LECTURE 3: STRUCTURED QUERY LANGUAGE (SQL)

COMP2004J: Databases and Information Systems

Dr. Ruihai Dong (ruihai.dong@ucd.ie)
UCD School of Computer Science
Beijing-Dublin International College

SQL

- The name means Structured Query Language.
- SQL is an extremely powerful language for querying relational databases.
- SQL was first proposed in 1974 by IBM and first implemented in 1981.
- SQL was first defined as a standard in 1986 and updated in 1989, 1992 and 1999.
- Most relational systems support the basic functionality of the standard and offer custom extensions.

Different SQL Versions

- Each database management systems will implement different components of the language.
  - We will focus on the version of SQL supported by MySQL.
    - MySQL manual: http://dev.mysql.com/doc/refman/5.7/en/
  - However, we will also see some features of other RDBMS systems that MySQL does not support.

Creating, Using and Viewing Databases

- When you log in to MySQL, the first thing you must do is select a database to use.
- Each database consists of a group of tables (relations) that make up the database.
- You can find a list of databases using the following command:
  - `SHOW DATABASES;`
- To create a new database (e.g. named “my_db”):
  - `CREATE DATABASE my_db;`
- To select a database (e.g. named “my_db”):
  - `USE my_db;`

What tables are in my database?

- After selecting a database to use, you can see a list of tables (relations) in that database:
  - `SHOW TABLES;`
    - For a new database, it will display the message “Empty Set”.
- If you want to see the schema of a table, use the “describe” command (e.g. for a table called “my_table”):
  - `DESCRIBE my_table;`
- Next, we will think about how to create a table, but first we must consider the attributes that are used in it.

Domains

- We have previously discussed the concept of a **domain**: the set of all possible values that an attribute may contain.
  - Frequently defined as a data type.
- When using standard SQL we must understand the different domains and how they are defined.
- Some implementations may use different domains, or have different names for similar domains.
- There are two categories of domains
  - Elementary (predefined by the standard)
  - User-defined (not supported by MySQL)

Characters and Strings

- A **`CHAR`** data type is a fixed-length string (if the data is shorter than the length, MySQL will store space characters in the empty space).
- A **`VARCHAR`** data type is a variable-length string.

- In both cases, we must state the maximum length when we create the attribute, e.g.
  - `CHAR(30)`
  - `VARCHAR(20)`

- Some systems describe these types as "`CHARACTER`" and "`CHARACTER VARYING`".

Characters and Strings, continued.

- The **`ENUM`** type allows us to restrict an attribute to a particular set of strings, e.g.:
  - ENUM('small', 'medium', 'large')

- For very large amounts of data, you can use a **`BLOB`** (stands for **B**inary **L**arge **OB**ject) or **`TEXT`** type.

Numeric Types

- Standard integer types are **`INT`** (or `INTEGER`) or **`SMALLINT`**.
  - As an extension to the standard, MySQL also supports other integer types: `TINYINT`, `MEDIUMINT`, `BIGINT`.
- Floating point types such as **`FLOAT`** or **`DOUBLE`.**
  - These are similar to float and double datatypes in C or Java. They are not exact.
- Fixed point types (exact values up to a specific number of decimal places): **`DECIMAL`** (also called `NUMERIC`)
  - `DECIMAL(P,S)`
    - P is **total** number of digits
    - S is number of digits **after decimal point**
  - e.g. `DECIMAL(5,2)` stores values from -999.99 to 999.99

Date and Time

- Describe an instant in time.
  - **`YEAR`**
  - **`DATE`**
    - 'YYYY-MM-DD'
  - **`TIME`**
    - 'HH:MM:SS'
  - **`DATETIME`**
    - 'YYYY-MM-DD HH:MM:SS'

User Defined Domains

- User Defined Domains are useful for abstracting common attribute types between tables into a single location for maintenance.
  - They are not supported by MySQL.
- For example, an email address column may be used in several tables, all with the same properties.
- Define a domain and use that rather than setting up each table's constraints individually.

User Defined Domains

- A domain is characterised by
  - name
  - elementary domain
  - default value
  - set of constraints

- Syntax:
  - CREATE DOMAIN DomainName [as] ElementaryDomain [DefaultValue] [Constraints]

- Example:
  - CREATE DOMAIN grade AS SMALLINT DEFAULT 0 CHECK(VALUE=> 0 AND VALUE =< 100)

Default Values

- Define the value that the attribute must store when a value is not specified during row insertion.

- Syntax:
  - DEFAULT **value**

- **Value** represents a value compatible with the domain, in the form of a constant or an expression.

- Example:
  - `DEFAULT 2`
  - `DEFAULT '2016-01-01'`
  - `DEFAULT NOW()`

Tables

- An SQL table definition consists of
  - an ordered set of **attributes**
  - a (possibly empty) set of **constraints**

- Statement **`CREATE TABLE`**
  - defines a relation schema, creating an empty instance

- Syntax:
  - CREATE TABLE table_name (attribute_name DOMAIN [DEFAULT_VALUE] [CONSTRAINTS] {, attribute_name DOMAIN [DEFAULT_VALUE] [CONSTRAINTS] … } [OTHER_CONSTRAINTS])

Table Example

- employee(empno, name, job, joined, deptno)

```
CREATE TABLE employee(
  empno CHAR(8),
  name VARCHAR(20),
  job VARCHAR (20),
  joined DATE,
  deptno INT
);
```

Intra-Relational Constraints

- Constraints are conditions that must be verified by every database instance.
- Intra-relational constraints involve a single relation:
  - **NOT NULL**
    - In SQL, the special value `NULL` is used to mean “unknown”.
    - A `NOT NULL` constraint states that an attribute can never be `NULL`.
  - **UNIQUE** defines a key (i.e. the value stored in an attribute or a set of attributes cannot be repeated)
    - for single attributes: **`UNIQUE`**, after the domain
    - for multiple attributes: **`UNIQUE( attribute_name {, attribute_name } )`**
  - **PRIMARY KEY**: defines the primary key
    - There can only be one primary key for a table
    - This implies the `NOT NULL` and `UNIQUE` constraints
    - Syntax is the same as unique

Example of Intra-Relational Constraints

- Consider a table with the attributes **given_name** and **family_name**.
- If we want to have the combination of these attributes be unique we first declare them.
- Later in the other constraints section we add the constraint to state they should be unique.

```
CREATE TABLE name (
    given_name  VARCHAR(20) NOT NULL,
    family_name VARCHAR(20) NOT NULL,
    UNIQUE(given_name,family_name) );
```

Example of Intra-Relational Constraints

```
CREATE TABLE name (
    given_name  VARCHAR(20) NOT NULL,
    family_name VARCHAR(20) NOT NULL,
    UNIQUE(given_name,family_name) );
```

- Notice the difference between the above and the stricter definition below

```
CREATE TABLE name (
    given_name  VARCHAR(20) UNIQUE,
    family_name VARCHAR(20) UNIQUE );
```

Inter-Relational Constraints

- Constraints may take into account several relations.
- The most important are constraints to enforce **referential integrity**.
  - **Integrity:** The data stored in the database is accurate and consistent.
  - **Referential Integrity:** Referential integrity specified between two relations and is used to maintain the consistency among rows in the two tables. Informally, the referential integrity constrain states that a row in one table that refers to another table must refer to an existing row in that table.
    - **e.g.** a module_code attribute in a “result” relation might refer to a module_code attribute in a “module” table. Only module codes that actually exist in the “module” table can be stored in the “result” table.

Inter-Relational Constraints

- **`REFERENCES`** and **`FOREIGN KEY`** permit the definition of referential integrity constraints; syntax:
  - for single attributes
    - `REFERENCES` after the domain
  - for multiple attributes, in the “other constraints” section:
    - `FOREIGN KEY ( Attribute {, Attribute } ) REFERENCES ...`

- It is possible to associate reaction policies to violations of referential integrity.
  - i.e. what happens if the constraint becomes violated.

Example of Inter-Relational Constraints

- Setting an inter-relation constraint based on the fact that a value in one table is used as a key in another table (foreign key).

- If we were to add an attribute to a result table for storing student grades
  - `studentid INT REFERENCES student(studentid)`

- Here we are adding the attribute and stating that it references the studentid attribute in the student table.
  - Only student IDs that are stored in the “student” table can be stored.

Example of Inter-Relational Constraints

- The foreign key constraint is used in the same way but is added to the “other constraints” portion of the table definition:

- `FOREIGN KEY(studentid) REFERENCES student(studentid)`

Reaction Policies for Referential Integrity Constraints

- When we link an attribute to another table's foreign key the two rows are linked.
- In our example if we make a change to the studentid in the student table this may break our rules.
  - Meaning we have a result but no student with that ID.
- These violations may be introduced by **updates** on the referred attribute or by row **deletions**.
- These external changes may cause a **reaction** on the our table.

Reaction Policies for Referential Integrity Constraints

- There are a number of different reactions that can happen after a constraint is broken
  - **CASCADE**:
    - The change that was made in the external table should also be made here.
  - **SET NULL**:
    - The current value of the field is set to null so we are no longer connected to the other table.
  - **SET DEFAULT**:
    - Whatever the default value is for the attribute is assigned in place of the current value.
  - **NO ACTION**:
    - This prevents the change from taking place on the external table.

Reaction Policies for Referential Integrity Constraints

- Reactions may depend on the event that caused the violation: it can be an **`UPDATE`** operation (that changed the data) or a **`DELETE`** operation (that removed a row from the table)

- **`ON UPDATE CASCADE`**
- **`ON UPDATE SET NULL`**
- **`ON UPDATE SET DEFAULT`**
- **`ON UPDATE NO ACTION`**
- **`ON DELETE CASCADE`**
- **`ON DELETE SET NULL`**
- **`ON DELETE SET DEFAULT`**
- **`ON DELETE NO ACTION`**

Example of inter-relational constraint

```
CREATE TABLE student(
StudentNo INT(8) PRIMARY KEY,
FirstName VARCHAR(50) NOT NULL,
Surname VARCHAR(50) NOT NULL,
Major ENUM('SE','Iot','Fin'),
Gender ENUM('M','F') DEFAULT 'F'
) ENGINE=INNODB;
```

```
CREATE TABLE course(
CourseNo CHAR(9) PRIMARY KEY,
Title VARCHAR(50) NOT NULL
UNIQUE,
Description VARCHAR(200)
) ENGINE=INNODB;
```

```
CREATE TABLE result(
Grade DECIMAL(3,1) DEFAULT 0,
StudentNo INT(8) NOT NULL,
CourseNo CHAR(9) NOT NULL,
PRIMARY KEY(StudentNo, CourseNo),
FOREIGN KEY(StudentNo) REFERENCES
student(StudentNo) ON UPDATE CASCADE ON DELETE
CASCADE,
FOREIGN KEY(CourseNo) REFERENCES course(CourseNo)
ON UPDATE CASCADE  ON DELETE CASCADE
)ENGINE=INNODB;
```

Updating Schemas

- SQL statements:
  - ALTER TABLE (to change the attributes and/or constraints in the table)
  - DROP DATABASE (delete an entire database).
  - DROP TABLE (delete an entire table, including its contents).

- Example
  - ALTER TABLE department ADD COLUMN numberoffices INT;
  - DROP TABLE temptable;

SQL Queries

- SQL expresses queries in declarative way
  - queries specify the properties of the result, not the way to obtain it.
- Queries are translated by the query optimizer into the procedural language internal to the DBMS.
- The programmer should focus on readability, not on efficiency.

SQL Queries

- SQL queries are expressed by the **`SELECT`** statement.

- Syntax:

SELECT attr_expr [[AS] alias] {, attr_expr [[AS] alias] }
FROM table_name [[AS] alias] {, table_name [[AS] alias] }
[WHERE condition] ;

- the three parts of the query are usually called:
  - target list (the attributes you want to retrieve, and/or expressions based on these attributes).
  - from clause (the table(s) to select from)
  - where clause (the condition on which to select rows)

Examples

employee

empno	name	job	salary	deptno
1234	Sean Russell	Teacher	50000	10
4567	Jamie Heaslip	Manager	47000	10
6542	Leo Cullen	Teacher	45000	10
1238	Brendan Macken	Technician	25000	20
1555	Sean O'Brien	Designer	50000	20
1899	Brian O'Driscoll	Manager	45000	20
2525	Peter Stringer	Designer	25000	30
1585	Denis Hickey	Architect	20000	30
1345	Ronan O'Gara	Manager	29000	30

department

deptno	deptname	office	division	managernumber
10	Training	Lansdowne	D1	4567
20	Design	Belfield	D2	1899
30	Implementation	Donnybrook	D1	1345

Query Examples

- employee(empno, name, job, salary, *deptno*)
- department(deptno, deptname, office, division, managerno)

- SELECT salary FROM employee WHERE job = 'Technician';

salary
25000

- SELECT * FROM employee WHERE job = 'Manager';

empno	name	job	salary	deptno
4567	Jamie Heaslip	Manager	47000	10
1899	Brian O'Driscoll	Manager	45000	20
1345	Ronan O'Gara	Manager	29000	30

Attribute expressions

- employee(empno, name, job, salary, *deptno*)
- department(deptno, deptname, office, division, managerno)

- SELECT name, salary / 12 AS monthlysalary FROM employee;

name	monthlysalary
Sean Russell	4166.6667
Brendan Macken	2083.3333
Ronan O'Gara	2416.6667
Sean O'Brien	4166.6667
Denis Hickey	1666.6667
Brian O'Driscoll	3750
Peter Stringer	2083.3333
Jamie Heaslip	3916.6667
Leo Cullen	3750

This is an example of an "alias": giving another name to a column.

This is an example of an "attribute expression". Instead of retrieving the attribute itself, we can do some calculation with it.

Join Query

- Find the names of the employees and the office they work in:
- **`SELECT name, deptname FROM employee, department WHERE employee.deptno = department.deptno;`**

Name	DeptName
Sean Russell	Training
Brendan Macken	Design
Ronan O'Gara	Implementation
Sean O'Brien	Design
Denis Hickey	Implementation
Brian O'Driscoll	Design
Peter Stringer	Implementation
Jamie Heaslip	Training
Leo Cullen	Training

Aliases

- We can rename attributes and tables in our queries.
- We have previously seen attribute renaming.
- We can also rename tables to shorten our queries.
- `SELECT name, deptname FROM employee e, department d WHERE e.deptno = d.deptno;`

More Complex Queries

- Find the names of the employees who work in the Lansdowne office of division D1:
- **`SELECT name FROM employee e, department d WHERE e.deptno = d.deptno AND d.division = 'D1' AND d.office = 'Lansdowne';`**

name
Sean Russell
Jamie Heaslip
Leo Cullen

More Complex Queries

- Find the names of the employees who work in either the Lansdowne office or the Belfield office:

- ```
  SELECT name FROM employee e, department d
  WHERE e.deptno = d.deptno AND (d.office =
  'Belfield' OR d.office = 'Lansdowne');
  ```

name
Sean Russell
Brendan Macken
Sean O'Brien
Brian O'Driscoll
Jamie Heaslip
Leo Cullen
  ```

Operator Like

- Find the details of all employees where the second letter of their name is 'e'

- **`SELECT * FROM employee WHERE name LIKE '_e%';`**
  - the '_' represents one single character (any character)
  - the '%' represents any number of characters
  - it can be used in any place in the string

- Find the details of all employees who's job title contains exactly seven letters

- **`SELECT * FROM employee WHERE job LIKE '_______';`**