

# MySQL and PostgreSQL

Lab notes for the course “Basi Dati e Sistemi  
Informativi”

lectured by Prof. Gabriel Kuper

Lab assist. Ilya Zaihrayeu

Lab notes are available from:  
[http://www.dit.unitn.it/~ilya/sql\\_labs.pdf](http://www.dit.unitn.it/~ilya/sql_labs.pdf)



# What is MySQL?

- MySQL is a relational database management system
- A relational database stores data in separate tables rather than putting all the data in one big storeroom
  - adds speed and flexibility
  - tables are linked by defined relations making it possible to combine data from several tables on request
- The SQL part of "MySQL" stands for "Structured Query Language"—the most common standardised language used to access relational databases



# Getting Started

- Log in to the system using your 'kirk' account
- Run telnet and connect to [vulcan.science.unitn.it](http://vulcan.science.unitn.it)
- Run `mysql -u userNN -p` from the command prompt, where userNN is your MySQL user ID (given by lab assistant)
- When 'password' prompt appears, enter your password, which is the same as your ID, i.e. 'userNN'



4

# Create Database

- Once connected, you can create your database to work with:
- `CREATE DATABASE DBNN;`
- Do not forget to use capitalized letters and assigned to you number NN in the name of the database!
- E.g. 

```
mysql> CREATE DATABASE DB46;  
Query OK, 1 row affected (0.01 sec)
```



# Create Database, syntax

- CREATE DATABASE [IF NOT EXISTS] db\_name
- Creates a database with the given name
- An error occurs if the database already exists and you didn't specify IF NOT EXISTS
- Databases in MySQL are implemented as directories containing files that correspond to tables in the database
- Because there are no tables in a database when it is initially created, the CREATE DATABASE statement only creates a directory under the MySQL data directory



# Show Statement

- You can see the list of available databases now by means of the `SHOW DATABASES` statement
- Analogously you can see tables in the databases, columns of those tables, etc.

○ E.g.

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| DB46     |
| test     |
+-----+
2 rows in set (0.03 sec)
```



7

# Show Statement, syntax

SHOW DATABASES [LIKE wild]  
or SHOW [OPEN] TABLES [FROM db\_name] [LIKE wild]  
or SHOW [FULL] COLUMNS FROM tbl\_name [FROM db\_name] [LIKE wild]  
or SHOW INDEX FROM tbl\_name [FROM db\_name]  
or SHOW TABLE STATUS [FROM db\_name] [LIKE wild]  
or SHOW STATUS [LIKE wild]  
or SHOW VARIABLES [LIKE wild]  
or SHOW LOGS  
or SHOW [FULL] PROCESSLIST  
or SHOW GRANTS FOR user  
or SHOW CREATE TABLE table\_name  
or SHOW MASTER STATUS  
or SHOW MASTER LOGS  
or SHOW SLAVE STATUS



# Select your database

- In order to use your database, i.e. create tables, manipulate data, etc you should activate it:
- `\u [DBNN]` or `use [DBNN]`
- After the command is successfully executed, all SQL commands relate to objects in the chosen database
- E.g. 

```
mysql> use DB46
Database changed
```



9

# Create Table

- You can create tables in your database by means of **CREATE TABLE** statement
- Create example tables as they are given in the lecture notes:

```
CREATE TABLE Movie (title TEXT, director  
TEXT, actor TEXT);
```

```
CREATE TABLE Schedule (theater TEXT, title  
TEXT);
```

Field name

Data type

# Create Table, cont'd

- As the result of the operation, you should see one of the following results:

```
mysql> CREATE TABLE Movie (title TEXT, director TEXT, actor TEXT);  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> CREATE TABLE Schedule (theater TEXT, title TEXT);  
Query OK, 0 rows affected (0.00 sec)
```

Tables are created

```
mysql> CREATE TABLE Schedule (theater TEXT, title TEXT);  
ERROR 1050: Table 'Schedule' already exists  
mysql> _
```

Table Name conflicts

# Create Table, cont'd

- Now you can see tables, present in your database and their structure:

```
mysql> show tables from test;
+-----+
| Tables_in_test |
+-----+
| Movie          |
| Schedule       |
+-----+
2 rows in set (0.00 sec)
```

To see what tables you have:  
**SHOW TABLES FROM [database name]**

```
mysql> show fields from Movie;
+-----+-----+-----+-----+-----+-----+
| Field      | Type  | Null  | Key  | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| title      | text  | YES   |      | NULL    |      |
| director   | text  | YES   |      | NULL    |      |
| actor      | text  | YES   |      | NULL    |      |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

To see table information:  
**SHOW FIELDS FROM [table name]**



# Create Table, syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)] [table_options] [select_statement]
create_definition: col_name type [NOT NULL | NULL] [DEFAULT
    default_value] [AUTO_INCREMENT] [PRIMARY KEY]
    [reference_definition]
or PRIMARY KEY (index_col_name,...) or KEY [index_name]
    (index_col_name,...)
or INDEX [index_name] (index_col_name,...)
or UNIQUE [INDEX] [index_name] (index_col_name,...)
or FULLTEXT [INDEX] [index_name] (index_col_name,...)
or [CONSTRAINT symbol] FOREIGN KEY [index_name]
    (index_col_name,...) [reference_definition]
or CHECK (expr)
```



# Create Table, syntax cont'd

- Type:

type:

INTEGER[(length)] [UNSIGNED] [ZEROFILL]

or REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]

or DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]

or FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]

or CHAR(length) [BINARY]

or VARCHAR(length) [BINARY]

or DATE

or TIME

or DATETIME

or TEXT

or ... *many other types*

# Create Table, syntax cont'd

- `CREATE TABLE` creates a table with the given name in the current database
- An error occurs if there is no current database or if the table already exists
- In MySQL Version 3.23, you can use the `TEMPORARY` keyword when you create a table. A temporary table will automatically be deleted if a connection dies
- In MySQL Version 3.23 or later, you can use the keywords `IF NOT EXISTS` so that an error does not occur if the table already exists

# Insert Statement

- After the tables are created, you can input data into them using **INSERT** statement
- In our example you can do it in two ways:
  - I) **INSERT INTO Movie (title, director, actor) VALUES ('Shining', 'Kubrick', 'Nicholson');**
  - II) **INSERT INTO Movie VALUES ('Shining', 'Kubrick', 'Nicholson');**
- In (I) some fields and corresponding values may be omitted
- In (II) you have to list values for fields in the order fields are put in the table



# Insert Statement, syntax

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
  [INTO] tbl_name [(col_name,...)] VALUES
  ((expression | DEFAULT),...),(...),...
```

or

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
  [INTO] tbl_name [(col_name,...)] SELECT ...
```

or

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
  [INTO] tbl_name SET col_name=(expression |
  DEFAULT), ...
```



# Insert Statement, syntax cont'd

- `INSERT` inserts new rows into an existing table
- `INSERT ... VALUES` form of the statement inserts rows based on explicitly specified values
- `INSERT ... SELECT` form inserts rows selected from another table or tables
- `INSERT ... VALUES` form with multiple value lists is supported in MySQL Version 3.22.5 or later
- `col_name=expression` syntax is supported in MySQL Version 3.22.10 or later

# Insert Statement, input data

- Recall the (pared-down) syntax: `INSERT INTO tbl_name [(col_name,...)] VALUES ((expression),...);`
- Input in `Movie` and `Schedule` relations the following data

Movie	title	director	actor
	Shining	Kubrick	Nicholson
	Player	Altman	Robbins
	Chinatown	Polanski	Nicholson
	Chinatown	Polanski	Polanski
	Repulsion	Polanski	Deneuve

Schedule	theater	title
	Le Champo	Shining
	Le Champo	Chinatown
	Le Champo	Player
	Odéon	Chinatown
	Odéon	Repulsion

- To see what data you have type: `SELECT * from tbl_name;`
- Try to avoid imputing duplicate rows!

# Insert Statement, input data cont'd

- Results should be like:

INSERT  
with fields  
declaration

```
mysql> INSERT INTO Movie (title, director, actor) VALUES ('Player', 'Altman', 'Robbins');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Movie (title, director, actor) VALUES ('Chinatown', 'Polanski', 'Nicholson');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Movie (title, director, actor) VALUES ('Chinatown', 'Polanski', 'Polanski');
Query OK, 1 row affected (0.00 sec)
```

Pared-down INSERT syntax

```
mysql> INSERT INTO Movie VALUES ('Repulsion', 'Polanski', 'Deneuve');
Query OK, 1 row affected (0.00 sec)
```



# Load Data Infile

- Syntax:

```
LOAD DATA [LOW_PRIORITY |  
CONCURRENT] [LOCAL] INFILE  
'file_name.txt' [REPLACE | IGNORE]  
INTO TABLE tbl_name  
[FIELDS [TERMINATED BY '\t']  
[[OPTIONALLY] ENCLOSED BY '"']  
[ESCAPED BY '\\' ] ]  
[LINES TERMINATED BY '\n'] [IGNORE  
number LINES] [(col_name,...)]
```



## Load Data Infile, cont'd

- `LOAD DATA INFILE` statement reads rows from a text file into a table at a very high speed
- If the `LOCAL` keyword is specified, the file is read from the client host
- If `LOCAL` is not specified, the file must be located on the server
- When reading text files located on the server, the files must either reside in the database directory or be readable by all

# Select Statement

- `SELECT` statement is used to retrieve data from specified tables what satisfy certain criteria, for instance:
- “Show titles of all movies present in the database”:

```
SELECT title FROM Movie;
```

```
mysql> SELECT title FROM Movie;
+-----+
| title |
+-----+
| Shining |
| Player |
| Chinatown |
| Chinatown |
| Repulsion |
+-----+
5 rows in set (0.00 sec)
```

# Select Statement, cont'd

- “Find directors who acted in their movies”:

```
SELECT director
FROM Movie
WHERE director = actor;
```

```
mysql> SELECT director
-> FROM Movie
-> WHERE director = actor;
+-----+
| director |
+-----+
| Polanski |
+-----+
1 row in set <0.00 sec>
```

- “Show data stored in Schedule”:

```
mysql> SELECT * FROM Schedule;
+-----+-----+
| theater | title |
+-----+-----+
| Le Champo | Shining |
| Le Champo | Chinatown |
| Le Champo | Player |
| Odeon | Chinatown |
| Odeon | Repulsion |
+-----+-----+
5 rows in set <0.00 sec>
```

```
SELECT *
FROM Schedule;
```

- “\*” stands for ‘all columns’

# Select Statement, syntax

```
SELECT [STRAIGHT_JOIN] [SQL_SMALL_RESULT]
       [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
       [SQL_CACHE | SQL_NO_CACHE]
       [SQL_CALC_FOUND_ROWS] [HIGH_PRIORITY]
       [DISTINCT | DISTINCTROW | ALL]
select_expression,...
[INTO {OUTFILE | DUMPFILE} 'file_name' export_options]
[FROM table_references [WHERE where_definition]
 [GROUP BY {unsigned_integer | col_name | formula}
 [ASC | DESC], ... [HAVING where_definition] [ORDER BY
 {unsigned_integer | col_name | formula} [ASC | DESC] ,...]
 [LIMIT [offset,] rows] [PROCEDURE procedure_name]
 [FOR UPDATE | LOCK IN SHARE MODE]]
```

# Select Statement, syntax cont'd

- `SELECT` is used to retrieve rows selected from one or more tables
- `select_expression` indicates the columns you want to retrieve
- You can refer to a column as `col_name`, `tbl_name.col_name`, or `db_name.tbl_name.col_name`
- `SELECT` can be used for output of the results to a file

# Select Statement, syntax cont'd

- A table reference may be aliased using `tbl_name [AS] alias_name`, example:
- `SELECT M.title, S.theater FROM Movie AS M, Schedule AS S WHERE M.title = S.title;`
- Is equal to “List directors, and theaters in which their movies are playing”

```
mysql> SELECT M.director, S.theater FROM Movie AS M, Schedule AS S WHERE M.title=S.title;
+-----+-----+
| director | theater |
+-----+-----+
| Polanski | Odeon   |
| Polanski | Odeon   |
| Altman   | Le Champo |
| Polanski | Le Champo |
| Polanski | Le Champo |
| Kubrick  | Le Champo |
| Polanski | Odeon   |
+-----+-----+
7 rows in set (0.00 sec)
```

# Select Statement, examples

- “Find theaters showing movies directed by Polanski”:

```
SELECT Schedule.theater
FROM Schedule, Movie
WHERE
Movie.title = Schedule.title
AND
Movie.director = 'Polanski';
```

```
mysql> SELECT Schedule.theater
-> FROM Schedule, Movie
-> WHERE Movie.title = Schedule.title
-> AND Movie.director = 'Polanski';
+-----+
| theater |
+-----+
| Le Champo |
| Odeon |
| Le Champo |
| Odeon |
| Odeon |
+-----+
5 rows in set (0.02 sec)
```

# Select Statement, examples, cont'd

- “Find theaters showing movies featuring Nicholson”:

```
SELECT Schedule.theater
FROM Schedule, Movie
WHERE Movie.title =
       Schedule.title
AND Movie.actor =
       'Nicholson';
```

```
mysql> SELECT Schedule.theater
-> FROM Schedule, Movie
-> WHERE Movie.title = Schedule.title
-> AND Movie.actor = 'Nicholson';
+-----+
| theater |
+-----+
| Le Champo |
| Le Champo |
| Odeon     |
+-----+
3 rows in set (0.00 sec)
```

# Select Statement, examples, cont'd

- “Find actors who played in movies directed by Kubrick *or* Polanski”:

```
SELECT actor
```

```
FROM Movie
```

```
WHERE director = 'Kubrick' OR director = 'Polanski';
```

Nicholson  
played in  
movies of  
both Kubrick  
and Polanski

```
mysql> SELECT actor
-> FROM Movie
-> WHERE director = 'Kubrick' OR director = 'Polanski';
+-----+
| actor |
+-----+
| Nicholson |
| Nicholson |
| Polanski |
| Deneuve |
+-----+
4 rows in set (0.00 sec)
```

# Select Statement, examples, cont'd

- If you need to avoid duplicates in the results, use **DISTINCT**

```
SELECT DISTINCT actor
FROM Movie
WHERE director = 'Kubrick' OR director = 'Polanski';
```

Now  
Nicholson is  
reported only  
once

```
mysql> SELECT DISTINCT actor
-> FROM Movie
-> WHERE director = 'Kubrick' OR director = 'Polanski';
+-----+
| actor |
+-----+
| Nicholson |
| Polanski |
| Deneuve |
+-----+
3 rows in set (0.01 sec)
```

# Union Operator

- The “*Kubrick or Polanski*” example could be also done using **UNION** operator:

```
SELECT actor
FROM Movie
WHERE director = 'Kubrick'
UNION
SELECT actor
FROM Movie
WHERE director = 'Polanski'
```

```
mysql> SELECT actor
-> FROM Movie
-> WHERE director = 'Kubrick'
-> UNION
-> SELECT actor
-> FROM Movie
-> WHERE director = 'Polanski';
+-----+
| actor |
+-----+
| Nicholson |
| Polanski |
| Deneuve  |
+-----+
3 rows in set (0.00 sec)
```

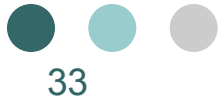


# Union Operator, syntax

```
SELECT ...  
UNION [ALL]  
SELECT ...
```

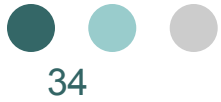
```
  [UNION  
  SELECT ...]
```

- UNION is used to combine result from many SELECT statements (even from different tables) into one result set
- The columns listed in the select\_expression portion of the SELECT should have the same type
- The column names used in the first SELECT query will be used as the column names for the results returned



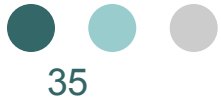
# MySQL Limitations

- Current version of MySQL **does not** support sub-queries, EXCEPT, INTERSECT, IN, NOT IN and some other operators that makes impossible to do more complicated and interesting queries
- Let us examine a more advanced RDBMS...



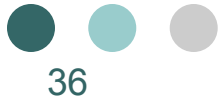
# PostgreSQL

- PostgreSQL is a sophisticated Object-Relational DBMS, supporting almost all SQL constructs, including subselects, transactions, and user-defined types and functions
- It is the most advanced open-source database available anywhere (source: [www.postgresql.org](http://www.postgresql.org))



# Connecting to PostgreSQL

- Connect to **kirk** server
- Run **PostgreSQL** client using:
  - `psql -U userNN`, where **NN** is your assigned number (you should be in directory `/local/pgsql/bin`)
  - When prompted for password enter `'user'`
- Now you are in the **userNN** database environment



# PostgreSQL, some hints

- `\?` – display help
- `\l` – list all databases
- `\d` – list tables
- `\h [cmd]` – help on syntax of SQL commands
- `\i <file>` - read and execute queries from <file>
- `\o [file]` – send all query results to [file]
- `\q` – quit psql

# PostgreSQL, input data

- Create tables `Movie` and `Schedule` using `CREATE TABLE ...`
- Enter data given bellow using `INSERT` statement

Movie	title	director	actor
	Shining	Kubrick	Nicholson
	Player	Altman	Robbins
	Chinatown	Polanski	Nicholson
	Chinatown	Polanski	Polanski
	Repulsion	Polanski	Deneuve

Schedule	theater	title
	Le Champo	Shining
	Le Champo	Chinatown
	Le Champo	Player
	Odéon	Chinatown
	Odéon	Repulsion

- Check content of tables by means of `SELECT * FROM <table_name>;`



# Alter Table Statement

- `ALTER TABLE` allows to change the structure of an existing table, e.g.
  - add columns
  - create or destroy indexes (will come later)
  - change the type of existing columns
  - rename columns or the table itself

# Alter Table Statement, syntax

```
ALTER TABLE [ ONLY ] table [ * ]  
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]  
ALTER TABLE [ ONLY ] table [ * ]  
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }  
ALTER TABLE [ ONLY ] table [ * ]  
    ALTER [ COLUMN ] column SET STATISTICS integer  
ALTER TABLE [ ONLY ] table [ * ]  
    RENAME [ COLUMN ] column TO newcolumn  
ALTER TABLE table  
    RENAME TO new_table  
ALTER TABLE table  
    ADD table_constraint_definition  
ALTER TABLE [ ONLY ] table  
    DROP CONSTRAINT constraint { RESTRICT | CASCADE }  
ALTER TABLE table  
    OWNER TO new_owner
```



# Alter Table, Compatibility

SQL92 specifies some additional capabilities for `ALTER TABLE` statement which are not yet directly supported by PostgreSQL:

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT | CASCADE }
```

Removes a column from a table. Currently, to remove an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT did, city FROM distributors;  
DROP TABLE distributors;  
CREATE TABLE distributors (  
    did    DECIMAL(3) DEFAULT 1,  
    name   VARCHAR(40) NOT NULL);
```

```
INSERT INTO distributors SELECT * FROM temp;  
DROP TABLE temp;
```

# Alter Table Statement, example

- Let us add 'year' field to the Movie relation:
- ALTER TABLE Movie ADD COLUMN year INTEGER;

```
ilya=> ALTER TABLE Movie ADD COLUMN year INTEGER;  
ALTER
```

```
ilya=> SELECT * FROM Movie;  
title | director | actor | year  
-----+-----+-----+-----  
Shining | Kubrick | Nicholson |  
Player | Altman | Robbins |  
Chinatown | Polanski | Nicholson |  
Chinatown | Polanski | Polanski |  
Repulsion | Polanski | Deneuve |  
<5 rows>
```

This is how  
Movie relation  
looks now

# Alter Table Statement, example cont'd

- In the previous example values from newly-created field 'year' are undefined, i.e. set to 'NULL'
- [The NULL value means "no data" and is different from values such as 0 for numeric types or the empty string for string types]
- Let us add year values to the table!

# Update Statement

- Years can be added using UPDATE statement, e.g.

Relation name

Field to be updated and value

- UPDATE Movie SET year=1974 WHERE title='Chinatown';

Update condition

- The result of this statement should be:

```
ilya=> UPDATE Movie SET year=1974 WHERE title='Chinatown';  
UPDATE 2
```

Mention, that 2 rows matched

# Update Statement, syntax

```
UPDATE [ ONLY ] table SET col = expression [, ...]  
  [ FROM fromlist ]  
  [ WHERE condition ]
```

- **UPDATE** updates columns in existing table rows with new values
- **SET** clause indicates which columns to modify and the values they should be given
- **WHERE** clause, if given, specifies which rows should be updated. Otherwise, all rows are updated

# Update Statement, input data

- Recall the syntax:

```
UPDATE Movie SET year=1974 WHERE title='Chinatown';
```

- Add the following years to the Movie table:
  - The Shining: 1980
  - The Player: 1992
  - Chinatown: 1974
  - Repulsion: 1965.

```
ilya=> select * from Movie;
title | director | actor | year
-----+-----+-----+-----
Chinatown | Polanski | Nicholson | 1974
Chinatown | Polanski | Polanski | 1974
Repulsion | Polanski | Deneuve | 1965
Player | Altman | Robbins | 1992
Shining | Kubrick | Nicholson | 1980
(5 rows)
```

Movie table  
after all years  
are added

# Delete Statement

- DELETE statement is used when it is necessary to delete rows from a table, e.g.
- Delete row ('Chinatown', 'Polanski', 'Polanski', 1974) from Movie table:
- DELETE FROM Movie WHERE title = 'Chinatown' AND actor = 'Polanski';

```
ilya=> DELETE FROM Movie WHERE title = 'Chinatown' AND actor = 'Polanski';  
DELETE 1
```

```
ilya=> select * from Movie;  
title | director | actor | year  
-----+-----+-----+-----  
Chinatown | Polanski | Nicholson | 1974  
Repulsion | Polanski | Deneuve | 1965  
Player | Altman | Robbins | 1992  
Shining | Kubrick | Nicholson | 1980  
(4 rows)
```

Movie table  
after the  
DELETE  
operation



# Delete Statement, syntax

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

- `DELETE` deletes rows from `table` that satisfy `condition`, and returns the number of records deleted
- If `WHERE` clause is omitted, all rows are deleted
- Be sure to specify explicitly the condition in the `WHERE` clause to avoid undesirable data loss



# Drop Table

- Syntax:

`DROP TABLE name [, ...]`

- `DROP TABLE` removes one or more tables
- All table data and the table definition are *removed*, so **be careful** with this command!



# Drop Database

- Syntax:

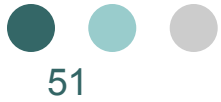
`DROP DATABASE name`

- `DROP DATABASE` drops all tables in the database and deletes the database
- Be **VERY** careful with this command!
- [You will not be able to delete your databases due to privileges restrictions..]



# Exercises

1. “Find natural join of Movie and Schedule”
2. “Find theatres showing Polanski’s Movies” [nested queries, IN]
3. “Find all actors who are not directors” [EXCEPT]
4. “Find all actors who are also directors” [INTERSECT]
5. “Find directors whose movies are playing in all theatres” [NOT EXISTS, EXCEPT]
6. “Find directors whose movies are playing at Le Champo” [EXISTS]
7. “Find directors whose movies are playing at Le Champo” [IN]
8. “Find actors who did not play in a movie by Kubrick” [EXCEPT, NOT IN]



# Indexes, Introduction

- Indexes are primarily used to enhance database performance (faster searches, for example)
- But inappropriate use will result in slower performance
- Indexes are also used for duplicates control
- Indexes can be manipulated using such statements as `CREATE TABLE`, `ALTER TABLE`, `CREATE INDEX` or `DROP INDEX`

# Create Index, Syntax

```
CREATE [ UNIQUE ] INDEX index_name ON table
  [ USING acc_method ] ( column [ ops_name ] [, ...] )
  [ WHERE predicate ]
```

```
CREATE [ UNIQUE ] INDEX index_name ON table
  [ USING acc_method ] ( func_name( column [, ... ] ) [ ops_name ] )
  [ WHERE predicate ]
```

- When the **WHERE** clause is present, a *partial index* is created (a partial index is an index that contains entries for only a portion of a table)
- If **UNIQUE** is specified, it causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added
  - Attempts to insert or update data which would result in duplicate entries will generate an error



# Indexes, example

- Consider the Movie relation again
- If we want, for instance, rows to be uniquely identified by combination of all fields, i.e. avoid duplicates in the table, we might use:
- CREATE UNIQUE INDEX table\_record ON Movie (title, director, actor, year);

```
ilya=> CREATE UNIQUE INDEX table_record ON movie (title, director, actor, year);  
CREATE
```

## Indexes, example cont'd

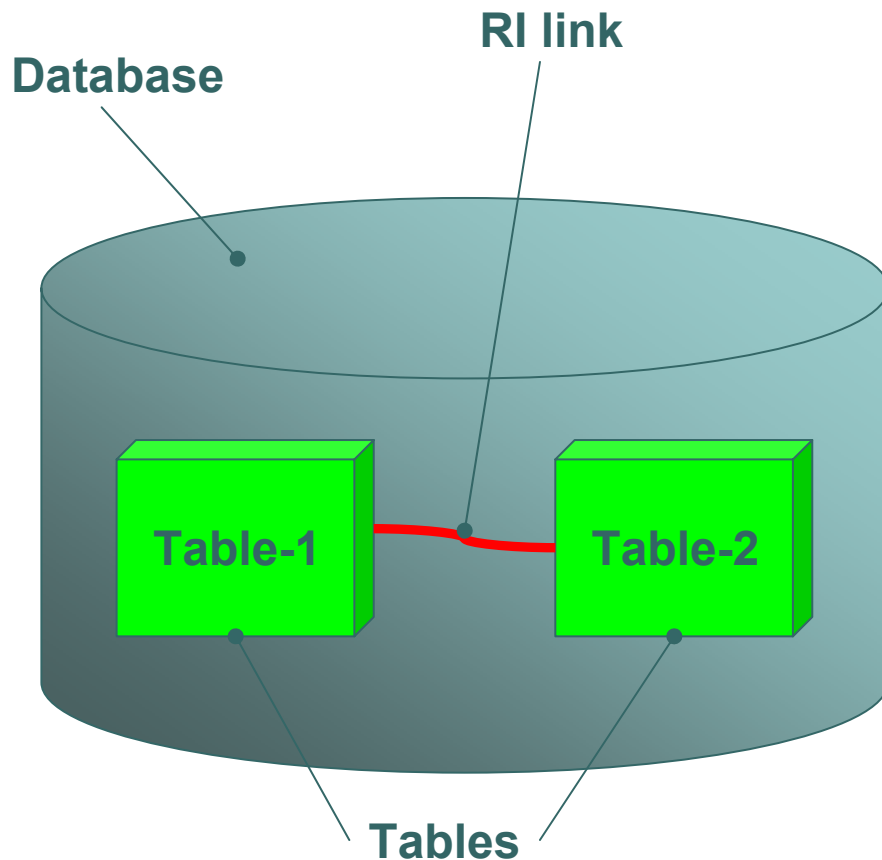
- Now, if we try to add a new record that already exists in the table, we get an error message:

```
ilya=> INSERT INTO Movie VALUES ('Shining', 'Kubrick', 'Nicholson', 1980);  
ERROR: Cannot insert a duplicate key into unique index table_record
```

- To remove the index use `DROP INDEX table_record;`

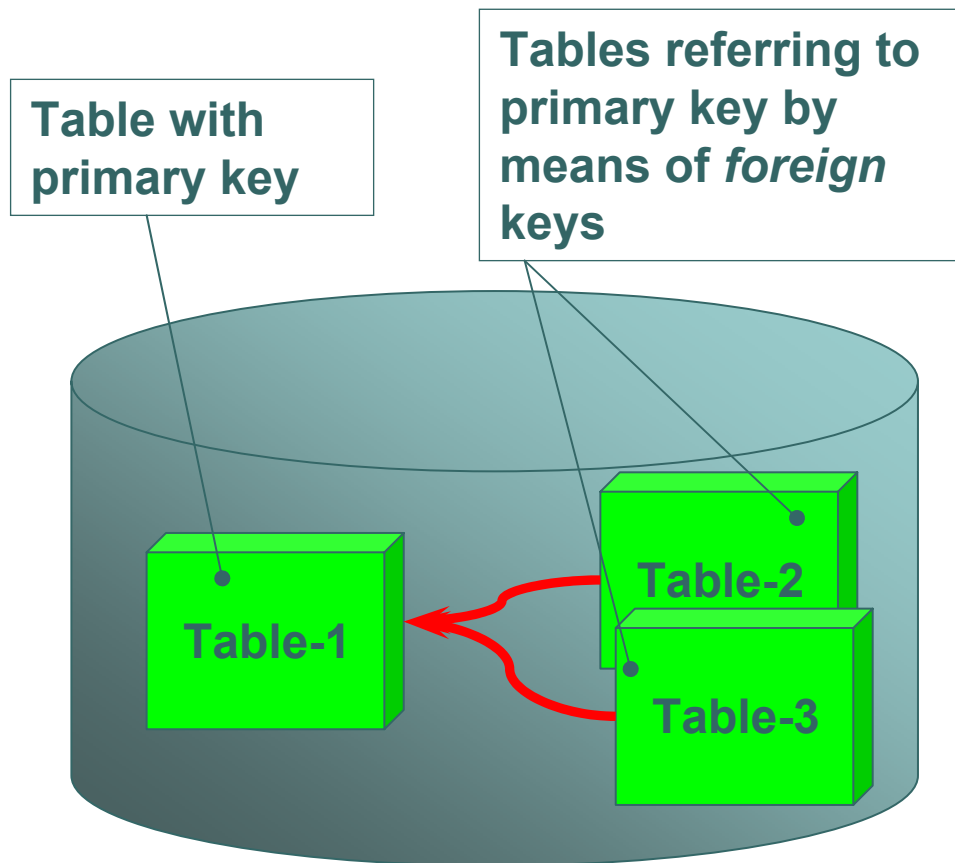
```
ilya=> DROP INDEX table_record;  
DROP
```

# Referential integrity



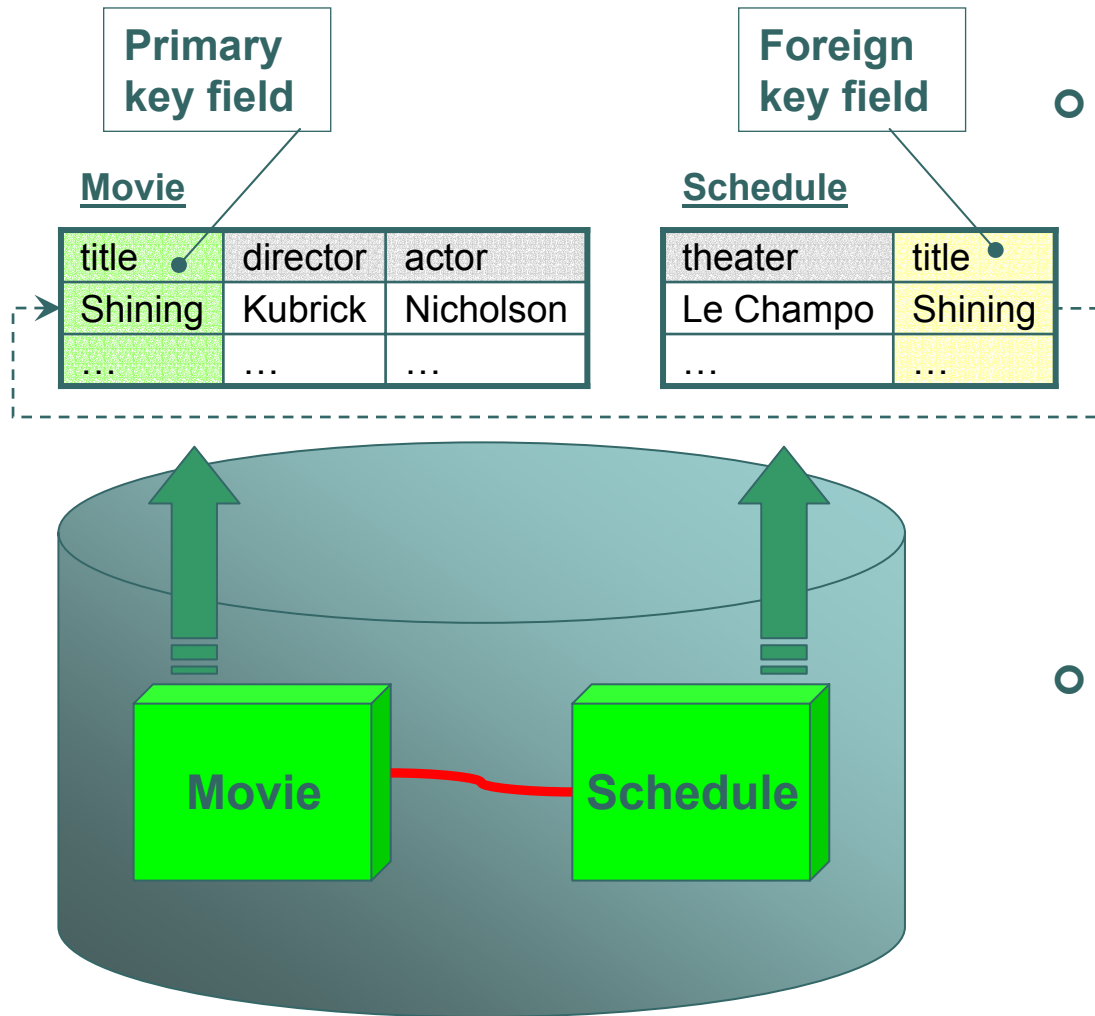
- Referential integrity (RI) is where information in one table refers to information in another table and the database enforces the relationship
- RI is used where you have information in at least one table which must refer to information in another table, and there must be no possibility of them being out of sync.
- Database can also be instructed to automatically take care of updating and removing linked entities in the tables

# Referential integrity, cont'd



- The field of a table created for other tables to refer to is called the *primary* key
- Any field in any other table can refer to this primary key field, as long as it is of the same type as the primary key. This field is called the *foreign* key

# Referential integrity, cont'd



- For example, with RI properly in place for Movie and Schedule tables, you can make sure that for each record for movie *M* in Schedule, at least one record of *M* exists in Movie
- RI link makes impossible to insert a non existing in table Movie movie into Schedule

# Referential integrity, example

- Movie table, *primary key*
  - *Delete duplicates in 'title' field, if any*
  - *Create temporary table:*  
*CREATE TABLE Temp (title TEXT PRIMARY KEY, director TEXT, actor TEXT, year INTEGER);*
  - *Copy data from Movie to Temp:*  
*INSERT INTO Temp SELECT \* FROM Movie*
  - *Delete Movie: DROP TABLE Movie;*
  - *Rename Temp into Movie: ALTER TABLE Temp RENAME TO Movie;*

# Referential integrity, example

- Schedule table, *foreign key*
  - Add a foreign key constraint:  
`ALTER TABLE Schedule ADD CONSTRAINT  
mov_link FOREIGN KEY (title) REFERENCES  
Movie (title);`
- Now, if you try to add a record into  
Schedule with 'non-existing' movie:

```
ilya=> INSERT INTO Schedule VALUES ('Odeon', 'Mission impossible');  
ERROR:  mov_link referential integrity violation - key referenced from schedule  
not found in movie
```



# Referential integrity, example

- An error occurs if you try to delete record for 'Player' from Movie – a record for 'Player' in Schedule still exists!

```
ilya=> DELETE FROM Movie WHERE title='Player';  
ERROR: mov_link referential integrity violation - key in movie still referenced  
from schedule
```

- But, if you delete first from Schedule, and after from Movie – no errors occur:

```
ilya=> DELETE FROM Schedule WHERE title='Player';  
DELETE 1  
ilya=> DELETE FROM Movie WHERE title='Player';  
DELETE 1
```

# Referential integrity, options

- Recall the ALTER statement for foreign key definition: `ALTER TABLE Schedule ADD CONSTRAINT mov_link FOREIGN KEY (title) REFERENCES Movie (title);`
- The REFERENCES keyword can be also followed by: `[ON DELETE action] [ON UPDATE action]`
- It is possible to specify the behaviour of database on DELETE and UPDATE statements when referential integrity restrictions are involved

# Referential integrity, options cont'd

**Action** can be:

- NO ACTION (*default action*)
  - Produce an error indicating that the deletion or update would create a foreign key constraint violation.
- RESTRICT
  - Same as NO ACTION.
- CASCADE
  - Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.
- SET NULL
  - Set the referencing column values to NULL.
- SET DEFAULT
  - Set the referencing column values to their default value.

# Referential integrity, options example

- On delete in Movie delete referencing rows in Schedule. On update in Movie, report an error:

```
ilya=> ALTER TABLE Schedule ADD CONSTRAINT mov_link FOREIGN KEY (title) REFEREN  
CES Movie(title) ON DELETE CASCADE ON UPDATE RESTRICT;  
NOTICE: ALTER TABLE ... ADD CONSTRAINT will create implicit trigger(s) for FORE  
IGN KEY check(s)  
CREATE
```

Delete:

```
ilya=> DELETE FROM Movie WHERE title='Player';  
DELETE 1  
ilya=> SELECT * FROM Schedule;  
 theater | title  
-----+-----  
 Le Champo | Shining  
 Le Champo | Chinatown  
 Odeon | Chinatown  
 Odeon | Repulsion  
<4 rows>
```

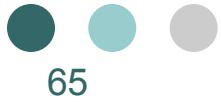
Update:

```
ilya=> UPDATE Movie SET title='Shining-2' WHERE title='Shining';  
ERROR: mov_link referential integrity violation - key in movie still referenced  
from schedule
```



# Primary and foreign keys, resume

- The *primary key* of a table is the column whose values are different in every row. Because they are different, they make each row unique. If no one such column exists, the primary key is a *composite* of two or more columns whose values, taken together, are different in every row.
- A *foreign key* is simply a column or group of columns in one table that contains values that match the *primary key* in another table



# Aggregate functions

- Oftentimes, we're also interested in summarizing our data to determine trends or produce top-level reports (overall sales volume, for example)
- SQL provides **aggregate functions** to assist with the summarization of large volumes of data
- Let's see how it works

## Aggregate functions, cont'd

- Create an integer field in the Movie table that will store duration of movies in minutes
  - ALTER TABLE Movie ADD COLUMN duration INTEGER;
- Add durations, e.g.: Player: 106; Shining: 180; Chinatown: 92; Repulsion: 120.
  - E.g. UPDATE Movie SET duration=106 WHERE title='Player';



# Aggregate functions, cont'd

- SUM ()
  - returns the summation of a series of values or expressions
- AVG ()
  - provides the mathematical average of a series of values or expressions
- MAX()
  - returns the largest value of the expression in a given data series
- MIN()
  - returns the minimum value for the expression
- An *expression* can be either a column or mathematical expression that takes columns as arguments
  - E.g. `SELECT AVG(UnitPrice * Quantity) As AveragePrice`

# Aggregate functions, cont'd

- COUNT
  - retrieves the number of records in a table that meet given criteria
- E.g. “How many movies have duration longer or equal than 120 minutes”:

```
SELECT COUNT (*) AS duration FROM Movie WHERE duration >=120;
```

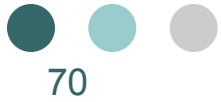
```
ilya=> SELECT COUNT (*) AS duration FROM Movie
ilya-> WHERE duration >=120;
duration
-----
                2
<1 row>
```

- Another example: “Find average duration of movies by Polanski”

```
ilya=> SELECT AVG (duration) AS Average_duration
ilya-> FROM Movie WHERE director='Polanski';
average_duration
-----
106.0000000000
<1 row>
```

# Aggregation and grouping

- GROUP BY expression, [, ...] clause in the SELECT statement
  - condenses into a single row all selected rows that share the same values for the grouped columns
  - aggregate functions, if any, are computed across all rows making up each group



# Aggregation and grouping, example

- For each director, return the average time of his/her movies:

```
SELECT director, AVG (duration) AS Average  
FROM Movie  
GROUP BY director
```

```
ilya=> SELECT director, AVG (duration) AS Average  
ilya-> FROM Movie  
ilya-> GROUP BY director;  
director | average  
-----+-----  
Altman   | 106.0000000000  
Kubrick  | 180.0000000000  
Polanski | 106.0000000000  
<3 rows>
```



# Selection based on aggregation results

- **HAVING** *boolean\_expr* clause in the SELECT statement
  - specifies a grouped table derived by the elimination of group rows that do not satisfy the *boolean\_expr*
  - **HAVING** is different from **WHERE**: **WHERE** filters individual rows before application of **GROUP BY**, while **HAVING** filters group rows created by **GROUP BY**

# Selection based on aggregation results, cont'd

- Find directors and average duration of their movies, provided they made at least one movie that is longer than 2 hours:

```
SELECT director, AVG (duration)
FROM Movie
GROUP BY director
HAVING MAX (duration) >120;
```

```
ilya=> SELECT director, AVG (duration)
ilya-> FROM Movie
ilya-> GROUP BY director
ilya-> HAVING MAX (duration) >120;
director |      avg
-----+-----
Kubrick  | 180.0000000000
<1 row>
```



# Aggregate functions, exercises

1. Find the average duration of all movies;
2. Find the longest (shortest) movie (s);
3. Find movies with the duration longer or equal than the average of all movies;
4. How many movies have duration shorter than the average duration of all movies?
5. List actors, and average duration of movies they played in and which are shown now in theatre (s);
6. For each theatre showing at least one movie that is longer than 2 hours, find the average length of movies playing there.

# Aggregate functions, exercises' answers

```
ilya=> SELECT AVG (duration) AS average_duration FROM Movie;
average_duration
-----
124.500000000000
(1 row)
```

Find the average duration of all movies

```
ilya=> SELECT title FROM Movie WHERE duration = (SELECT MAX(duration) FROM Movie);
title
-----
Shining
(1 row)
```

Find the longest movie (s)

```
ilya=> SELECT title FROM Movie WHERE duration >= (SELECT AVG (duration) FROM Movie);
title
-----
Shining
(1 row)
```

Find movies with the duration longer or equal than the average of all movies

# Aggregate functions, exercises' answers cont'd

```
ilya=> SELECT COUNT (*) AS Movie_number FROM Movie WHERE duration <=(SELECT AVG (duration) FROM Movie);
movie_number
-----
3
<1 row>
```

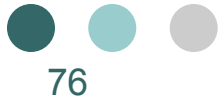
How many movies have duration shorter than the average duration of all movies?

```
ilya=> SELECT actor, AVG(duration) FROM Movie M, Schedule S
ilya-> WHERE M.title=S.title GROUP BY actor;
actor      |      avg
-----+-----
Deneuve    | 120.0000000000
Nicholson  | 136.0000000000
<2 rows>
```

List actors, and average duration of movies they played in and which are shown now in theatre (s)

```
ilya=> SELECT S.theater, AVG (M.duration)
ilya-> FROM Schedule S, Movie M
ilya-> WHERE S.title=M.title
ilya-> GROUP BY S.theater
ilya-> HAVING MAX (M.duration) >120;
theater    |      avg
-----+-----
Le Champo  | 136.0000000000
Odeon      | 130.6666666667
<2 rows>
```

For each theatre showing at least one movie that is longer than 2 hours, find the average length of movies playing there



# Views

- Views are intermediate query results
- Usually views are used when the result of a certain query is needed often
- Views are not physically materialized
- Instead, a query rewrite retrieve rule is automatically generated to support retrieve operations on views
- Syntax: `CREATE VIEW view [ ( column name list ) ] AS SELECT query`



# Views, example

- Suppose we need theatres, directors whose movies are playing there, and duration of those movies:

```
CREATE VIEW view1 (theater, director, duration) AS  
SELECT S.theater, M.director, M.duration  
FROM Movie M, Schedule S  
WHERE S.title=M.title;
```

```
ilya=> CREATE VIEW view1 (theater, director, duration) AS  
ilya-> SELECT S.theater, M.director, M.duration  
ilya-> FROM Movie M, Schedule S  
ilya-> WHERE S.title=M.title;  
CREATE
```

# Views, example cont'd

- Now it is possible to work with view1 as with a table, although in read-only mode

```
ilya=> SELECT * FROM view1;
 theater | director | duration
-----+-----+-----
 Le Champo | Polanski | 92
 Odeon    | Polanski | 92
 Odeon    | Polanski | 120
 Le Champo | Kubrick  | 180
<4 rows>
```

Show data referenced by the view

```
ilya=> SELECT theater, director
ilya-> FROM view1 WHERE duration >=120;
 theater | director
-----+-----
 Odeon    | Polanski
 Le Champo | Kubrick
<2 rows>
```

List theaters showing movies longer than 2 hours, and directors of those movies

```
ilya=> SELECT AVG (duration) AS Average_Dur
ilya-> FROM view1 WHERE director='Polanski';
 average_dur
-----
 101.3333333333
<1 row>
```

Find average duration of movies played in theaters directed by Polanski

## Views, example cont'd

- If, for example, you add a new row to Schedule table: ('Odeon', 'Shining')
- ...and query the view1 view: a new record corresponding to the currently playing in Odeon movie appears:

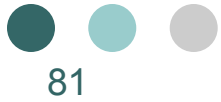
```
ilya=> select * from view1;
 theater | director | duration
-----+-----+-----
 Le Champo | Polanski | 92
 Odeon    | Polanski | 92
 Odeon    | Polanski | 120
 Le Champo | Kubrick  | 180
 Odeon    | Kubrick  | 180
 <5 rows>
```



## Views, cont'd

- After a view is no longer needed, it can be deleted by means of `DROP VIEW name [, ...]`

```
ilya=> DROP VIEW view1;  
DROP
```



# Transactions

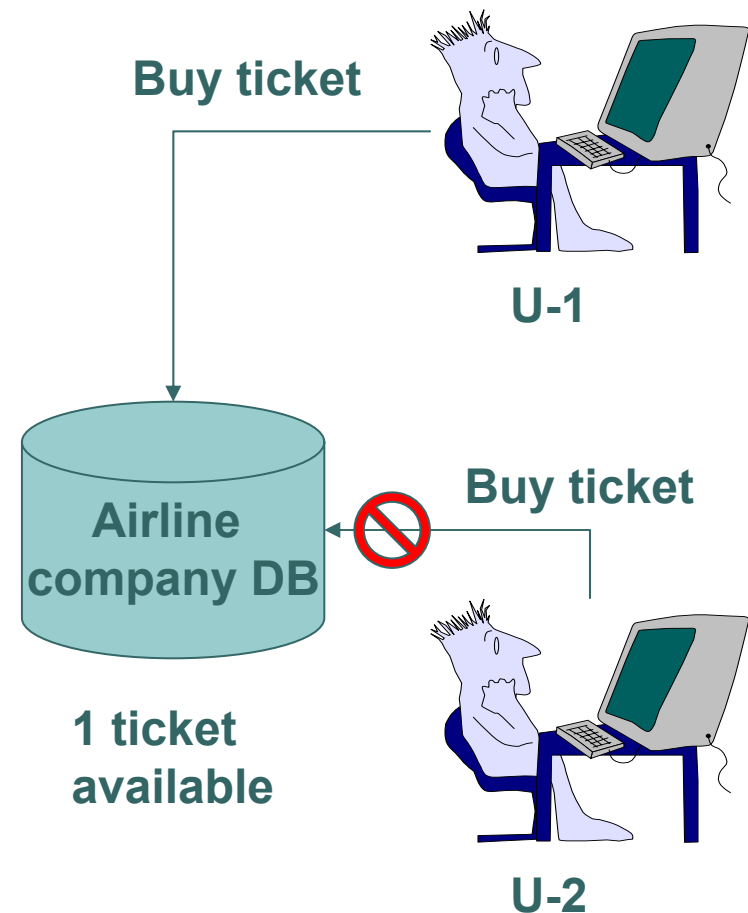
- A *transaction* is an execution of a user program, and is seen by the RDBMS as a series of actions which are reads and writes of database objects
- Examples:
  - Reserve an airline seat. Buy an airline ticket
  - Withdraw money from an ATM
  - Verify a credit card sale
  - Reserve a book in the library

# Concurrency control

- Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other
- Goal of concurrency control is to ensure that transactions execute *atomically*, i.e.:
  - each transaction accesses shared data without interfering with other transactions
  - if a transaction terminates normally, then all of its effects are made permanent; otherwise it has no effect at all
- One of the techniques used – set locks on shared data

# Example

- User U-1 and user U-2 both want to buy the (same) last ticket
- U-1 started ordering process before U-2
- Certain part of the DB is locked while system checks solvency of U-1, etc.
- Thus U-2 can not buy the same last ticket



# Concurrency control in PostgreSQL

- `BEGIN` - start a transaction block
- `COMMIT` - commit the current transaction
- `ROLLBACK` or `ABORT` - abort the current transaction
- Statements can optionally be followed by `[ WORK | TRANSACTION ]`, e.g. `ABORT TRANSACTION`

# Concurrency control in PostgreSQL, cont'd

- Within a transaction block it is possible to lock tables by means of:

```
LOCK [ TABLE ] name [, ...]
```

```
LOCK [ TABLE ] name [, ...] IN lockmode MODE
```

where lockmode is one of:

```
ACCESS SHARE | ROW SHARE | ROW  
EXCLUSIVE | SHARE UPDATE EXCLUSIVE |  
SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE |  
ACCESS EXCLUSIVE
```

# Concurrency control in PostgreSQL, cont'd

- EXCLUSIVE
  - An exclusive lock prevents other locks of the same type from being granted
- SHARE
  - A shared lock allows others to also hold the same type of lock, but prevents the corresponding EXCLUSIVE lock from being granted
- ACCESS
  - Locks table schema
- ROW
  - Locks individual rows



# Lock example

- Suppose we want to delete a movie  $\mathcal{M}$  from the **Movie** table and all its corresponding records from the **Schedule** table
- For this we should first delete all occurrences of  $\mathcal{M}$  from **Schedule** (i) and then delete it from **Movie** (ii)
- A problem appears if one inserts a record with  $\mathcal{M}$  to **Schedule** after (i) and before (ii)



## Lock example, cont'd

```
BEGIN WORK;
```

```
LOCK TABLE Schedule IN SHARE ROW  
EXCLUSIVE MODE;
```

```
DELETE FROM Schedule WHERE  
title='Shining';
```

```
DELETE FROM Movie WHERE title='Shining';
```

```
COMMIT WORK;
```



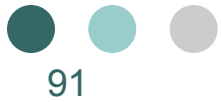
# Normal Forms

- Normal Forms defined in relational database theory represent guidelines for record design
- Each of the Forms are designed to logically address potential problems in referring to and working with information stored in a database
- A database is said to be in one of the Normal Forms if it satisfies the rules required by that Form
  - it also will not suffer from any of the problems addressed by the Form
- There are 5 Normal Forms in relational DB theory
- It is usually the case that a schema designer wants all relations created to be in as many normal forms as possible



# First Normal Form (1NF)

- A relation is in 1NF if every field value is non-decomposable (atomic) and every tuple is unique
- “Atom” comes from a Greek word that meant, essentially, the smallest possible piece of something
- In a database, this means that each column should only be designed to hold one and only one piece of information



# 1NF, example

**Bad**

Name	Address
Ben Goren	Post Office Box 964, Tempe, Arizona 85280- 0964
Jane Doe	1234 Main Street, Mesa, Arizona 85345

**Good**

First_Name	Last_Name	Street_Address	City	State	ZIP
Ben	Goren	Post Office Box 964	Tempe	Arizona	85280- 0964
Jane	Doe	1234 Main Street	Mesa	Arizona	85345

**If it is also necessary to store several addresses (e.g. residence address)?**

**One can find a solution – which is still incorrect – is to add columns. It may lead to Null values presence in your tables.**

**When you have Nulls, especially a lot of them, in your database, that should be a signal that you need to make certain that the database is in 1NF**

# 1NF, example, cont'd

- o A good solution to the problem might be:

Table: People			Table: Addresses					
ID*	First_Name	Last_Name	Person*->People:ID	Kind*	Address	City	State	ZIP
1	Ben	Goren						
2	Jane	Doe						
			1	Mailing	Post Office Box 964	Tempe	Arizona	85280-0964
			1	Residence	1331 West Third Street Apartment 2	Tempe	Arizona	85281
			2	Mailing	1234 Main Street	Mesa	Arizona	85345

\*Primary key

# Second Normal Form (2NF)

- A relation is in 2NF if and only if it is in 1NF and every non-key attribute is irreducibly dependent on the primary key
- The point of 2NF is to avoid the following:

ID	First_Name	Last_Name	Spouses_Birthday
3	John	Smith	02/07/65
4	Sue	Jones	07/26/76

- The problem is that the last column relates not to John Smith, but to another person entirely
- John has only one spouse, who only has one birthday...but that's too tenuous a relationship to embed into a single table

## 2NF, cont'd

- The decomposition technique for producing a second-normal-form relation is fairly simple: construct a separate relation to embody the partial dependency, and remove the dependent attribute from the original relation
- The previous example might have the following solution, which is in 2NF:

Table: People			
ID *	First_Name	Last_Name	Birthday
3	John	Smith	08/01/62
4	Sue	Jones	10/12/75
5	Mary	Smith	02/07/65

Table: Spouses	
Husband*->People:ID	Wife*->People:ID
3	5

\*Primary key



# Third Normal Form (3NF)

- A relation is in 3NF if and only if it is in 2NF and every non-key attribute is nontransitively dependent on the primary key
- 3NF actually takes the process started in 2NF to its logical conclusion
- Columns in a table should have information that is intrinsically a part of the nature of the primary key



# 3NF, example

- Example of 3NF violation:  
Emp (Employee, Department, Location)
- If each department is located in one place, then the Location field is a fact about the Department – in addition to being a fact about the Employee
- To satisfy 3NF, relation Emp should be decomposed into two relations:  
Emp (Employee, Department)  
Dep (Department, Location)



# Forth Normal Form (4NF)

- Under 4NF, a record type should not contain two or more independent multi-valued facts about an entity. In addition, the record must satisfy 3NF
- 4NF continues the process of reducing interdependencies between columns (not tables)



## 4NF, example

- Consider employees, skills, and languages, where an employee may have several skills and several languages

- Under 4NF, these data should not be represented in a single record such as

Emp (Employee, Skill, Language)

- Instead, they should be represented in the two records:

Emp\_Skill (Employee, Skill)

Emp\_Lang (Employee, Language)



## 4NF, cont'd

- In the definition of 4NF *independent multi-valued facts* were mentioned. Consider the extended explanations on the example:

Emp (Employee, Skill, Language)

- The two many-to-many relationships, **Employee:Skill** and **Employee:Language**, are "independent" in that there is no direct connection between skills and languages
- There is only an indirect connection because they belong to some common employee



# Fifth Normal Form (5NF)

- A table is in 5NF if it cannot have a lossless decomposition into any number of smaller tables
- Stated another way, 5NF indicates when an entity cannot be further decomposed
- 5NF is based on the concept of join dependence
- Join dependency means that a table, after it has been decomposed into three or more smaller tables, must be capable of being joined again on common keys to form the original table



# 5NF, example

- Consider example, involving agents, companies, and products
- Agents represent companies, companies make products, and agents sell products, then we might want to keep a record of which agent sells which product for which company
- Thus we need the combination of three fields to know which combinations are valid and which are not
- This information could be kept in one record type with three fields, which is in 5NF:  
Agents (Agent, Company, Product)

# Query optimisation

- Query optimization is the process of analyzing individual queries to determine what resources they use and whether the use of resources can be reduced
- For any query, you need to understand how it accesses database objects, the size of the objects, and indexing on the tables in order to determine whether it is possible to improve the query's performance
- In order to reach better query execution performance, DBMSs use *query optimisers*

# Query plans

- The input to the optimizer is a parsed SQL query; the output from the optimizer is a query plan.
- The *query plan* is the ordered set of steps required to carry out the query, including the methods (table scan, index choice, and so on) to access each table
- A query plan is compiled code that is ready to run
- The difference in cost between a good plan and a bad plan can be several orders of magnitude:
  - A good query plan can evaluate the query in seconds
  - Bad one might take days!

# Query plans, cont'd

In developing query plans, the optimizer examines:

- The size of each table in the query, both in rows and data pages
- The indexes, and the types of indexes, that exist on the tables and columns used in the query
- Whether the index covers the query, that is, whether the query can be satisfied by retrieving data from index keys without having to access the data pages
- Optimizable join clauses and the best join order, considering the costs and number of table scans required for each join and the usefulness of indexes in limiting the scans
- The size of the available data cache, the size of I/O supported by the cache, and the cache strategy to be used
- The estimated cost of physical and logical reads and cost of caching
- Some other factors...

## Query plans, cont'd

- Optimiser also distinguishes and analyse the following types of queries:
  - Search arguments in the **where** clause
  - Joins
  - Queries using **or** clauses and the **in (*values\_list*)** predicate
  - Aggregates
  - Subqueries

# Query optimisation, examples

- Consider some examples given in the class notes:

```
SELECT *  
FROM Students  
WHERE grade='A'  
      AND sex='female'
```

is better than

```
SELECT *  
FROM Students  
WHERE sex='female'  
      AND grade='A'
```

- Because usually there are fewer A students than female students.

# Query optimisation, examples cont'd

- Joins are better than nested queries

```
SELECT S.Theater
FROM Schedule S
WHERE S.Title IN (SELECT M.Title
                  FROM Movies M
                  WHERE M.director='Spielberg')
```

is likely to run slower than

```
SELECT S.Theater
FROM Schedule S, Movies M
WHERE S.Title = M.Title
      AND M.director='Spielberg'
```

# Course Project: Database system for a library

- Projects requirements are available from: [www.dit.unitn.it/~ilya/project.pdf](http://www.dit.unitn.it/~ilya/project.pdf)
- Each person does his/her own project
- In English (preferably) or in Italian
- Each student should send an e-mail before the next lab to [ilya@dit.unitn.it](mailto:ilya@dit.unitn.it) and indicate:
  - his/her name
  - assigned number / request for a number if not given yet or has a shared one
  - (optionally) password to access the database

# References

- Course lecture material
- “Concurrency Control and Recovery In Database Systems”, Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman
- More information on MySQL and SQL is available at <http://www.mysql.com>
- Find more about PostgreSQL at <http://www.postgresql.org>
- Still have questions? Contact [ilya@dit.unitn.it](mailto:ilya@dit.unitn.it)



# Appendix A: Exercises answers

```

ilya=> SELECT * FROM Movie NATURAL JOIN Schedule;
  title | director | actor | theater
-----+-----+-----+-----
Chinatown | Polanski | Nicholson | Odeon
Chinatown | Polanski | Nicholson | Le Champo
Chinatown | Polanski | Polanski | Odeon
Chinatown | Polanski | Polanski | Le Champo
Player | Altman | Robbins | Le Champo
Repulsion | Polanski | Deneuve | Odeon
Shining | Kubrick | Nicholson | Le Champo
<7 rows>

```

1. "Find natural join of Movie and Schedule"

```

ilya=> SELECT Schedule.theater
ilya-> FROM Schedule
ilya-> WHERE Schedule.title IN
ilya->   <SELECT Movie.title
ilya<>   FROM Movie
ilya<>   WHERE Movie.director='Polanski'>;
  theater
-----
Le Champo
Odeon
Odeon
<3 rows>

```

2. "Find theatres showing Polanski's Movies"

# Appendix A: Exercises answers, cont'd

```
ilya=> SELECT actor AS Person
ilya-> FROM Movie
ilya-> EXCEPT
ilya-> SELECT director AS Person
ilya-> FROM Movie;
person
-----
Deneuve
Nicholson
Robbins
(3 rows)
```

3. "Find all actors who are not directors"

```
ilya=> SELECT actor AS Person
ilya-> FROM Movie
ilya-> INTERSECT
ilya-> SELECT director AS Person
ilya-> FROM Movie;
person
-----
Polanski
(1 row)
```

4. "Find all actors who are also directors"

# Appendix A: Exercises answers, cont'd

```

ilya=> SELECT DISTINCT M.director
ilya-> FROM Movie M
ilya-> WHERE NOT EXISTS
ilya->   (SELECT S.theater
ilya<>   FROM Schedule S
ilya<>   EXCEPT
ilya<>   SELECT S1.theater
ilya<>   FROM Schedule S1, Movie M1
ilya<>   WHERE S1.title=M1.title
ilya<>   AND M1.director = M.director);
director
-----
Polanski
(1 row)

```

5. “Find directors whose movies are playing in all theatres”  
[using NOT EXISTS, EXCEPT]

```

ilya=> SELECT M.director
ilya-> FROM Movie M
ilya-> WHERE EXISTS
ilya->   (SELECT *
ilya<>   FROM Schedule S
ilya<>   WHERE S.title = M.title
ilya<>   AND S.theater = 'Le Champo');
director
-----
Kubrick
Altman
Polanski
Polanski
(4 rows)

```

6. “Find directors whose movies are playing at Le Champo”  
[using EXISTS]

# Appendix A: Exercises answers, cont'd

```
ilya=> SELECT M.director
ilya-> FROM Movie M
ilya-> WHERE M.title IN
ilya->   (SELECT S.title
ilya(>   FROM Schedule S
ilya(>   WHERE S.theater = 'Le Champo');
director
-----
Kubrick
Altman
Polanski
Polanski
(4 rows)
```

7. “Find directors whose movies are playing at Le Champo” [using IN]

```
ilya=> SELECT M.actor
ilya-> FROM Movie M
ilya-> WHERE M.actor NOT IN
ilya->   (SELECT M1.actor
ilya(>   FROM Movie M1
ilya(>   WHERE M1.director = 'Kubrick');
actor
-----
Robbins
Polanski
Deneuve
(3 rows)
```

8. “Find actors who did not play in a movie by Kubrick”