SQL – Part 1

SQL and Data Definition Language

What is SQL?

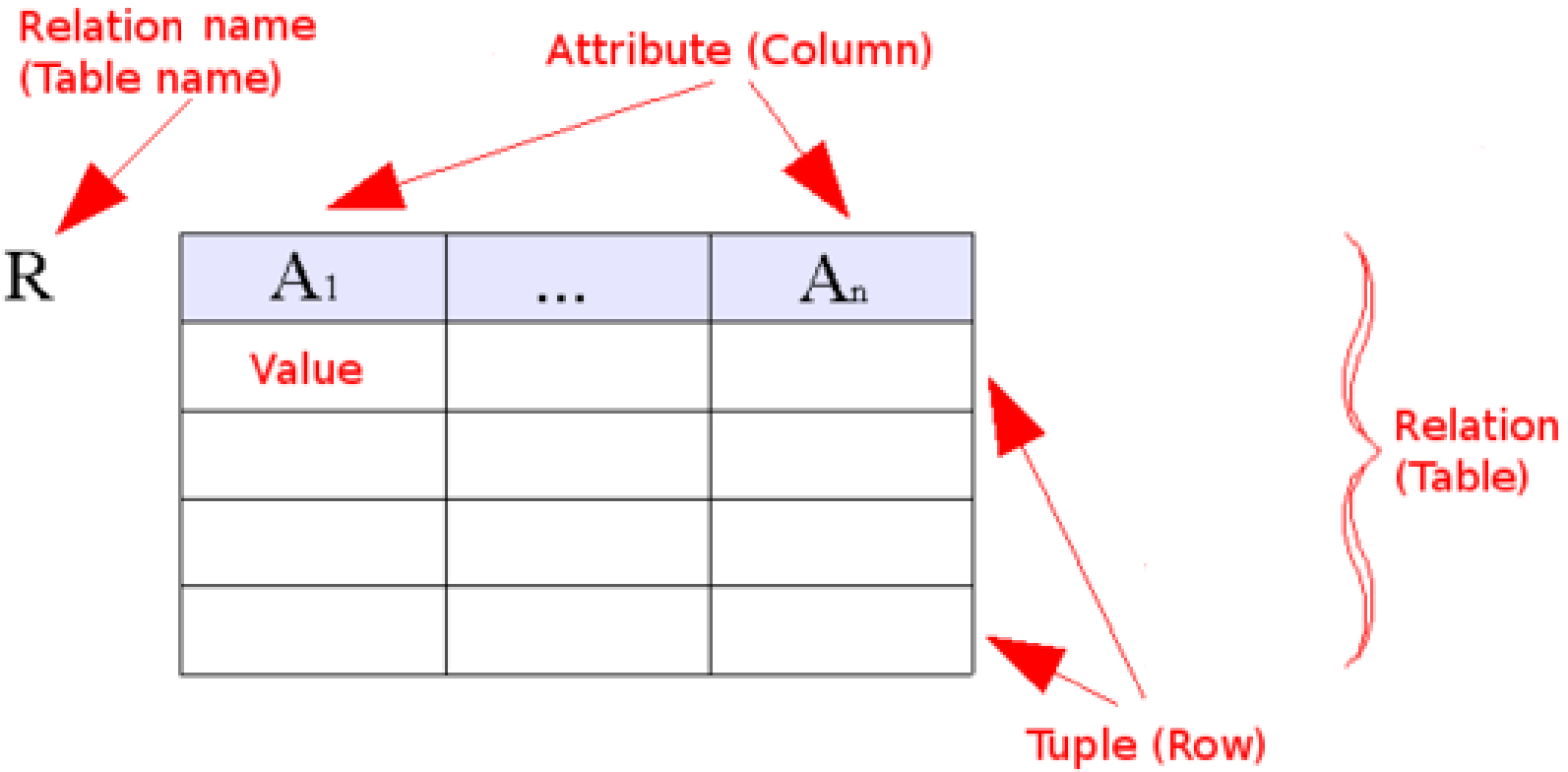
- **SQL** stands for **S**tructured **Q**uery **L**anguage
- SQL was initially developed at IBM (SEQUEL $\rightarrow$ SQL), as one of the first commercial languages for the relational data model.
  - 1986 – SQL was standardised by ANSI and ISO ($\rightsquigarrow$ SQL-86).
  - 1989 – SQL was revised ($\rightsquigarrow$ SQL-89).
  - 1992 – SQL was strengthened and much expanded ($\rightsquigarrow$ SQL-92).
  - 1999 – SQL was expanded and divided into a core specification plus optional specialised packages ($\rightsquigarrow$ SQL:1999).
  - 2003 – SQL was further expanded, e.g., XML support ($\rightsquigarrow$ SQL:2003).
  - 2011 — SQL was further expanded, e.g., improved support for temporal databases ($\rightsquigarrow$ SQL:2011).

What is SQL?

- SQL provides an interface to relational database systems, including:
  - Data Definition Language (DDL);
  - Data Manipulation Language (DML);
  - Data Control Language (DCL);
  - Transaction Control Language (TCL).

Relational Data Model and SQL

- Unlike the relational data model that is based on **sets**, SQL is based on **multisets**. It means that SQL allows a relation to have duplicate tuples.



Data Definition Language

StudentID	Name	CourseNo	Semester

Data Definition Language – Create Table

- The `CREATE TABLE` statement is used to create a new relation schema by specifying its name, its attributes and, *optionally*, its constraints.

```
CREATE TABLE table_name
       (attribute_name data_type [attribute constraints],
        ...,
        attribute_name data_type [attribute constraints],
       [table constraints]);
```

- For each attribute in a relation, we specify its name, its type and, *optionally,* a constraint specific to the attribute (i.e., attribute constraint).

```
attribute_name data_type [attribute_constraint]
```

Create Table – Example

```
CREATE TABLE Student
        (StudentID INT,
         Name VARCHAR(50),
         DoB Date,
         Email VARCHAR(100));
```

StudentID	Name	DoB	Email

```
CREATE TABLE Course
        (No VARCHAR(20),
         Cname VARCHAR(50),
         Unit SMALLINT);
```

No	Cname	Unit

```
CREATE TABLE Enrol
        (StudentID INT,
         CourseNo VARCHAR(20),
         Semester VARCHAR(50),
         Status VARCHAR(50)),
```

StudentID	CourseNo	Semester	Status

Attribute Data Types

- **Numeric types**:
  - `INT` and `SMALLINT` provide domains of integer numbers of various sizes.
  - `FLOAT` or `REAL`, and `DOUBLE PRECISION` provide floating point numbers of various precision.
  - `NUMERIC(i,j)` or `DECIMAL(i,j)` provide fixed point numbers with parameters *precision i* and *scale j*:
    - **precision** for the total number of digits;
    - **scale** for the number of digits following the decimal point.
- **String types**:
  - `CHAR(n)` allows character strings of fixed length, where *n* is the number of characters.
  - `VARCHAR(n)` allows character strings of varying length, where *n* is the maximum number of characters.
  - `BIT(n)` allows bit strings of fixed length, where *n* is the number of bits.
  - `BIT VARYING(n)` allows bit strings of varying length, where *n* is the maximum number of bits.

Attribute Data Types

- **Date and time types**:
  - `DATE` provides date values (year, month, day).
  - `TIME` provides time values (hour, minute, second).
  - `TIMESTAMP` includes the `DATE` and `TIME` fields, plus a minimum of six positions for seconds and an optional `WITH TIME ZONE` qualifier.
  - `INTERVAL` specifies a relative value that can be used to increment or decrement a value of a date, time or timestamp.
- **Boolean type**: has the values of `TRUE` or `FALSE`.
- The `CREATE DOMAIN` statement is used to create a domain that is essentially a specific data type.

```
CREATE DOMAIN domain_name AS data_type
       [default expression][constraint,...,constraint];
```

Example: `CREATE DOMAIN` ssn_type `AS CHAR(9);`

Attribute Data Types – Example

```
CREATE TABLE Student
       (StudentID INT,
        Name VARCHAR(50),
        DoB Date,
        Email VARCHAR(100));
```

StudentID	Name	DoB	Email

```
CREATE TABLE Course
       (No VARCHAR(20),
        Cname VARCHAR(50),
        Unit SMALLINT);
```

No	Cname	Unit

```
CREATE TABLE Enrol
       (StudentID INT,
        CourseNo VARCHAR(20),
        Semester VARCHAR(50),
        Status VARCHAR(50));
```

StudentID	CourseNo	Semester	Status

Attribute Constraints

- The following constraints can be specified in SQL.

NOT NULL:	specify that NULL is not allowed for an attribute.
DEFAULT:	set a default value for an attribute.
CHECK:	limit the values taken from the domain of an attribute.
UNIQUE:	ensure that uniqueness of the values for an attribute or a set of attribute in a table.
PRIMARY KEY:	uniquely identify each tuple in a table.
FOREIGN KEY:	enforce referential integrity between two tables.
INDEX:	provides accelerated access to the rows of table.

Attribute Constraints – Not Null, Default and Check

```
CREATE TABLE Course
       (No VARCHAR(20) PRIMARY KEY,
        Cname VARCHAR(50) NOT NULL,
        Unit SMALLINT NOT NULL Default 6);

CREATE TABLE Enrol
       (StudentID INT NOT NULL CHECK (StudentID>0),
        CourseNo VARCHAR(20) NOT NULL,
        Semester VARCHAR(50) NOT NULL,
        Status VARCHAR(50),
        ...);
```

- If we don't want to have missing and unknown data, we can specify NOT NULL for attributes to forbid NULL values.
- Unit of any new tuple in COURSE is set to 6 if no explicit value is provided.
- CHECK() for StudentID excludes the student IDs such as 0 and -37.

Attribute Constraints – Unique and Primary Key

```
CREATE TABLE Course
       (No VARCHAR(20) PRIMARY KEY,
        Cname VARCHAR(50) UNIQUE,
        Unit SMALLINT NOT NULL Default 6);

CREATE TABLE Enrol
       (StudentID INT NOT NULL CHECK (StudentID>0),
        CourseNo VARCHAR(20) NOT NULL,
        Semester VARCHAR(50) NOT NULL,
        Status VARCHAR(50),
        PRIMARY KEY(StudentID, CourseNo, Semester),
        ...);
```

- If a primary key contains only one attribute, PRIMARY KEY can be defined as an attribute constraint (e.g., in COURSE); otherwise it is defined as a table constraint (e.g., in ENROL).
- PRIMARY KEY specifies a key while UNIQUE specifies additional keys.

Attribute Constraints – Foreign Key

```
CREATE TABLE Student
       ( StudentID INT PRIMARY KEY,
        Name VARCHAR(50),
        DoB Date,
        Email VARCHAR(100));

CREATE TABLE Course
       ( No VARCHAR(20) PRIMARY KEY,
        Cname VARCHAR(50),
        Unit SMALLINT);

CREATE TABLE Enrol
       ( StudentID INT,
        CourseNo VARCHAR(20),
        Semester VARCHAR(50),
        Status VARCHAR(50));
```

- Every StudentID appearing in Enrol must exist in Student.
- Every CourseNo appearing in Enrol must exist in Course.

Attribute Constraints – Foreign Key

```
CREATE TABLE Student
       ( StudentID INT PRIMARY KEY,
        Name VARCHAR(50),
        DoB Date,
        Email VARCHAR(100));

CREATE TABLE Course
       ( No VARCHAR(20) PRIMARY KEY,
        Cname VARCHAR(50),
        Unit SMALLINT);

CREATE TABLE Enrol
       ( StudentID INT,
        CourseNo VARCHAR(20),
        Semester VARCHAR(50),
        Status VARCHAR(50),
        FOREIGN KEY(StudentID) REFERENCES Student(StudentID),
        FOREIGN KEY(CourseNo) REFERENCES Course(No));
```

- StudentID in Enrol references StudentID in Student.
- CourseNo in Enrol references No in Course.

Attribute Constraints – Foreign Key

```
CREATE TABLE ENROL
       ( StudentID INT,
         CourseNo VARCHAR(20),
         Semester VARCHAR(50),
         Status VARCHAR(50),
         FOREIGN KEY(StudentID) REFERENCES STUDENT(StudentID),
         FOREIGN KEY(CourseNo) REFERENCES COURSE(No));

CREATE TABLE STUDENT
       ( StudentID INT PRIMARY KEY,
         Name VARCHAR(50),
         DoB Date,
         Email VARCHAR(100));

CREATE TABLE COURSE
       ( No VARCHAR(20) PRIMARY KEY,
         Cname VARCHAR(50),
         Unit SMALLINT);
```

- Can we define ENROL before STUDENT and COURSE?

  Answer: No. ENROL has the foreign keys that reference STUDENT and COURSE.

Attribute Constraints – Index

- Indexes are used for fast retrieval based on columns other than the primary key.

```
CREATE TABLE Customer
       (CustomerID INT not null,
        Name VARCHAR(50) not null,
        DOB DATE not null,
        Address VARCHAR(80),
        Phone INT CHECK (Phone>0),
        PRIMARY KEY(CustomerID));

CREATE INDEX index1 ON  Customer (Name, DOB);

CREATE UNIQUE INDEX index2 ON Customer (Phone);
```

Data Definition Language – Alter and Drop Table

- The `ALTER TABLE` statement is used to modify an existing relation schema, including:
  - changing the name of a table;
  - adding or dropping an attribute;
  - changing the definition of an attribute;
  - adding or dropping table constraints.

- The `DROP TABLE` statement is used to remove an existing relation schema from a database schema.

Data Definition Language – Alter and Drop Table

- Add a NOT NULL constraint:

```
ALTER TABLE Customer ALTER COLUMN Address SET NOT NULL;
```

- Add a UNIQUE constraint:

```
ALTER TABLE Customer ADD UNIQUE(Phone);
```

- Add a check() constraint:

```
ALTER TABLE Customer
ADD CONSTRAINT positive_id CHECK (CustomerID > 0);
```

- Add a Foreign Key constraint:

```
ALTER TABLE Enrol
ADD FOREIGN KEY(StudentID) REFERENCES Student(StudentID);
```

Data Definition Language – Alter and Drop Table

- Add an attribute EMAIL into the table CUSTOMER:

```
ALTER TABLE Customer ADD Email VARCHAR(100);
```

- Drop the attribute EMAIL in the table CUSTOMER:

```
ALTER TABLE Customer DROP COLUMN Email;
```

- Drop the table ENROL:

```
DROP TABLE Enrol;
```

- Drop the table CUSTOMER (if exists):

```
DROP TABLE IF EXISTS Customer;
```

SQL – Part 2

Data Manipulation Language (Insert, Update, Delete)

Data Manipulation Language (DML)

- Data Manipulation Language
  - INSERT
  - UPDATE
  - DELETE
  - SELECT

Data Manipulation Language – Insert, Update, Delete

- The INSERT statement is used to add tuples into a relation.

```
INSERT INTO table_name
            [(attribute_name,...,attribute_name)]
     VALUES (value,...,value),...,(value,...,value);
```

- The UPDATE statement is used to modify attribute values of one or more selected tuples.

```
UPDATE table_name
   SET attribute_name = value,...,attribute_name = value
[WHERE selection_condition];
```

- The DELETE statement is used to remove tuples from a relation.

```
DELETE FROM table_name
     [WHERE selection_condition];
```

Insert - Examples

- The following three ways of inserting tuples into the relation STUDENT are equivalent.

```
INSERT INTO Student
     VALUES (456, 'Tom', '25/01/1988', 'tom@gmail.com'),
            (458, 'Peter', '20/02/1991', 'peter@hotmail.com');

INSERT INTO Student(Name, StudentID, DoB, Email)
     VALUES ('Tom', 456, '25/01/1988', 'tom@gmail.com'),
            ('Peter', 458, '20/02/1991', 'peter@hotmail.com');

INSERT INTO Student
     VALUES (456, 'Tom', '25/01/1988', 'tom@gmail.com');
INSERT INTO Student
     VALUES (458, 'Peter', '20/02/1991', 'peter@hotmail.com');
```

Insert - Primary Key Violation

- Suppose that we have the relation STUDENT with the primary key on StudentID:

StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com
...	...	...	...

- What would happen if we try to recycle Tom's StudentID?

```
INSERT INTO STUDENT(StudentID, Name, DoB, Email)
     VALUES (456, 'Smith', '27/08/1989', 'smith@gmail.com');
```

- DBMSs will not allow two tuples with the same primary key value in STUDENT.

Insert - Foreign Key Violation

- Consider the relations STUDENT, and ENROL with the foreign key [StudentID]$\subseteq$ STUDENT[StudentID].

StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com
459	Fran	11/09/1987	frankk@gmail.com

- If we only have the above three tuples in STUDENT, can we add the following tuple into ENROL?

```
INSERT INTO Enrol(StudentID, CourseNo, Semester, Status)
     VALUES (460, 'COMP2400', '2016 S2', 'active');
```

- Again, DBMSs will not allow a tuple in ENROL which has a student ID not appearing in any tuples of STUDENT due to the foreign key [StudentID]$\subseteq$ STUDENT[StudentID] on ENROL.

Update and Delete - Examples

- If we want to change Tom's email and name stored in the relation STUDENT, then we use

```
UPDATE Student
   SET Name='Tom Lee', Email='tom.lee@yahoo.com'
   WHERE StudentID=456;
```

- If we want to delete Tom's information from the relation STUDENT, we use

```
DELETE FROM Student WHERE StudentID=456;
```

- We can delete all the tuples in the relation STUDENT by using

```
DELETE FROM Student;
```

- **Question:** *What is the difference between the above statement and the following one?*

```
DROP Table Student;
```

- **Answer:** **The table STUDENT (empty) exists after the first statement, but would disappear if applying the second one.**

Update and Delete - Referential Actions

- Referential actions specify what happens in case of deleting or updating referenced tuples (via foreign key constraints).
- SQL offers the following possibilities:
  - `NO ACTION` (default) will throw an error if one tries to delete a row (or update the primary key value) referenced.
  - `CASCADE` will force the referencing tuples to be deleted (or updated with new primary key value).
  - `SET NULL` will force the corresponding values in the referencing tuples to be set to a null value (i.e., unknown).
  - `SET DEFAULT` will force the corresponding values in the referencing tuples to be set to a specified default value.

Referential Actions – Foreign Key

```
CREATE TABLE Student
       ( StudentID  INT PRIMARY KEY,
        Name VARCHAR(50),
        DoB Date,
        Email VARCHAR(100));
CREATE TABLE Course
       (No VARCHAR(20) PRIMARY KEY,
        Cname VARCHAR(50),
        Unit SMALLINT);
CREATE TABLE Enrol
       ( StudentID  INT,
        CourseNo VARCHAR(20),
        Semester VARCHAR(50),
        Status VARCHAR(50),
        FOREIGN KEY(StudentID) REFERENCES Student(StudentID)
        ON DELETE NO ACTION ,
        FOREIGN KEY(CourseNo) REFERENCES Course(No));
```

Referential Actions - Examples

- Consider the following foreign key defined on ENROL:

```
FOREIGN KEY(StudentID) REFERENCES STUDENT(StudentID)
ON DELETE NO ACTION
```

ENROL				
StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP1130	2016 S1	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

STUDENT			
StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

- The deletion of a student who has enrolled at least one course will throw out an error concerning the foreign key.

Referential Actions - Examples

- Consider the following foreign key defined on ENROL:

```
FOREIGN KEY(StudentID) REFERENCES STUDENT(StudentID)
ON DELETE CASCADE
```

ENROL				
StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP1130	2016 S1	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

STUDENT			
StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

- Deleting a student in STUDENT will also delete all of his enrolled courses in ENROL. We would have ENROL below after deleting the student 456.

StudentID	CourseNo	Semester	Status	EnrolDate
458	COMP1130	2016 S1	active	25/02/2016

SQL – Part 3

Data Manipulation Language (Simple SQL Queries)

Simple SQL Queries

- SQL provides the `SELECT` statement for retrieving data from a database.
- The `SELECT` statement has the following basic form:

```
   SELECT attribute_list
     FROM table_list
   [WHERE condition]
[GROUP BY attribute_list [HAVING group_condition]]
[ORDER BY attribute_list];
```

Note:

- Only `SELECT` and `FROM` are mandatory.
- The symbol $*$ means all the attributes.
- Attribute names may be qualified with the table name (required, if attribute-names are not unique).
- Attribute and table names can be given an alias.
- `DISTINCT` is used for removing duplicate tuples in the query result.

SQL Queries – Select Clause

ENROL				
StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP2600	2016 S2	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

```
SELECT * FROM Enrol;
```

StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP2600	2016 S2	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

SQL Queries – Select Clause

ENROL				
StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP2600	2016 S2	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

```
SELECT ENROL.StudentID, Semester FROM ENROL;

SELECT e.StudentID as SID, e.Semester FROM ENROL e;

SELECT DISTINCT StudentID, Semester FROM ENROL;
```

StudentID	Semester
456	2016 S2
458	2016 S1
456	2016 S2

SID	Semester
456	2016 S2
458	2016 S1
456	2016 S2

StudentID	Semester
456	2016 S2
458	2016 S1

SQL Queries – Where Clause

- Unspecified WHERE-clause means no condition.
  - all tuples of a relation in the FROM-clause are selected.
  - if multiple relations are specified in the FROM-clause without join conditions, the Cartesian product of relations is selected (**be careful**).
- The condition in the WHERE-clause can be simple or complicated.

```
SELECT * FROM Student;
SELECT * FROM Student, Course;
SELECT * FROM Student WHERE StudentID BETWEEN 100 AND 500;
SELECT * FROM Student WHERE Email is NOT NULL;
SELECT * FROM Student WHERE Email like '%@gmail.com';
```

- **Question:** *Assume that we have 1000 tuples in* STUDENT *and 100 tuples in* COURSE*. How many tuples we will have in the results of the first two queries?*
- **Answer:** **1st query result: 1000 tuples; 2nd query result: 100000 tuples.**

SQL Queries – Group By Clause

- `GROUP BY` *attribute_list* groups tuples for each value combination in the attribute_list.
- Aggregate functions can be applied to aggregate a group of attribute values into a single value, e.g.,
  - `COUNT` returns the total number of argument values
  - `AVG` returns the average of argument values
  - `MIN` returns the minimum value of the arguments
  - `MAX` returns the maximum value of the arguments
  - `SUM` returns the sum of the argument values
- We can use `HAVING` *condition* to add the condition on the groups.

SQL Queries – Group By Clause

- List the total number of courses, the sum of the units of courses, the minimum unit in COURSE

COURSE		
No	Cname	Unit
COMP1130	Introduction to Advanced Computing I	6
COMP2400	Relational Databases	6
COMP3600	Algorithms	4

```
SELECT COUNT(*), SUM(unit), MIN(unit)
  FROM Course;
```

- The query result may look like:

COUNT	SUM	MIN
3	16	4

SQL Queries – Group By Clause

- List each course offered in Semester 2 2016 together with the number of students who have enrolled in the course

```
SELECT e.CourseNo, COUNT(*) AS NumberOfStudents
  FROM ENROL e
 WHERE e.Semester = '2016 S2'
GROUP BY e.CourseNo ;
```

ENROL

StudentID	CourseNo	Semester	Status	EnrolDate
458	COMP2400	2016 S2	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	25/02/2016
...	....	....	...	....

SQL Queries – Group By Clause

- List each course offered in Semester 2 2016 together with the number of students who have enrolled in the course

```
SELECT e.CourseNo, COUNT(*) AS NumberOfStudents
  FROM Enrol e
 WHERE e.Semester = '2016 S2'
GROUP BY e.CourseNo;
```

- The query result may look like:

CourseNo	NumberOfStudents
COMP2400	120
COMP2600	100
COMP1130	150
...	...

SQL Queries – Having Clause

- List each course offered in Semester 2 2016 together with the number of students that is at least 120

```
SELECT e.CourseNo, COUNT(*) AS NumberOfStudents
  FROM Enrol e
 WHERE e.Semester = '2016 S2'
GROUP BY e.CourseNo
 HAVING COUNT(*)>= 120 ;
```

- The query result may look like:

CourseNo	NumberOfStudents
COMP2400	120
COMP1130	150
...	...

SQL Queries – Order By Clause

- The `ORDER BY` clause allows us to sort the tuples in a query result.
  - `ASC` indicates ascending order (default).
  - `DESC` indicates descending order.
- We can sort the previous result by

```
SELECT e.CourseNo, COUNT(*) AS NumberOfStudents
  FROM Enrol e
 WHERE e.Semester = '2016 S2'
GROUP BY e.CourseNo
ORDER BY NumberOfStudents DESC;
```

- This would return all tuples sorted by the number of enrolled students in descending order.

CourseNo	NumberOfStudents
COMP1130	150
COMP2400	120
COMP2600	100
...	...

SQL – Part 4

Data Manipulation Language (Advanced SQL Queries)

Advanced SQL Queries – Set Operations

- SQL incorporates several set operations: `UNION` (set union) and `INTERSECT` (set intersection), and sometimes `EXCEPT` (set difference / minus).
- Set operations result in return of a relation of tuples (no duplicates).
- Set operations apply to relations that have the same attribute types appearing in the same order, e.g., list all students who have either a gmail or hotmail email account.

```
(SELECT * FROM Student WHERE Email like '%@gmail.com')
 UNION
(SELECT * FROM Student WHERE Email like '%@hotmail.com');
```

- For example, the following query will not work

```
(SELECT StudentID, Name FROM Student)
 UNION
(SELECT Email FROM Student);
```

Advanced SQL Queries – Join Operations

- When we want to retrieve data from *more than one relations*, we often need to use **join** operations.
- Consider the following queries, which both need a join operation between two relations:
  - List the names of all courses which have been enrolled by at least one student.
  - List all students, and their enrolled courses if any.

STUDENT			
StudentID	Name	DoB	Email

COURSE		
No	Cname	Unit

ENROL				
StudentID	CourseNo	Semester	Status	EnrolDate

Advanced SQL Queries – Inner Join

- **Inner Join**: tuples are included in the result only if there is at least one matching in both relations.
- For the query "list the names of all courses which have been enrolled by at least one student", we use:

```
SELECT DISTINCT c.Cname
  FROM Course c INNER JOIN Enrol e ON c.No=e.CourseNo;
```

Course		
No	Cname	Unit
COMP2400	Relational Databases	6
COMP3900	Advanced Database Concepts	6

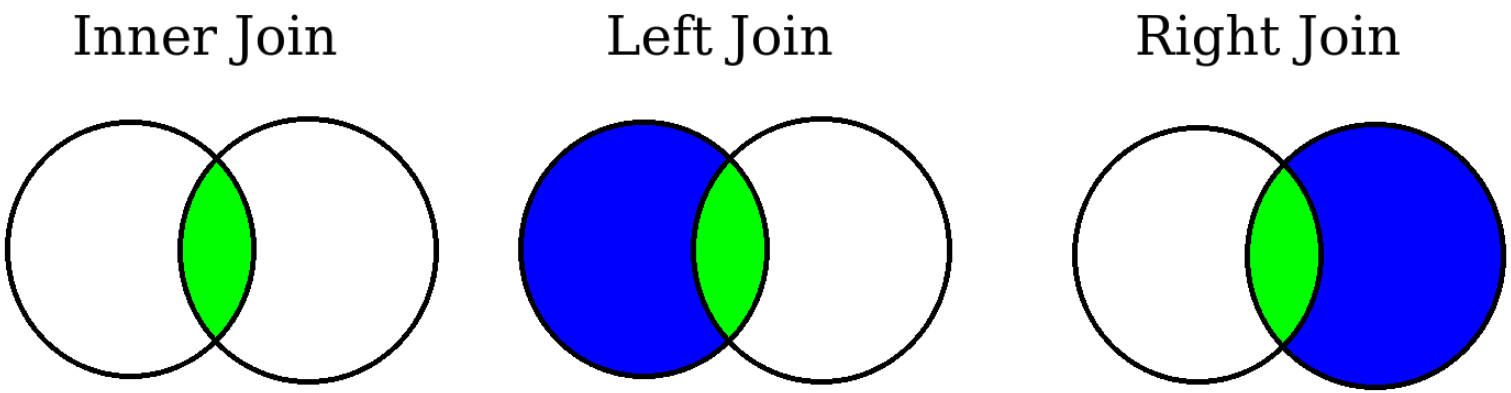
Enrol				
StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP1130	2016 S1	active	25/02/2016
458	COMP1130	2016 S1	active	25/02/2016
456	COMP2400	2016 S2	active	09/03/2016

- Result:

Cname
Relational Databases

Advanced SQL Queries – Outer Join

- **Outer Join** includes **Left Join** and **Right Join**.
- **Left/Right Join**: all tuples of the left/right table are included in the result, even if there are no matches in the relations.



Advanced SQL Queries – Outer Join

- **Left Join**: A left join retains all rows of the left table regardless of whether there is a row that matches on the right table.

ENROL1

StudentID	CourseNo	Semester
456	COMP1130	2016 S1
457	COMP1130	2016 S1
456	COMP2400	2016 S2

STUDENT

StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

```
SELECT *
  FROM STUDENT s LEFT JOIN ENROL1 e
                          ON s.StudentID=e.StudentID;
```

StudentID	Name	DoB	Email	StudentID	CourseNo	Semester
456	Tom	25/01/1988	tom@gmail.com	456	COMP1130	2016 S1
456	Tom	25/01/1988	tom@gmail.com	456	COMP2400	2016 S2
458	Peter	20/02/1991	peter@hotmail.com	null	null	null

Advanced SQL Queries – Outer Join

- **Right Join**: A right join retains all rows of the right table regardless of whether there is a row that matches on the left table.

ENROL1		
StudentID	CourseNo	Semester
456	COMP1130	2016 S1
457	COMP1130	2016 S1
456	COMP2400	2016 S2

STUDENT			
StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

```
SELECT *
 FROM STUDENT s RIGHT JOIN ENROL1 e
                          ON s.StudentID=e.StudentID;
```

StudentID	Name	DoB	Email	StudentID	CourseNo	Semester
456	Tom	25/01/1988	tom@gmail.com	456	COMP1130	2016 S1
null	null	null	null	457	COMP1130	2016 S1
456	Tom	25/01/1988	tom@gmail.com	456	COMP2400	2016 S2

Advanced SQL Queries – Outer Join

- For the query "list all students, and their enrolled courses if any", we can use either of the following statements:

```
SELECT s.*, e.CourseNo, e.Semester
  FROM STUDENT s LEFT JOIN ENROL1 e
                         ON s.StudentID=e.StudentID;

SELECT s.*, e.CourseNo, e.Semester
  FROM ENROL1 e RIGHT JOIN STUDENT s
                         ON e.StudentID=s.StudentID;
```

- If we have 1000 tuples in STUDENT, then the query result should contain at least 1000 tuples (one tuple in STUDENT may occur multiple times) with the following attributes:

StudentID	Name	DoB	Email	CourseNo	Semester
...	...	...	...	...	...

Advanced SQL Queries – Natural Join

- **Motivation**: An inner join retains all the data of the two tables for , with duplication
-
```
SELECT *
  FROM STUDENT s INNER JOIN ENROL1 e
                          On s.StudentID=e.StudentID;
```

ENROL1		
StudentID	CourseNo	Semester
456	COMP1130	2016 S1
457	COMP1130	2016 S1
456	COMP2400	2016 S2

STUDENT			
StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

- Result:

StudentID	Name	DoB	Email	StudentID	CourseNo	Semester
456	Tom	25/01/1988	tom@gmail.com	456	COMP1130	2016 S1
456	Tom	25/01/1988	tom@gmail.com	456	COMP2400	2016 S2

Advanced SQL Queries – Natural Join

- **Natural Join**: A natural join retains all the data of the two tables for only the matched rows, without duplication
- ```
  SELECT *
    FROM STUDENT s NATURAL JOIN ENROL1 e;
  ```

ENROL1

StudentID	CourseNo	Semester
456	COMP1130	2016 S1
457	COMP1130	2016 S1
456	COMP2400	2016 S2

STUDENT

StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

- Result:

StudentID	Name	DoB	Email	CourseNo	Semester
456	Tom	25/01/1988	tom@gmail.com	COMP1130	2016 S1
456	Tom	25/01/1988	tom@gmail.com	COMP2400	2016 S2
  ```

Advanced SQL Queries – Natural Join

- **Natural Join**: One kind of inner join, in which two relations are joined implicitly by comparing all attributes of the same names in both relations.
- For the query "list all students who have enrolled and their courses", use:

```
SELECT * FROM STUDENT NATURAL JOIN ENROL;
```

ENROL

StudentID	CourseNo	Semester	Status	EnrolDate
456	COMP1130	2016 S1	active	25/02/2016
457	COMP1130	2016 S1	active	25/02/2016

STUDENT

StudentID	Name	DoB	Email
456	Tom	25/01/1988	tom@gmail.com
458	Peter	20/02/1991	peter@hotmail.com

- Result:(STUDENT.StudentID=ENROL.StudentID is used in the query)

StudentID	Name	DoB	Email	CourseNo	Semester	Status	EnrolDate
456	Tom	25/01/1988	tom@gmail.com	COMP1130	2016 S1	active	25/02/2016

Advanced SQL Queries – Subqueries

- **Subqueries** are just queries that are used where a relation is required.
- Subqueries can be specified within the `FROM`-clause (usually in conjunction with aliases and renaming) to create *inline view* (exist only for the query)
- Subqueries can also be specified within the `WHERE`-clause, e.g.,
  - `IN` *subquery* tests if tuple occurs in the result of the subquery
  - `EXISTS` *subquery* tests whether the subquery results in non-empty relation
  - using `ALL`, `SOME` or `ANY` before a subquery makes subqueries usable in comparison formulae
  - in all these cases the condition involving the subquery can be negated using a preceding `NOT`

Subqueries – In

- Recall that, for the query "list all students who have enrolled and their courses", we have:

```
SELECT *
  FROM STUDENT NATURAL JOIN ENROL;
```

- Now if we want to query: "list all students who have enrolled in a course *that has less than 10 students enrolled* and the CourseNo of these courses", we have

```
SELECT s.*,e1.CourseNo
  FROM STUDENT s NATURAL JOIN ENROL e1
 WHERE e1.CourseNo IN
                (SELECT e2.CourseNo
                   FROM ENROL e2
                GROUP BY e2.CourseNo
                  HAVING COUNT(*)<10);
```

Subqueries – Exists

- For the query: “list all students who have enrolled in at least one course”, we have

```
SELECT s.*
  FROM Student s
 WHERE EXISTS (SELECT *
                 FROM Enrol e
                WHERE s.StudentID=e.StudentID);
```

- For the query: “list all students who have *not* enrolled in any course”, we have

```
SELECT s.*
  FROM Student s
 WHERE NOT EXISTS (SELECT *
                     FROM Enrol e
                    WHERE s.StudentID=e.StudentID);
```

Subqueries – More Complicated

- For the query: “list the courses that have the largest number of students enrolled in Semester 2 2016”, we have

```
SELECT e.CourseNo
  FROM (SELECT e1.CourseNo, COUNT(*) AS NoOfStudents
          FROM Enrol e1
         WHERE e1.Semester = '2016 S2'
         GROUP BY e1.CourseNo) e
 WHERE e.NoOfStudents =
            (SELECT MAX(e2.NoOfStudents)
               FROM (SELECT e1.CourseNo, COUNT(*) AS NoOfStudents
                       FROM Enrol e1
                      WHERE e1.Semester = '2016 S2'
                      GROUP BY e1.CourseNo) e2);
```

Subqueries – More Complicated

- For the query: “list all the courses that have more students enrolled than at least one other course in Semester 2 2016”, we have

```
SELECT e.CourseNo
  FROM (SELECT e1.CourseNo, COUNT(*) AS NoOfStudents
          FROM ENROL e1
         WHERE e1.Semester = '2016 S2'
         GROUP BY e1.CourseNo) e
 WHERE e.NoOfStudents > ANY
            (SELECT e2.NoOfStudents
               FROM (SELECT e1.CourseNo, COUNT(*) AS NoOfStudents
                       FROM ENROL e1
                      WHERE e1.Semester = '2016 S2'
                      GROUP BY e1.CourseNo) e2);
```

Views in SQL

- A view in SQL is a virtual table that is derived from other tables in the same database or previously defined views.
- How to Create Views?
  - Suppose we already have tables STUDENT(StudentID, Name, DoB, Email) and ENROL(StudentID, CourseNo, Semester, Status, EnrolDate). Then we can create a view ENROL1 as follows:

```
CREATE VIEW ENROL1
AS SELECT s.StudentID, s.Name, e.CourseNo, e.EnrolDate
FROM STUDENT s, ENROL e
WHERE s.StudentID=e.StudentID;
```