Structured Query Language (SQL) – Part 1

Marek Rychly
mrychly@strathmore.edu

Strathmore University,
@iLabAfrica

&

Brno University of Technology,
Faculty of Information Technology

Advanced Databases and Enterprise Systems
14 September 2015
Outline

1. Structured Query Language (SQL)
   - History of SQL and Relational DBMS
   - SELECT Statement
   - Sub-queries in the SELECT Statement
query, data manipulation and definition relational db. language
- to create database and relation structures
- to perform data management task (insert, update, delete)
- to perform both simple and complex queries

standard both by specification and by usage in practice
- specified by International Organization for Standardization (ISO)
- utilized by DBMS vendors (Oracle, IBM, open-source, etc.)

a transform-oriented language based on relational algebra
(an expression describes how to transform input relations to an output relation)

a non-procedural (declarative) language
(it describes what operations of relational algebra should be applied on input data, not how it should be done in terms of input data retrieval, their processing, etc.)
Early History of SQL

1970  E. F. Codd, from IBM labs., introduced the relational database model (including relational algebra and relational calculus)

1974  D. Chamberlin, also from IBM labs., introduced SEQUEL (a relational database language called the Structured English Query Language)

1976  Chamberlin et al. introduces SEQUEL/2 (later, it has been renamed to SQL for legal reasons)

IBM produced a prototype DBMS based on SEQUEL/2, called System R (also based on SQUARE (Specifying Queries As Relational Expressions), 1975)

1978–79  SDL/RSI company introduced a first versions of Oracle, V1 and V2 (Oracle V2 was the first commercial RDBMS; RSI became Oracle Corp. in 1982)

1987-89  The first/initial ISO standard of SQL was published and extended later (many relational features were missing, implemented by vendors in various ways)
Current History of SQL

1992 SQL-92, the first major/usable revision to the ISO standard occurred (still, it did not cover many advanced features implemented by these days DBMSs)

1999 SQL:1999 published by ISO, SQL formalized including advanced features (such as support object-oriented data management)

Oracle8i released as an RDBMS inter-operating better within the Internet

2003 SQL:2003 published by ISO, consisted of Core SQL and SQL Packages (the core mandatory, packages optional, e.g., for object and XML data, etc.)

2003-13 SQL/MM (SQL multimedia and application packages) by ISO (full-text, spatial, and still image data, data mining, history, meta-data registries)

2006 SQL:2006 as Part 14 of SQL:2003 published by ISO (a package that defines ways in which SQL can be used in conjunction with XML)

2008 SQL:2008 published by ISO, another restructuring of the specification (Framework, Foundation, Object Language Bindings, Information and Definition Schemas, SQL Routines and Types Using Java, XML-Related Specifications, etc.)

2011 SQL:2008 published by ISO as an update of SQL:2008 (new features include improved support for temporal databases)
Early History of RDBMS Systems

(adopted from “History of RDBMS, Data-e-Education, 2014”)
Current History of RDBMS Systems

(adopted from “RDBMS Genealogy, Hasso-Plattner-Institut, 2015”)

Marek Rychly
Web SQL (SQLite) Relational DBMS

- relational DBMS provided by web-browser by HTML5 API
  (based on the underlying SQLite RDBMS, e.g., SQLite v3.6.1 in Google Chrome)

- currently supported only by WebKit-based web-browsers: Chrome, Safari, Opera
  (API is deprecated by W3C since end of 2010 but it is still supported)

- utilized by Try-SQL Editor in W3Schools SQL Tutorial
  http://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all

- supports all SQLite features (including triggers)
  (all but storage management and unsecure features, e.g., ATTACH/VACUUM)

- suitable for practising the SQL

- examples shown in this lecture will be demonstrated on Try-SQL

- the examples will use tables provided by W3Schools SQL Tutorial
SQL can be divided into two parts
- Data Definition Language (DDL)
  (for defining the database structure and controlling access to the data)
- Data Manipulation Language (DML)
  (for retrieving and updating data)

SQL statements consist of two types of words
(both are used as case-insensitive, contrary to data-typed string values)
- reserved words
  (select, from, where, order by, group by, create, table, row, primary key, etc.)
- user-defined words
  (names of tables, columns, indices, views, etc.)

individual SQL statements/commands separated by semicolon “;”
non-numeric literals (constants) are enclosed in single quotes
(for example: 123, −1.23, ’John Smith’, date(’now’), date(’2015-09-114’), . . .)
SELECT [DISTINCT | ALL] {* | columnExpression [AS newName]] [, ...]} FROM tableName [alias] [, ...] [WHERE condition] [GROUP BY columnName [, ...]] [HAVING condition] [ORDER BY columnNameOrNewName [DESC | ASC] [, ...]]

- **columnExpression** is an expression with column names (in the resulting relation, the result of the expression can be known as **newName**)
- **tableName** is the name of an input table, **columnName** of a table column (in the select statement, the table can be referred by **alias**)
- **condition** is a logical expr. with table names/aliases and column names
- **columnNameOrNewName** is the name of a column or a column expr. above

The statement selects such data from input tables that meet “where” condition. For such, all or distinct, data rows, column expressions are computed. If the case of the expression with an aggregate function, the data are grouped by “group by” condition, the aggregate function is applied, and results is checked to meet “having” condition. Finally, the resulting data are ordered by values of given columns in ascending or descending order.
Example of SELECT Statement

```
SELECT C.Country, count(C.CustomerID) AS CustomersInCountry 
FROM Customers C 
WHERE C.CustomerName LIKE '%%%'
GROUP BY C.Country HAVING CustomersInCountry >= 3 
ORDER BY CustomersInCountry DESC;
```

- Take all customers that have a space in their names,
- group them into groups according to their country, 
  (all members of each group will have the same value of the country column)
- take just such groups that have at least 3 members (rows),
- for each group, print country value and number of group members,
- ordered by the number of group members descending.
Clauses in the SELECT Statement

SQL SELECT statements are processed in the following order: (note that the processing order is different from the order of the clauses in the statement)

- **FROM** – specifies the table or tables to be used,
- **WHERE** – filters the rows subject to some condition,
- **GROUP BY** – forms groups of rows with the same column value,
- **HAVING** – filters the groups subject to some condition,
- **SELECT** – specifies which columns are to appear in the output,
- **ORDER BY** – specifies the order of the output.

SELECT and FROM clauses are mandatory, others are optional.
The SELECT Clause

```sql
SELECT [DISTINCT | ALL] {* | columnExpression [AS newName]} [, ...]
```

- Specifies which columns are to appear in the output.
- Represents the **Projection operation** of relational algebra.
- The column expression consists of:
  - column names from tables in FROM clause,
  - constants,
  - expressions of the column names, constants, and operations.
    (e.g., arithmetic operations; string operations such as concatenation; etc.)
- Star symbol “*” means all possible columns.
  (e.g., `SELECT * FROM Customers;` lists all values form the “Customers” table)
- By default, SELECT does not eliminate duplicities.
  To eliminate duplicities, `DISTINCT` has to be used (`ALL` is default).
  (duplicate data may come from input tables or may be produced by the projection)
Examples on SELECT Clause

- List all cities of customers including duplicities.
  
  ```sql
  SELECT City FROM Customers;
  ```

- List all cities of customers without duplicities.
  
  ```sql
  SELECT DISTINCT City FROM Customers;
  ```

- List all columns of all rows from table “Employees”.
  
  ```sql
  SELECT * FROM Employees;
  ```

- List full names and age in days of all employees.
  
  ```sql
  SELECT FirstName || ' ' || LastName AS FullName,
          julianday('now') - julianday(BirthDate) AS AgeInDays
  FROM Employees;
  ```
The WHERE Clause

[WHERE condition]

- Filters the rows subject to some condition.
- Represents the Selection operation of relational algebra.
- The condition consists of
  - constants, column names from tables in FROM clause,
  - predicates applied on of the constants and column names,
  - logical operators AND, OR, and NOT, with parentheses
- There are several predefined predicates for testing
  - comparison: =, <>, <, >
    (compare the value of one expression to the value of another expression)
  - range: \(x \text{ BETWEEN } a \text{ AND } b, x \text{ NOT BETWEEN } a \text{ AND } b\)
    (test whether the value falls within a specified range of values)
  - set membership: \(x \text{ IN } (a, b, c), x \text{ NOT IN } (a, b, c)\)
    (test whether the value equals one of a set of value)
  - pattern match: \(x \text{ LIKE 'pattern'}, x \text{ NOT LIKE 'pattern'}\)
    (test whether a string matches a specified pattern, usually case-sensitive)
  - null: \(x \text{ IS NULL}, x \text{ IS NOT NULL}\)
    (test whether a column has a Null (unknown) value)
Pattern Matching an Null Search in WHERE Clause

columnExpression [NOT] LIKE 'pattern' [ESCAPE 'escapeChar']

Pattern is a text literal with two special pattern-matching symbols:
- % represents any sequence of zero or more characters (a wildcard),
- _ represents any single character.

The special symbols can be used as common characters if escaped by escapeChar. For example:

```sql
SELECT * FROM Sale WHERE Discount LIKE '%15#%' ESCAPE '#';
```

columnName IS [NOT] NULL

- The expression is true if and only if the column value is (not) Null.
- Null value used as operand in any operation will result into Null.
The ORDER BY Clause

\[
\text{[ORDER BY.columnNameOrNewName [DESC | ASC] [, ...]]}
\]

- Rows of an SQL query result table are ordered. (contrary to results of relational algebra operations)

- The ORDER BY clause allows the retrieved rows to be ordered.

- If unspecified by ORDER BY clause, the order is undefined. (usually, in the such cases, resulting rows are ordered by primary keys)

- Result can be ordered by multiple columns and for each
  \- **ASC** means ascending order on a given column,
  \- **DESC** means descending order on a given column.

- The ORDER BY clause has to be the last clause of the statement.
Examples on WHERE and ORDER BY Clauses

- List employees whose first names start with A and were born in February.
  
  ```sql
  SELECT * FROM Employees
  WHERE FirstName LIKE 'A%' AND BirthDate LIKE '____-02-__';
  ```

- List products with non-empty supplier and price between $10 and $20 arranged in descending order of price.
  
  ```sql
  SELECT * FROM Products WHERE Price BETWEEN 10 AND 20
  AND SupplierID IS NOT NULL ORDER BY Price DESC;
  ```

- List full names and birth dates of employees in ascending order of names.
  
  ```sql
  SELECT LastName || ', ' || FirstName as FullName, BirthDate
  FROM Employees ORDER BY FullName ASC;
  ```

  or, alternatively, ...
  ```sql
  ORDER BY LastName ASC, FirstName ASC;
  ```

- List suppliers that are neither from UK nor from USA.
  
  ```sql
  SELECT * FROM Suppliers WHERE Country NOT IN ('UK', 'USA');
  ```
Aggregation and Grouping in the SELECT Statement

```
SELECT [columnExpression [, ...]]
 {aggregateFunction(columnName) | COUNT(*)}, [, ...]

[GROUP BY columnName [, ...]] [HAVING condition]
```

- With an aggr. function in SELECT clause the data are aggregated.
- The aggregate function applies on all values in a given column.
  - all non-Null values in the case of a particular column name,
  - all values including Null values in the case of `COUNT(*)` function,
  - all distinct values in the case of `DISTINCT` word preceding the column name in the operand of the aggregate function.

- By GROUP BY clause, the aggr. function aggregates data for each group of rows with identical values of grouped columns.

- In the case of non-aggregated columns in the SELECT clause, those columns must appear also in the GROUP BY clause.

- By HAVING clause, aggregation results are filtered by condition.
Aggregate Functions

- The SQL standard by ISO defines five aggregate functions:
  - **COUNT** – returns the number of values in a specified column,
  - **SUM** – returns the sum of the values in a specified column,
  - **AVG** – returns the average of the values in a specified column,
  - **MIN** – returns the smallest value in a specified column,
  - **MAX** – returns the largest value in a specified column.

- Relational DBMSs usually provide additional aggregate functions.
  (e.g., group_concat(column,separator), stddev(column), variance(column), etc.)

- To filter by aggregate function results, a condition has to be put into **HAVING** clause, not into **WHERE** clause.
  (the **HAVING** condition may contain the aggregate functions applications or aliases of columns from **SELECT** clause where the aggregation functions are)
Examples on GROUP BY Clause

- For each product ID get total number of its orders, total sum and maximal ordered quantity, in the cases when the total sum is 100 units or above.

  ```sql
  SELECT ProductID, COUNT(*), SUM(Quantity), MAX(Quantity) 
  FROM OrderDetails 
  GROUP BY ProductID 
  HAVING SUM(Quantity) >= 100;
  ```

- Get ID and name of products that have their prices the same or nearly same with maximal difference of $5 as the average price of all products.

  ```sql
  SELECT ProductID, ProductName FROM Products 
  WHERE Price BETWEEN ( 
    SELECT AVG(Price)-5 FROM Products 
  ) AND ( 
    SELECT AVG(Price)+5 FROM Products 
  );
  ```
The FROM Clause and Multi-Table Queries

... FROM tableName [alias] [, ...] ...

- The FROM clause of a single query can refer to multiple tables. (then, the query will process data from all the tables)

- To combine columns from several tables in the query into a result table we need to use a Join operation of relational algebra.

- In multi-table query, columns values can be accessed as tableName.columnName (this is necessary if there is a column of the same name in several tables)

- There are several different types of the Join operation. ((inner) cross join, (Theta) join, natural join, left/right/full outer join, etc.)
(Inner) JOIN and CROSS JOIN Operations

... FROM OrderDetails, Products ...
... FROM OrderDetails JOIN Products ...
... FROM OrderDetails INNER JOIN Products ...
... FROM OrderDetails CROSS JOIN Products ...

- The clauses above produce the same Cartesian product of tables. (i.e., they will pair all rows from the first table with all row from the second table)

- To match particular rows, join criteria must be defined. (then, it wont be a plain Cartesian product but a Theta join as in relational algebra)

- In multi-table query, columns values can be accessed as
  tableName.columnName
  (this is necessary if there is a column of the same name in two or more tables)

- Join criteria can be put into WHERE clause or into FROM clause. (in the FROM clause, the join criteria are prefixed by “ON” word)
Examples of Multi-table Queries

- List details of all orders where the ordered quantity is 10 units and more.

  ```sql
  SELECT * FROM OrderDetails, Products
  WHERE OrderDetails.ProductID = Products.ProductID
  AND Quantity >= 10;
  ```

  or, alternatively,

  ```sql
  SELECT * FROM OrderDetails JOIN Products
  ON OrderDetails.ProductID = Products.ProductID
  WHERE Quantity >= 10;
  ```

- List all orders with their details, details of ordered products and suppliers of the products.

  ```sql
  SELECT * FROM Orders
  JOIN OrderDetails ON OrderDetails.OrderID = Orders.OrderID
  JOIN Products ON Products.ProductID = OrderDetails.ProductID
  JOIN Suppliers ON Products.SupplierID = Suppliers.SupplierID;
  ```
The clauses above produce the same Natural join of tables.

Join criteria are equalities of columns of the same names. 
(see the Equi-join and Natural join operations in relational algebra)

For particular choice of columns of the same names, it is possible to use JOIN USING with a list of columns for the join criteria.
For example,

```
SELECT * FROM Orders JOIN OrderDetails USING (OrderID);
```
OUTER JOIN Operations

... FROM Customers LEFT OUTER JOIN Orders
    ON Customers.CustomerID = Orders.CustomerID ...
... FROM Customers LEFT OUTER JOIN Orders USING (CustomerID) ...
... FROM Customers NATURAL LEFT OUTER JOIN Orders ...
... FROM Orders NATURAL RIGHT OUTER JOIN Customers ...

- The clauses above produce the same results.
  (they are a Theta-, Equi-, and Natural left-outer join operations in relational algebra)

- Left (right) outer join takes all rows from the left (right) table and matches them with respective rows of the other table if possible, or with Null values otherwise.
  (see the Outer join operations in relational algebra)

- As the right-join and full-outer join ops. can be computed by left-outer join op(s)., they may not be supported by some RDBMS.
  (e.g., SQLite, that is also Web-SQL supports only the left outer join operation)
Examples on OUTER JOIN Operations

- List all orders with their customers and also customers without any orders.

  ```sql
  SELECT * FROM Customers NATURAL LEFT OUTER JOIN Orders;
  ```

- List all possible replacements of products with the same price.

  ```sql
  SELECT Original.ProductID AS OrigID,
         Original.ProductName AS OrigName,
         Alternative.ProductID AS AltID,
         Alternative.ProductName AS AltName,
         Price
  FROM Products Original LEFT OUTER JOIN Products Alternative
  USING (Price)
  WHERE OrigID <> AltID;
  ```
Sub-queries in SELECT Statements (Sub-selects)

- It is possible to perform an inner SELECT statement (a nested query) and to use its result(s) in an outer SELECT statement.

- From the result(s) produced, we can distinguish
  - scalar sub-queries – return a single column&row (a single value),
  - row sub-queries – return a single row with multiple columns,
  - table sub-queries – return multiple rows&columns (a table).

- A scalar sub-query can be used when a single value is needed. (for example, in WHERE clause as an operand of a predicate, e.g, with equality)

- Scalar sub-queries are useful especially to integrate results of aggregate functions into complicated (outer) queries. For example,

```
SELECT ProductID, ProductName FROM Products
WHERE Price BETWEEN (SELECT AVG(Price)-5 FROM Products)
       AND (SELECT AVG(Price)+5 FROM Products);
```
Row and table sub-queries can be used with the following predicates:
(these predicates exist also in their negative forms, prefixed by “NOT” word, e.g., NOT IN)

- **IN** to test if a value is in a result set a particular sub-query
- **ALL** to test if a predicate holds for all results of a particular sub-query
- **ANY** to test if a predicate holds for at least one result of a sub-query¹
- **EXISTS** to test if a result set a particular sub-query is not empty

ALL and ANY (SOME) may not be supported by some RDBMS.
(for example, they are not supported by SQLite, that is also by Web SQL)

¹ ANY predicate is also known as SOME predicate; ANY and SOME are synonyms
Examples of Sub-queries

- List all employees that did not take any orders.

  ```sql
  SELECT * FROM Employees
  WHERE EmployeeID NOT IN (SELECT EmployeeID FROM Orders);
  ```

  or, alternatively,

  ```sql
  SELECT * FROM Employees E WHERE NOT EXISTS
  (SELECT 1 FROM Orders O WHERE O.EmployeeID = E.EmployeeID);
  ```

- Get product with the highest price.

  ```sql
  SELECT * FROM Products
  WHERE Price >= ALL (SELECT Price FROM Products);
  ```

  or, alternatively,

  ```sql
  SELECT * FROM Products
  WHERE Price >= (SELECT MAX(Price) FROM Products);
  ```
Table sub-queries and the whole selects can be combined by set operations as they are defined in the relational algebra:

(results combined by the set operations have to be **union-compatible**)

- **UNION** to merge results of both select statements into a single results
- **INTERSECT** to filter rows that are common to results of both statements
- **EXCEPT** to filter rows that appear in results of the first statement but not in results of the second one

There are modifiers\(^2\) of the set ops. that can be put after their names:

- **ALL** – to include duplicate rows in the result, e.g., `UNION ALL`
- **CORRESPONDING** – to perform a given set operation only on the columns that are common (the same names) to results of both statements
- **CORRESPONDING BY** – to perform an op. on the named columns only

\(^2\)the modifiers may not be supported by some RDBMS, e.g., by SQLite/Web SQL
Examples on Set Operations

- List cities where are both customers and suppliers.

  ```sql
  SELECT City, Country FROM Suppliers
  INTERSECT
  SELECT City, Country FROM Customers;
  ```

- Get all customers that are from Germany or UK countries or that have ordered something, or both, and count their orders or residences in the countries.
  (that is, the count for a customer who is from UK and made one order will be 2 while the count for a customer outside Europe who made three orders will be 3)

  ```sql
  SELECT CustomerID, COUNT(*) AS Active FROM (  
    SELECT CustomerID FROM Customers  
    WHERE Country IN ('Germany', 'UK')  
    UNION ALL  
    SELECT CustomerID FROM Orders  
  ) GROUP BY CustomerID ORDER BY Active DESC;
  ```
SQL consists of DDL and DML languages.
(DDL is for schema definition, while DML is for queries and data manipulation)

SELECT statement has several clauses.
(SELECT and FROM clauses are mandatory, others are optional)

Results of SELECT queries are ordered multi-sets.
(sets that keep order of its elements and may contain duplicities)

SELECT queries can be used as nested queries in other SELECT statements.

In the next lecture:
- Structured Query Language (SQL) – Part 2
  (Common Table Expressions/CTE queries, INSERT/UPDATE/DELETE, DDL, indices, triggers, etc.)
Thank you for your attention!

Marek Rychly
<mrychly@strathmore.edu>