Coding for Data Science and Data Management
Module of Data Management

SQL

Stefano Montanelli
Department of Computer Science
Università degli Studi di Milano
stefano.montanelli@unimi.it

SQL (Structured Query Language)

- **SEQUEL**
  - (Structured English QUEry Language)
- **‘70-’80**
  - Language developed for System R, the IBM Relational DBMS (San Jose, CA, USA)
- **‘86**
  - First SQL standard (ANSI)
  - Valid DML functionalities
  - Limited DDL functionalities

SQL (Structured Query Language)

- **‘89**
  - Extension of the standard (support to referential integrity SQL-89
- **‘92**
  - Second version of the standard (introduction of a number of DDL functionalities) SQL-92 or SQL-2
- **Today**
  - Third version of the standard with many extensions (e.g., trigger, composite types, recursive views, support to very large objects – BLOB/CLOB) SQL-99 or SQL-3

Example: the moviedb schema

- Notation:
  - **Primary keys** are underlined in bold
  - *Foreign keys* are in italic
  - Fields with possible null values are labeled with a star *

Example: the moviedb schema

COUNTRY(iso3, name)

MOVIE(id, official_title, budget, year, length, plot)

PERSON(id, bio, first_name, last_name, birth_date, death_date*)

GENRE(movie, genre)

CREW(person, movie, p_role, character*)

LOCATION(person, country, d_role, city, region)

RATING(source, movie, check_date, scale, score, votes)

PRODUCED(movie, country)

RELEASED(movie, country, released, title)

SIM(movie1, movie2, cause, score)

Coding for Data Science and Data Management
Module of Data Management

SQL
Data Definition Language

Stefano Montanelli
Department of Computer Science
Università degli Studi di Milano
stefano.montanelli@unimi.it

Schema creation

- A database is created through the following statement

  CREATE SCHEMA [schema name]
  [AUTHORIZATION Username]
  [{schema elements}]

- **Schema name** is the name of the created object
- **Username** is the name of the database owner
- **Schema elements** are the database structures to insert in the database schema

Content of a database schema

- The following schema elements can be created within a database schema through the corresponding SQL statement:
  - Domain (CREATE DOMAIN)
  - Table (CREATE TABLE)
  - Assertion (CREATE ASSERTION)
  - View (CREATE VIEW)
  - User (CREATE USER)
  - Provileges (GRANT / REVOKE)

The CREATE DATABASE statement

- Most of the DBMSs also provide the CREATE DATABASE statement that is NOT a standard SQL statement

  CREATE DATABASE DBname
  [[WITH] [OWNER [=] Username]
  [ENCODING [=] encoding]]

- **DBname** is the name of the database to create
- **Username** is the name of the database owner
- **Encoding** is the character encoding to use in the database (e.g., SQL_ASCII, UTF8)

Schema vs. database

- The relation between schema and database depends on the DBMS
- Example
  - Oracle Express Edition
  - Only one database (CREATE DATABASE is not supported) containing all the independent database schemas created through CREATE SCHEMA
  - PostgreSQL
  - Many databases can be created through CREATE DATABASE and each database can contain many schemas created through CREATE SCHEMA

Table creation

```
CREATE TABLE TableName (
   AttributeName Domain [DefaultValue]
      [constraints (attribute level)]
   {, AttributeName Domain [DefaultValue]
      [constraints (attribute level)]}
   [further constraints (table level)]
)
```

Elementary data types (domains)

Numeric exact (fix point)	Integer	Integer
		Smallint
	Integer and decimal	Numeric
		Decimal
Numeric approximate (floating point)	Real	*Comparisons between pairs of values are not possible*
	Double precision	
	Float	

Elementary data types (domains)

Textual	Character (char)
	Character varying (varchar)
Boolean	Bit, Boolean
	Bit varying
Temporal	Date
	Time
	Timestamp

Default values

- A default clause is set to specify the value to assign to an attribute instead of null
- In a CREATE TABLE:

  …

  AttributeName Domain DEFAULT *value*

  …
- The *value* is user-defined and it is compatible with the attribute domain
- The *value* can be a fixed constant or the result of dynamic expression

Intra-table constraints

- An intra-table constraint represents a condition that needs to be satisfied by all the tuples of the table on which the constraint is specified
- An intra-table constraint can be specified for a single attribute or a set of attributes
- In the latter case, the constraint has to be satisfied by the set of attribute as a whole

Intra-table constraints

AttributeName Domain ***NOT NULL***

- It specifies that the null value is not possible for the associated attribute
-

AttributeName Domain ***UNIQUE***

- It specifies that different tuples cannot have the same value on the associated attribute
- The null value is not considered by the unique constraint

Intra-table constraints

AttributeName Domain ***PRIMARY KEY***

- It specifies the primary key of the table
- It can be used one-and-only-one time within a table
- Two or more primary keys in a single relation/table are NOT possible nor meaningful

Inter-table constraints

- Inter-table constraints are relational integrity constraints
- A relational integrity constraint is defined between a single (or a set of) attribute a_R of a *referring table R* with a single (or a set of) attributes a_T of a *referred table T*
- In the CREATE TABLE of *R*:

  a_R *Domain REFERENCES* $T(a_T)$

Inter-table constraints

- A relational integrity constraint ensures that:
  - for each tuple of R, the value of the attribute a_R exists as value of the attribute a_T (if a_R is not null)
- The attribute a_T MUST be unique in T (in other words, the attribute a_T must be a key of T)
- Typically, the attribute a_T is the primary key of T

Referential integrity

- Referential integrity allows to specify an action to execute on the referring table R when a violation of the integrity constraint occurs on the referred table T
- Actions are triggered on update/delete operations on values on the referred attribute a_T of the foreign key

Violation of referential integrity

- Consider an update/delete operation of a value v_T in the referred attribute a_T of a foreign key
- What happens to the value v_R of the foreign key a_R in the referring table R?

p_sequence (T table)

id	official_title	year	length
1375666	Interstellar	2014	169

?

sequence_species (R table)

movie	person	role	character
0816692	0634240	director	

Violation of referential integrity

- Possible actions:
  - **CASCADE**: the value(s) v_R of the foreign key a_R are updated/deleted (the action executed on v_T is applied also to v_R in a cascade manner)
  - **SET NULL**: the value(s) v_R of the foreign key a_R are set to the NULL value

Violation of referential integrity

- Possible actions:
  - **SET DEFAULT**: the value(s) v_R of the foreign key a_R are set to the default value specified for a_R (if any, otherwise the SET NULL action is executed)
  - **NO ACTION**: the update/delete operation on the attribute value v_T in the referred attribute a_T is rejectedto preserve database integrity (this is the predefined option)

User-defined integrity constraints

- It is possible to specify user-defined constraints on the attribute values of a specific table
- The constraint is represented as a (combination of) boolean predicate
- In the CREATE TABLE:

  attribute Domain CHECK (condition)
- User-defined constraints about attributes of different tables require the specification of an ASSERTION

User-defined domains

- In addition to predefined domains, it is possible to specify custom attribute domains:

  CREATE DOMAIN DomainName AS
  BaseDomain [DefaultValue] [{Constraints}];

- **DomainName** is the user-defined domain name
- **BaseDomain** is the reference DBMS domain upon which the new domain is generated
- **DefaultValue** and **Constraints** represent custom conditions to require according to the conventional SQL syntax

Edit of database schema

- The ALTER statement is defined in SQL to change the structure of schema elements previously defined
- Explore the DBMS guide for a complete syntax of the ALTER statement
- Example:

  ALTER TABLE member ADD COLUMN annual_ticket decimal(8, 2) DEFAULT 0;

Deletion of schema elements

- The DROP statement is defined to delete/remove schema elements from a database

  DROP

  <SCHEMA | DOMAIN | TABLE | VIEW | ASSERTION>

  ElementName

Coding for Data Science and Data Management
Module of Data Management

SQL
Data Manipulation Language

Stefano Montanelli
Department of Computer Science
Università degli Studi di Milano
stefano.montanelli@unimi.it

SQL as a query language

- SQL expresses queries in a declarative way
  - queries specify the properties of the result, not the way to obtain it
- The DBMS (query processing and query optimizer modules) translates SQL queries into internal procedural language for query execution

SQL queries

```
SELECT      Target List
FROM        Table list
[WHERE      Condition]
```

- **SELECT**: attributes whose values have to be retrieved and shown in the query result
- **FROM**: relations on which the query is evaluted
- **WHERE**: boolean expression providing the condition to satisfy by the relations tuples to be included in the query result

Simple SQL query (example)

- Retrieve the title of movies with length higher than 120 minutes

```
SELECT official_title AS ‘movie title’
FROM movie
WHERE length > 120;
```

- Attributes can be renamed in the query result through the AS operator

The * operator in the SELECT clause

- The star (*) operator specifies to retrieve in the result all the attributes of the relations in the FROM clause
- Example: retrieve all the information about movies with length higher than 120 minutes

```
SELECT *
FROM movie
WHERE length > 120;
```

Attribute expressions

- The SELECT clause can contain expressions to manipulate the attribute values

```
SELECT annual_ticket/12 AS ‘monthly ticket’
FROM member;
```

WHERE clause

- The WHERE clause is a conjunction/disjunction of boolean predicates expressing conditions on tuples
  - **AND**: all the tuples that satisfy all the predicates in the clause are retrieved in the result
  - **OR**: all the tuples that satisfy at least one predicate in the clause are retrieved in the result
  - The **NOT** operator is also available:
    - all the tuples that DO NOT satisfy the predicate in the clause are retrieved in the result

Predicate conjunction

- Retrieve the movies with length higher than 120 minutes released in 2010

```
SELECT id, official_title
FROM movie
WHERE length > 120 AND year = '2010';
```

Predicate disjunction

- Retrieve the movies with length of 120 or 240 minutes

```
SELECT id, official_title
FROM movie
WHERE length = 120 OR length = 240;
```

- We can use parenthesis to build complex boolean predicates combining AND, OR, NOT

Pattern matching

- In the WHERE clause, predicates based on pattern matching are allowed through the user of the LIKE operator

  [NOT] LIKE pattern

- To set string patterns:
  - underscore '_' to denote an arbitrary character
  - percent '%' to denote a string of arbitratry length

Pattern matching

- Retrieve the movies about ‘star wars’

```
SELECT *
FROM movie
WHERE official_title like ‘%star wars%’;
```

Duplicates

- In SQL, it is possible that duplicate tuples are retrieved
- The DISTINCT keyword can be used to remove duplicate tuples from the result

```
SELECT DISTINCT official_title
FROM movie
WHERE year = '2010' OR length > 120;
```

The JOIN operator

- The JOIN operator is provided for retrieving corresponding tuples belonging to different tables

  R JOIN S ON $a_R = a_S$

- The JOIN operator has the goal to «combine» the tuples of *R* with the corresponding tuples of S
- Corresponding tuples are those with the same value on the attributes a_R of *R* and a_S of *S*
  - a_R is a foreign key of *R* referring the key a_S of *S* or viceversa

The JOIN operator

- SQL-2 introduced a syntax for explicitly expressing joins in the FROM clause
- Different kinds of JOIN are supported
  - INNER JOIN
  - NATURAL JOIN
  - RIGHT, LEFT, FULL OUTER JOIN

INNER and NATURAL JOIN

- The INNER JOIN between *R* and *S* returns the joined tuples of *R* and *S* where the condition $a_R = a_S$ is satisfied

  R INNER JOIN S ON $a_R = a_S$

- The NATURAL JOIN works as the INNER JOIN without requiring to specify the equality condition
  - Tuples are joined by considering the value equality between attribute pairs of *R* and *S* with the same name

    R NATURAL JOIN S

INNER JOIN example

- Retrieve the first and last name of actors that played in the movie ‘Interstellar’ (id = 0816692)

```
SELECT first_name, last_name
FROM person INNER JOIN
        crew ON
        person.id = crew.person
WHERE p_role = ‘actor’ AND
            movie = 0816692;
```

- Question: how to filter according to the movie title instead of the movie.id? (hint: need of one more join operation)

OUTER JOINS

- LEFT OUTER JOIN extends the INNER JOIN with the tuples of *R* (the relation on the left of the JOIN) that do NOT have matching tuples in *S*

  R LEFT OUTER JOIN S ON $a_R = a_S$

- RIGHT OUTER JOIN extends the INNER JOIN with the tuples of *S* (the relation on the right of the JOIN) that do NOT have matching tuples in *R*

  R RIGHT OUTER JOIN S ON $a_R = a_S$

OUTER JOIN example

- Retrieve all the movies with related ratings

```
SELECT movie.id, official_title, score
FROM movie LEFT OUTER JOIN
        rating ON movie.id = rating.movie;
```

- Also movies that are not associated with any rating are included in the result

Queries with NULL values

- The WHERE clause can contain conditions to test the presence (or not) of NULL values for attributes

  WHERE attribute IS NULL

- The predicate is evaluated TRUE for a tuple if the attribute contains a NULL value
- The IS NOT NULL condition can be used to retrieve the tuples with a NON-NULL value

Example

- Retrieve the persons without a bio

```
SELECT *
FROM person
WHERE bio IS NULL;
```

Management of NULL values

- SQL-89 uses a two-valued logic (TRUE, FALSE)
  - a comparison with a NULL value returns FALSE
- SQL-2 uses a three-valued logic (TRUE, FALSE, UNKNOWN)
  - a comparison with a NULL value returns UNKNOWN
- In query result:
  - Tuples for which the WHERE condition is evaluated TRUE are retrieved
  - Tuples for which the WHERE condition is evaluated FALSE/UNKNOWN are not retrieved

Ordering of results

- The ORDER BY clause si provided to specify the ordering of tuples in the results
- The ORDER BY clause is specified at the end of the query

  ORDER BY attribute [ASC | DESC]

  {, Attribute [ASC | DESC]}
- Multiple attributes can be specified and priority is from left to right
- Default ordering is ASC – ascending

Table variables (ALIAS)

- Table aliases can be considered as table variables
- The alias is used to refer to the table from within the query
- Aliases are useful not only to concisely refer to a table in query writing, but also to compare each other tuples of the same relation

Example

- Retrieve the movies with length higher than ‘Interstellar’ (sort result by title)

```
SELECT m2.*
FROM movie AS m1,
        movie AS m2
WHERE m1.official_title = ‘Interstellar’ AND
      m1.length < m2.length
ORDER BY m2.official_title;
```

Aggregate queries

- SQL offers aggregate operators to calculate aggregate values out of sets of tuples in the database relations
  - **COUNT**: count the number of tuples
  - **SUM**: sum the values on an attribute expression
  - **MAX**: find the max value on an attribute expression
  - **MIN**: find the min value on an attribute expression
  - **AVG**: find the average value on an attribute expression

The COUNT operator

- The count operator returns the number of distinct rows or distinct values
  - **distinct** considers each value just once
  - **all** considers all not-null values

COUNT (< * | [distinct | all] > attributeList)

Examples

- Retrieve the number of movies in the db

```
SELECT count(*) AS "movie count"
FROM movie;
```

- Retrieve the number of movies released in 2010

```
SELECT count(*) AS "movies of 2010"
FROM movie
WHERE year = ‘2010’;
```

Examples

- Retrieve the number of different roles that appear in the crew

```
SELECT count(distinct p_role)
FROM crew;
```

- Retrieve the number of persons with known birthdate (non-null birth_date)

```
SELECT count(all birth_date)
FROM person;
```

SUM-MAX-MIN-AVG operators

- SUM-MAX-MIN-AVG can be applied on the values of a considered attribute or attribute expression
  - **distinct** considers each value just once
  - **all** considers all not-null values

Example

- Retrieve the sum-max-min-avg of annual tickets paid by member users

```
SELECT sum(annual_ticket) AS "sum tickets",
       max(annual_ticket) AS "max ticket",
       min(annual_ticket) AS "min ticket",
       avg(annual_ticket) AS "avg ticket"
  FROM member;
```

GROUP BY queries

- Queries may apply aggregate operators to subsets of rows

  GROUP BY attributeList

- First the groups of rows are formed, then the aggregated operator is applied to EACH group

IMPORTANT NOTE on GROUP BY

- When the GROUP BY clause is specified, the SELECT clause can contain only
  - the attributes in the attributeList of the GROUP BY
  - aggregate operators on an attribute expression

Example

- Retrieve the number of actors for each movie

```
SELECT movie, count(person)
FROM crew
WHERE p_role = ‘actor’
GROUP BY movie;
```

Group predicates

- The HAVING clause can be used to specify conditions on groups

  GROUP BY attributeList
  HAVING predicate

- Only groups satisfying the HAVING condition are shown in the result

Example

- Retrieve the movies with a cast composed of more than 10 actors

```
SELECT movie, count(person)
FROM crew
WHERE p_role = ‘actor’
GROUP BY movie
HAVING count(*) > 10;
```

WHERE or HAVING clause?

- Retrieve the movies with length higher than 120 min and cast composed of more than 10 actors

```
SELECT movie, count(person)
FROM movie INNER JOIN
        crew ON movie.id=crew.movie
WHERE length > 120 AND p_role = ‘actor’
GROUP BY movie
HAVING count(*) > 10;
```

SET queries

- Set operations are provided to support UNION, INTERSECT, EXCEPT
  - Default behavior: duplicate removal
  - **ALL**: keep duplicates in the result

Example

- Retrieve the persons that are born OR dead in Italy (iso3 code = ITA)

```
SELECT person
FROM location
WHERE d_role = ‘birth’ AND country = ‘ITA’
UNION
SELECT person
FROM location
WHERE d_role = ‘dead’ AND country = ‘ITA’;
```

Example

- Retrieve the persons that are born AND dead in Italy (iso3 code = ITA)

```
SELECT person
FROM location
WHERE d_role = ‘birth’ AND country = ‘ITA’
INTERSECT
SELECT person
FROM location
WHERE d_role = ‘dead’ AND country = ‘ITA’;
```

Example

- Retrieve the persons that are born in Italy (iso3 code = ITA), but dead elsewhere

```
SELECT person
FROM location
WHERE d_role = ‘birth’ AND country = ‘ITA’
EXCEPT
SELECT person
FROM location
WHERE d_role = ‘dead’ AND country = ‘ITA’;
```

Nested queries

- In the WHERE clause we have a predicate whose right part is an SQL query
- The goal is to compare an attribute value (or the result of an attribute expression) with the result of the SQL query on the right

Example (ANY operator)

- Retrieve the movies that have a genre in common with the ‘Interstellar’ movie

```
SELECT id, official_title
FROM movie INNER JOIN genre ON
movie.id = genre.movie
WHERE genre = ANY
        (SELECT genre FROM movie
         INNER JOIN genre ON
        movie.id = genre.movie
        WHERE official_title = ‘Interstellar’);
```

Example (IN operator)

- Retrieve the movies that have a genre in common with the ‘Interstellar’ movie

```
SELECT id, official_title
FROM movie INNER JOIN genre ON
movie.id = genre.movie
WHERE genre IN
        (SELECT genre FROM movie
         INNER JOIN genre ON
        movie.id = genre.movie
        WHERE official_title = ‘Interstellar’);
```

Example

- Retrieve the movies that have not been released in Italy (iso3 code = ITA)

```
SELECT id, official_title
FROM movie
WHERE id NOT IN
        (SELECT movie FROM released
         WHERE country = 'ITA');
```

- Alternative solutions are possible. Any idea?

Example

- Retrieve the movies that have a rating higher than the average of ratings of the ‘Interstellar’ movie (id = 0816692)

```
SELECT DISTINCT movie
FROM rating
WHERE score >
        (SELECT avg(score)
         FROM rating
         WHERE movie = 0816692);
```

Correlated nested queries

- The nested subquery (internal query) is executted only once; resulting set is used to evaluated the WHERE clause of the external query
- Correlated nested query are complex nested queries where the nested query needs to be executed for each tuple of the external query

Example

- Retrieve the movies that have a rating from a source S higher than the average of all the ratings provided by S

```
SELECT x1.movie, x1.score
FROM rating AS x1
WHERE x1.score >
        (average of all the ratings provided by
the source of x1, namely x1.source);
```

Example

- Retrieve the movies that have a rating from a source S higher than the average of all the ratings provided by S

```
SELECT x1.movie, x1.score
FROM rating AS x1
WHERE x1.score >
        (SELECT AVG(score)
         FROM rating AS x2
         WHERE x1.source = x2.source);
```

Correlated nested queries - EXISTS

- Predicate EXISTS(sq) is TRUE if the subquery sq returns a non-empty result; it is FALSE otherwise

- Predicate NOT EXISTS(sq) is the negation of EXISTS

Example

- Retrieve the movies that are not released in the countries where they are produced

```
SELECT x.*
FROM movie AS x
WHERE NOT EXISTS
        (SELECT y.country FROM produced AS y
         WHERE (x.id = y.movie)
         INTERSECT
         SELECT z.country FROM released AS z
         WHERE (x.id = z.movie));
```