerek.
  - Nemcsak DML triggerek létezhetnek, hanem pl. olyanok, amelyek bejelentkezéskor, vagy pl. a teljes rendszer elindulása vagy leállása esetén futnak le.

14. A funkcionalitás kiterjesztése SQL-en túlra

- A többmillió adatbázis-felhasználó cégnek (esetleg embernek) gyakran van olyan közös igénye, amely túlmutat az SQL-en, de mégis praktikus. Ilyen pl:
  - Ütemezett feladatok végrehajtása
  - Email-ek küldése
  - Fájlok olvasása/írása
  - Üzenetek küldése és fogadása
  - …
- Amennyiben létezik procedurális lehetőség az adatbázison belül, akkor rendszerint a gyártó biztosít ilyen funkcionális kiterjesztéseket.
- Nyílt forráskódú adatbáziskezelő rendszer esetén ez lehet közösségi fejlesztés is.

15. Magas rendelkezésre állás

- A „High Availability” egyre fontosabbá válik.
- Egyre gyakrabban van szükségünk 7*24 órás rendelkezésre állásra. Persze valamiféle állásidőnek manapság még ilyenkor is mindenképpen lennie kell.
- Az állásidőnek két fajtája van:
  - Betervezett (előre bejelentett)
  - Nem betervezett (valamilyen hiba miatt)
- Pl. az „5 kilences” rendelkezésre állás (vagyis a 99.999%-os rendelkezésre állás) azt jelenti, hogy évente kb. 5-6 percet állunk csupán. Ez nagyon nehezen teljesíthető, de nem lehetetlen.
- Oracle esetén pl. a RAC opció a legfőbb magas rendelkezésre állási képesség.
- Problémás a szoftverek frissítése: oprendszer, adatbáziskezelő rendszer, az adatbázis adatszótára…

16. Helyreállíthatóság

- Előbb utóbb nemcsak összeomlik egy szoftver, hanem adatvesztés is történhet (pl. lemezhiba miatt).
- Az adatbáziskezelő rendszernek olyan mentési mechanizmus kell, amely biztosítja nemcsak a régi mentésre való visszaállást („Restore”), hanem a legfrissebb állapotba való helyreállást is („Recovery).
- A helyreállítás ideje is kritikus:
  - a jó eset manapság néhány perc
  - a nem túl jó eset több óra
  - Az elfogadhatatlanul lassú helyreállítási idő manapság a több nap
- Fontos, hogy a helyreállítás könnyű és megbízhatóan végrehajtható feladat legyen.
- Azért mindenre kiterjedő abszolút biztonság nem létezik.

17. Katasztrófatűrés

- Nemcsak a ténylegesen várható veszélyekre illik felkészülni, hanem olyan katasztrófa-helyzetekre is, amelyek szinte kizártnak tűnnek:
  - Tűzvész, árvíz, földrengés
  - Terrortámadás
- Aki ilyen ellen is védekezni akar (egyre többen), azok távoli adatbázis-másolatokat szeretnének működtetni.
- Ezt megoldhatja valamilyen hardveres vagy szoftveres távoli tükrözés is, de lehet ez akár az adatbáziskezelő rendszer extra képessége.
- Oracle esetén a DataGuard a katasztrófatűrő megoldás.

18. Hatékonyság (gyors működés)

- Szándékosan nem az elsők között került említésre. Sajnos a korai fázisban sokan ez alapján választanak. Egy erősebb hardver ellensúlyozhatja a szoftver hatékonyságát.
- Leginkább a funkcionalitás rovására válik 1-1 adatbáziskezelő rendszer gyorssá.
- Máskor azért gyors mert memóriában dolgozik. Ez persze sérülékenyebbé teszi.
- Ha a sebesség kiemelkedően fontos, akkor is csak a releváns terheléseket vegyük figyelembe.
- Szinte mindegyik adatbáziskezelő rendszer gyártója kozmetikázza a hatékonysági mutatók értékeit.
- Minden újabb verzióról azt mondják, hogy gyorsabb mint a megelőző verzió, közben rendszerint lassabb.

19. Skálázhatóság

- A sebesség mellett legalább olyan fontos kérdés, hogy ha növekszik a terhelés, tudjuk-e ehhez igazodva növelni a teljesítményt.
- Ideális a lineáris skálázhatóság lenne: kétszer annyi hardverrel kétszer annyi munka elvégzése.
- A lineáris skálázhatóság szinte csak álom
- Lehet egyre több processzor egy gépen belül, vagy egyre több számítógép egy „cluster”-ben.
- Oracle esetén a RAC lenne a válasz a skálázhatóságra is.

20. Adatok titkosítása

- A merevlemez-alapú adatbázisoknál az adatok a merevlemezen fájlokban tárolódnak
- Komoly kockázat, hogy ezeket a fájlokat esetleg ellopja valaki és így jut hozzá az adatokhoz
- Ezt a komoly adatbáziskezelők úgy védik ki, hogy a fájlban már titkosított módon tárolhatnak adatot
- Ilyenkor az INSERT utasítás „titkosítva ír”, és a SELECT utasítás fejti azt vissza.
- Aki tehát SQL művelettel fér hozzá az adatokhoz, azt ez a titkosítás „nem érinti”.
- Ugyanilyen titkosítás létezhet a kliens-szervet kommunikáció során is.
- Oracle esetén ezt „Advanced Encryption Option”-nek nevezik. És sajnos feláras lehetőség.

21. A használat auditálása

- Az adatbáziskezelő rendszerek zsargonjában auditálásnak nevezik az adatbázis-használat figyelését biztonsági célokból.
- Azt kell tudnunk, hogy melyik felhasználó mikor milyen műveleteket hajtott végre.
- Az auditálás konfigurálható kell hogy legyen:
  - Opcionálisan eldönthető, hogy legyen-e audit
  - Ha van audit, akkor konfigurálható kell hogy legyen annak a részletessége
- Az auditálás működtetése nem lassíthatja számottevően az adatbáziskezelő rendszert.
- Hatékony elemzési lehetőségek kellenek az „Audit Record”-ok felett.
- Figyelem: ez nemcsak van/nincs kérdés. Lehet auditálni úgy is, hogy az hasznavehetetlen.

22. A mentések sokszínűsége

- A jó adatbáziskezelő rendszernek saját mentési program/módszer kell, de legyen alternatíva is.
- Ezzel nemcsak lementhető, hanem egyúttal logikailag ellenőrizhető is az adatok tartalma.
- Előny az, ha ez a program az adatbáziskezelő rendszer belsejébe van beleépítve.
- Hatékony (valószínűleg párhuzamosítható) mentési módszer kell.
- Teljes és inkrementális mentésre is legyen lehetőség.
- A mentés elvégezhető legyen online.
- A mentési katalógus is nagyon hasznos.
- Lemezre és szalagos mentőegységre is lehessen menteni.
- Harmadik fél eszközeivel (módszereivel) is lehessen menteni.

23. Milyen segédprogramok léteznek

- A gyakorlatban felmerül az igény különböző nem-SQL feladatok egyszerű és hatékony elvégzésére.
- Ilyenek pl.:
  - Adatok betöltése az adatbázistáblákba:
    - szöveges fix formátumú állományokból
    - CSV („Comma Separated Values”) fájlokból
    - Excell táblákból
    - XML vagy JSON állományokból
  - Adatok igény szerinti áthordozása adatbázistáblákból bináris állományokba és később ezek visszatöltése ugyanabba, vagy másik adatbázisba
- Pl. Oracle esetén ezek az eszközök az SQL*Loader és az Oracle DataPump Export és DataPump Import.

24. Szakember általi monitorozás, hangolás lehetősége

- Az adatbázis sebessége ugyan nem a legfontosabb kiválasztási kritérium, de annak mégis hatalmas jelentősége van, ha egy rendszer nagyon precízen monitorozható, és kideríthető róla, hogy hol van a szűk-keresztmetszete
- Ha ezután még át is paraméterezhető a működés, úgy, hogy a szűkkeresztmetszet eltűnjön, vagy csupán enyhüljön, az már főnyeremény.
- Nagyon fontos, hogy ez a monitorozhatóság SELECT utasításokkal történjék. Ezáltal ugyanis harmadik fél is gyártani tud monitoring eszközt (nyílt rendszerek).

25. Automatikus monitorozás és hangolás

- Egyre több rendszert működtetünk. Egyes cégeknél többszáz adatbázis van napi használatban. Nem érkeznek a rendszergazdák mindegyikre folyamatosan odafigyelni.
- Nő a „monitoring”, mint tevékenységi kör jelentősége.
- A jó rendszereket nem elsősorban kívülről figyelik („polling”), hanem azok saját magukat figyelik belülről, és riasztanak minket, ha baj van. Erre még nincs egységes szóhasználat, de gyakran „Server Generated Alert System” a neve.
- Az igazán jó rendszerek nemcsak riasztanak, hanem megoldást is javasolnak, sőt esetleg a javasolt változtatást meg is teszik automatikusan.

26. A szoftver fejlődésének a képessége és üteme

- Az igények folyamatosan bővülnek és változnak.
- Azt a szoftvert el kell kerülni, amelynek a gyártója nem fejleszti a terméket nagy tempóban.
- Egyes szoftverek nagyon gyenge architektúrális alapokra épültek, és ezért nem tudnak továbbfejlődni.
- Másik probléma lehet, ha a kód már ősrégi, és most már csak nehezen karbantartható.
- Fontos kérdés, hogy milyen programozási nyelvet használtak az adatbáziskezelő rendszer gyártói.
- E téren pl. az Oracle C-ben íródott, ami ma már inkább rossz tulajdonság, mint jó, és a kód jelentős része igen „dohos”. Ezt próbálják ellensúlyozni azzal, hogy rengetegen fejlesztik.

27. A szoftvertámogatás minősége

- „Bugs are facts of life” – úgy tűnik, hogy sajnos elkerülhetetlenek.
- Ha már vannak hibák, kritikus kérdés, hogy van-e aki javítsa őket?
- Hányan és hány problémát oldanak meg? A mi problémánkat jellemzően megoldják-e, és ha igen, mennyi idő alatt?
- A megoldási módszer is lényeges, hiszen érzékeny adatokat tárolunk. Ki férhet hozzá a hiba felderítése során?
- A „support” általában borsos áron történik.
- Jellemzően nemcsak a hibajavításokat, hanem az új verziókat is fedi a support-díj.
- A támogatás nemcsak a téves kódra, hanem a téves használatra is kiterjed-e vajon?

28. Feltörési lehetőségek, adatlopások

- Minden adatlopás egy tragédia. Sok esetben a következmény a cég megszűnése.
- Nem létezik 100%-os biztonság.
- Mégis érdemes felmérni a választás előtt, hogy az adott szoftvert milyen gyakran törik fel? Hány incidens került nyilvánosságra az elmúlt években?
- Hogyan reagál a szoftvergyártó, ha kiderül 1-1 sebezhetőség?
- Megtesz-e a gyártó minden tőle telhetőt ahhoz, hogy megelőzze az újabb sebezhetőségek kialakulását.
- Csupán „Denial-Of-Service” típusú sérülékenységekről beszélünk, vagy komoly adatlopások, adatmódosítások is történhetnek (pl. „SQL Injection”)?

29. Mennyire védettek az adatok a DBA-tól

- Egy komoly és eldöntendő kérdés, hogy veszélyforrásként tekint-e a cég a rendszergazdákra, vagy nem.
- Ha félünk a DBA-k és az operációs rendszerek rendszergazdáinak a jogosultságaitól, akkor vajon létezik-e olyan szoftververzió, ahol a DBA és a rendszergazda sem férhetnek hozzá az adatokhoz?
- Pl. az Oracle adatbázis esetén ezt a megoldást Oracle Database Vaultnak nevezzük.

30. És még sok egyéb szempont is felmerülhet

- Az eddigiekben felsorolt kritériumok nem egy végleges listát alkotnak.
- Az évek során és a technológia fejlődésével állandóan új szempontok merülnek fel.
- Az látszik, hogy a „melyik a jobb adatbátiskezelő rendszer” kérdés nagyon sokrétű, komplex.

Rövid kitekintés a NoSQL adatbázisok világába

Milyen volt a múlt és mik a jelen prolémái

- A relációs adatbázisok előtt is volt élet. Sőt adatbázisok is voltak:
  - hierarchikusak
  - hálósok
  - egyéb…
- A relációsok az 1970-es évektől 2000-ig egyértelműen domináltak
- Időközben objektumorientált adatbázisok, OLAP adatbázisok:csekély siker
- Kezdetben nagygépes környezetre tervezték a relációs adatbázisokat
- Jól struktúrált adatokat képzeltek el az adatbázisokban
- Elsősorban OLTP típusú rendszerek: indexek, tranzakciókezelés, lockolás
- Később adattárházak is: párhuzamosítás,
- Később „commodity” hardver: sok kisgép.
- Megjelenik a clusterezés, adatok particionálása, ami új problémákat vet fel
- A transzparencia nem sikerül teljesen

A nyolc téves feltételezés

1. A hálózat megbízható
2. A hálózati késleltetés nulla
3. A sávszélesség végtelen
4. A hálózat mebízható
5. A topológia nem változik
6. Egy rendszergazda adminisztrtálja
7. Az adatátvitel költsége 0
8. A hálózat homogén

Paradigmaváltás: a régi rendszerek igyekeztek transzparenssé tenni a sokgépes adattárolási és feldolgozási modellt, a NoSQL adatbázisok nem teszik ezt. Ehelyett az alkalmazás ismerje az adatok elosztott mivoltát és használja ezt a tudást

A System R és az ebből fakadó „utódok” architekturális jellemzői

- Merevlemez-orientált tárolás
- Egyidejűleg több szál feldolgozása (multithraeding)
- zárolások a konkurrenciakezelés érdekében
- Napló-alapú helyreállítás (log based recovery)

A H-Store prototípus és tudományos következményei

- Az MIT-n (Stonebraker és mások) a modern hardverek új lehetőségeire és számos új szoftverötletre építve készítettek egy prototípust: H-Store néven, ahol a sebesség volt a fő célkitűzés
- Még a TPC-C benchmarkban is 82-szeres sebességnövekedést értek el

- Ebből azt a következtetést vonták le, hogy specializált adatbáziskezelő rendszerek kellenek minimum a következő 5 részterület számára:
  - Adattárházak
  - Stream processing: hierarchikus adatmodellek felé elmozdulás
  - Text processing: sosem működött jól relációs adatmodellel
  - Tudományos-kutatás orientált adatbázisok: tömbök a táblák helyett
  - Félig strukturált adatok: pl. XML adatbázisok

A NoSQL mozgalom

- Először azt jelentette ez, hogy: No SQL
- Később lett Not Only SQL ☺
- Hatalmas méretű adat esetén nem skálázódnak elég jól a relációs SQL-alapú rendszerek
- Pl. IWIW: Oracle relációs adatbázisra épült, míg a Facebook NoSQL-re
- A BigData világban olyan rendszerek kellenek, amelyek olcsó gépeken futnak és szinte végtelenül skálázhatóak
- Az adatoknak valamiféle particionálása történik
- ACID kontra BASE:
  - Basically Available
  - Soft state
  - Eventual consistency

NoSQL adatbázisok

Key-Value Cache

- Apache Ignite, Coherence, Hazelcast, ...

Key-Value Store (AP/EC)

- Amazon Dynamo, Riak, Oracle NoSQL, Voldemort, ...

Key-Value Store (Ordered)

- FoundationDB, InfinityDB, MemCacheDB, …

Document Store

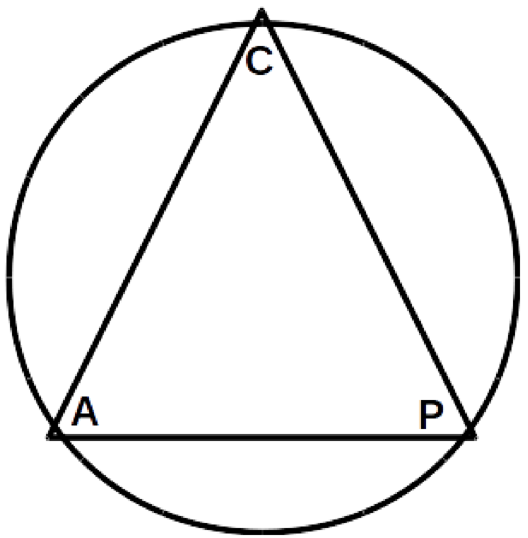
- ArangoDB, BaseX, Couchbase, CouchDB, MongoDB, IBM Domino, ...

Wide Column Store

- Amazon DynamoDB, Google Bigtable, Cassandra, Druid, HBase, Hypertable

Source: http://www.christof-strauch.de/nosqldbs.pdf

CAP-tétel



Elosztott rendszerekben… válassz maximum kettőt

- Consistency: egy igazság mindenhol
- Availability: mindig elérhető
- Partition-tolerance: működik akkor is, ha nem minden gép elérhető

CA: relational
CP: HBase, MongoDB, BigTable…
AP: Cassandra, Amazon Dynamo