CSEN501 – Databases I

Topics:
Structured Query Language (SQL)

Prof. Dr. Slim Abdennadher

1.11.2012

Structured Query Language: SQL

The most used "programming language" – **extracting data**

- **Data Definition Language (DDL)**
  - Create/delete/modify relations (and views)
  - Define integrity constraints (ICs)
  - Grant/revoke privileges (security)
- **Data Manipulation language (DML)**
  - **Update language**
    - Insert/delete/modify tuples
    - Interact with ICs
  - **Query language**
    - Relationally complete!
    - Beyond relational algebra!

SQL: DDL (I)

- **Create relations**:

```
CREATE TABLE Student(sid INTEGER, sname CHAR(10), gpa REAL)
CREATE TABLE Course(cid INTEGER, cname CHAR(10), credit INTEGER,
                    teacher CHAR(10))
CREATE TABLE Enroll(sid INTEGER, cid INTEGER, grade CHAR(1))
```

- **Domain types:**
  - **Numeric data types:**
    * `INTEGER`
    * `REAL`
    * `NUMERIC(n,d)`: `n` is the total number of decimal digits and `d` is the number of digits after the decimal point.
  - **Character-string data types:**
    * `CHAR(n)`: fixed length where `n` is the number of characters.
    * `VARCHAR(n)`: varying length where `n` the maximum number of characters.
    * A literal string value is placed between **single quotation** and it is **case sensitive**.

SQL: DDL (II)

- **Bit-string data types:**
  - `BIT(n)`: fixed length `n`
  - `BITVARYING(n)`: varying length where `n` is the maximum number of bits.
  - Literal bit strings are placed between single quotes but preceded by a `B`, e.g. `B'10101'`.
- **A boolean data type:** `TRUE` or `FALSE`. In SQL because of `NULL` values, a third possible value `UNKNOWN` is added.
- Data types for **date** and **time:**
  - `DATE`: in the form `YYYY-MM-DD`, e.g. `'1967-11-11'`.
  - `TIME`: in the form `HH:MM:SS`, e.g. `'08:45:30'`.
- A **Timestamp data type:** includes both `DATE` and `TIME`.

IC: Keys

- **Key** for a relation: a minimum set of fields that uniquely identify a tuple.
  - **Candidate key:** possibly many, specified using `UNIQUE`.
  - **Primary key:** unique, specified using `PRIMARY KEY`.
- **Example:**

```
CREATE TABLE Student
(sid INTEGER,
 sname CHAR(10),
 gpa REAL,
 PRIMARY KEY (sid))
```

```
CREATE TABLE Student
  (sid INTEGER,
   sname CHAR(10),
   gpa REAL
   PRIMARY KEY (sid),
   UNIQUE (sname))
```

Foreign Keys

- **Foreign keys:** a set of fields in one relation R that is used to refer to another relation S.
- Fields should be a **key** (primary key) for S.
- In tuples of R, field values must match values in some S tuple – **no dangling pointers**.

```
CREATE TABLE Enroll
 (sid INTEGER,
  cid INTEGER,
  grade CHAR(1),
  PRIMARY KEY (sid, cid),
  FOREIGN KEY (sid) REFERENCES Student,
  FOREIGN KEY (cid) REFERENCES Course)
```

IC: Other Constraints

- `check` **condition:**

  ```
  gpa NUMERIC(2,1) check (gpa < 5.0)
  ```

- `not null` **condition:**

  ```
  sname CHAR(20) not null
  ```

- `default` **condition:**

  ```
  sname CHAR(20) DEFAULT 'Amira'
  ```

- Constraints may be given **constraint name** using the `CONSTRAINT` keyword.

  ```
  CONSTRAINT PKSTUDENT
   PRIMARY KEY (sid)
  ```

Null Values

- Attribute values in a tuple are sometimes **unknown** or **inapplicable** (e.g. no spouse's name for a single). These are treated as special value: `NULL`.
- Keys cannot have null values (but foreign keys can)
- **Three-valued logic:**
  - **Comparison operations**: e.g. `3 < null` – **unknown**.
  - **Logic connectives**:

FALSE AND UNKNOWN?	FALSE
TRUE AND UNKNOWN?	UNKNOWN
TRUE OR UNKNOWN?	TRUE
FALSE OR UNKNOWN?	UNKNOWN

Enforcing Referential Integrity

Recall: **Deletion/Update** strategies: to delete a student tuple:

- Also delete all Enroll tuples that refer to it (`CASCADE`).
- Rejection (`NO ACTION`)
- Set `sid` in Enroll tuples that refer to it to a default `sid` (null): (`SET NULL/ SET DEFAULT`).

SQL supports all of these. Default is `NO ACTION`.

```
CREATE TABLE Enroll
 (sid INTEGER,
  cid INTEGER,
  grade CHAR(1),
  PRIMARY KEY (sid, cid),
  FOREIGN KEY (sid) REFERENCES Student ON DELETE CASCADE,
  FOREIGN KEY (cid) REFERENCES Course
     ON DELETE CASCADE
     ON UPDATE SET DEFAULT)
```

Update Language: Inserting New Tuples

- **Single tuple insertion**:

```
INSERT INTO Student (sid, sname, gpa)
VALUES (1, 'Dina', 1.0)
INSERT INTO Student (sid, sname, gpa)
VALUES (2, 'Ahmed', 2.3)
INSERT INTO Student (sid, sname, gpa)
VALUES (3, 'Maria', 0.7)
```

- An insert command that causes an **IC violation** is rejected!
- **Question:** What if we tried to insert `(3, 'Ali', 1.0)`?
- **Other operation**: multiple record insertion, deletion, modification: We will come back to this topic.

Simple SQL Queries

- **Projection:** Find the names of the students:

  Recall $\pi_{\textbf{sname}}(\textbf{Student})$

  ```
  SELECT sname
  FROM Student
  ```

- **Selection:** Find the courses taught by Slim

  Recall $\sigma_{\textbf{teacher = Slim}}(\textbf{Course})$

  ```
  SELECT *
  FROM Course
  WHERE teacher = 'Slim'
  ```

Project/Select

Find the names of students with gpa less than 1.5.

$\pi_{\mathbf{sname}}(\sigma_{\mathbf{gpa} < \mathbf{1.5}}(\mathbf{Student}))$

SQL **does not** eliminate duplicates unless you ask explicitly!

```
SELECT sname
FROM   Student
WHERE  gpa < 1.5
```

```
SELECT DISTINCT sname
FROM   Student
WHERE  gpa < 1.5
```

sid	**sname**	**gpa**
1	Dina	1.0
2	Ahmed	2.3
3	Maria	0.7
4	Dina	1.0

sname
Dina
Maria
Dina

sname
Dina
Maria

Basic Syntax of SQL Queries

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE condition
```

- `relation-list` is a list of relation names, possibly with a range variable after some name.
- `attribute list` is a list of attributes of relations in relation-list. A `*` can be used to denote all attributes. You may rename the attributes
- `condition`
  - Comparison: `Attr op Const` or `Attr op Attr`
  - op: `<, >, =, <=, >=, <>`.
  - Boolean connectives: `AND, OR, NOT`.
  - Other conditions: `like` performs pattern matching in string data, e.g. `sname like 'f%'` (%: one or more characters, _: one character)
- `DISTINCT:` is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates **are not eliminated**.

Conceptual Evaluation Strategy

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE condition
```

- **Compute the cross-product** of `relation-list`.
- **Discard resulting tuples** if they do not satisfy `condition`.
- **Delete attributes** that are not in `attribute-list`.
- If `DISTINCT` is present, **eliminate duplicate tuples**.

This strategy is probably the **least efficient** way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

Example of Conceptual Evaluation – Product

```
SELECT *
FROM Student, Enroll
```

Student

sid	sname	gpa
1	Dina	1.0
2	Ahmed	2.3
3	Maria	0.7

Enroll

sid	cid	grade
1	501	A
2	502	A

Student.sid	sname	gpa	sid	cid	grade
1	Dina	1.0	1	501	A
2	Ahmed	2.3	1	501	A
3	Maria	0.7	1	501	A
1	Dina	1.0	2	502	A
2	Ahmed	2.3	2	502	A
3	Maria	0.7	2	502	A

Find the names of students who are taking Database.

$\pi_{\textbf{sname}}(\sigma_{\textbf{cid = 501}}(\textbf{Student} \bowtie \textbf{Enroll}))$

```
SELECT sname
FROM   Student, Enroll
WHERE   Student.sid = Enroll.sid AND Enroll.cid = 501
```

Question: What is the result?

Student

sid	sname	gpa
1	Dina	1.0
2	Ahmed	2.3
3	Maria	0.7

Enroll

sid	cid	grade
3	501	A
3	502	B
1	501	A

A Note on Range Variable

- Find the names of students who are taking databases
- It is a bit awkward to write `Student.sid`

```
SELECT sname
FROM   Student, Enroll
WHERE   Student.sid = Enroll.sid AND Enroll.cid = 501
```

- We can write it using **range variables**

```
SELECT  S.sname
FROM   Student  S, Enroll  E
WHERE   S.sid =  E.sid AND  E.cid = 501
```

  – Really needed only if the **same relation appears twice** in the `FROM` clause.

  – It is **good style**, however, to use range variable all the time.

A Note on Range Variable – Example

- **Example:** Find the names of students who do not have the highest gpa.
- **Relational Algebra**: **Self Join**

  $\pi_{\textbf{sname}}(\textbf{Student} \bowtie_{\textbf{gpa2}<\textbf{gpa}}$
  $\rho_{\textbf{Student2(sid2, sname2, gpa2)}}(\textbf{Student(sid, sname, gpa)}))$
- **SQL**

```
SELECT S1.sname
FROM Student S1, Student S2
WHERE S2.gpa < S1.gpa
```

More on Joins

- There is **no explicit natural join** in SQL
- Find the names of students who are taking a course taught by Slim

  $\pi_{\textbf{sname}}(\textbf{Student} \bowtie \textbf{Enroll} \bowtie \sigma_{\textbf{teacher=Slim}}(\textbf{Course}))$

  ```
  SELECT S.sname
  FROM   Student S, Enroll E, Course C
  WHERE  S.sid = E.sid AND E.cid = C.cid AND C.teacher = ‘Slim‘
  ```

  – SQL supports **conditional join**. Recall that natural join is a special case of conditional join.

  – To do natural join, you have to **explicitly** list all the equality conditions, i.e. equality on all the common fields.

Union

Find the names of students who are taking a course by Slim or Haytham.

- Using UNION:

```
SELECT S.sname
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.cid = C.cid AND C.teacher = 'Slim'
UNION                        /*  UNION ALL reserves duplicates
SELECT S.sname
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.cid = C.cid AND C.teacher = 'Haytham'
```

- You may write this using OR:

```
SELECT S.sname
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.cid = C.cid AND
       (C.teacher = 'Slim'  OR C.teacher = 'Haytham')
```

What Does Union-Compatible mean?

```
SELECT S.sid
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.sid = C.cid AND C.teacher = 'Slim'

 UNION

SELECT S.sname
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.sid = C.cid AND C.teacher = 'Haytham'
```

- What is the result of this query?
- By SQL standard, this is an **error**.

Intersection

Find the ids of the students who are taking a course taught by Slim and a course taught by Haytham.

$$\pi_{\textbf{sid}}(\textbf{Enroll} \bowtie \sigma_{\textbf{teacher = Slim}}(\textbf{Course})) \quad \cap$$
$$\pi_{\textbf{sid}}(\textbf{Enroll} \bowtie \sigma_{\textbf{teacher = Haytham}}(\textbf{Course}))$$

- Using `INTERSECT`:

```
SELECT S.sid
FROM   Enroll E, Course C
WHERE  E.cid = C.cid AND C.teacher = 'Slim'
INTERSECT
SELECT S.sid
FROM   Enroll E, Course C
WHERE  E.cid = C.cid AND C.teacher = 'Haytham'
```

- Another way: **Nested Queries**!

Set Difference

Find the ids of students who are not taking databases.

$$\pi_{\textbf{sid}}(\textbf{Student}) - \pi_{\textbf{sid}}(\sigma_{\textbf{cid=501}}(\textbf{Enroll}))$$

- Using EXCEPT or DIFFERENCE or MINUS:

```
SELECT S.sid
FROM   Student S
EXCEPT
SELECT E.sid
FROM   Enroll E
WHERE  E.cid = 501
```

- Another way: **Nested Queries**!

Nested Queries – Intersection

Find the names of students who are taking a course taught by Slim and a course taught by Haytham.

```
SELECT S.sname
FROM Student S, Enroll E, Course C
WHERE S.sid = E.sid AND E.cid = C.cid AND C.teacher = 'Slim' AND
      S.sid  IN (SELECT S2.sid
                 FROM Student S2, Enroll E2, Course C2
                 WHERE S2.sid = E2.sid AND E2.cid = C2.cid AND
                       C2.teacher = 'Haytham')
```

- A very powerful feature of SQL: a `WHERE` clause can itself contain a SQL query!
- In fact, so can `FROM` and `SELECT` clause.
- The query in `WHERE` clause is called a **subquery**.

Nested Queries – Set Difference

Find the names of students who are not taking databases.

```
SELECT S.sname
FROM Student S
WHERE S.sid  NOT IN (SELECT E.sid
                     FROM Enroll E
                     WHERE E.cid = 501)
```

Nested Queries with Correlation – EXISTS, NOT EXISTS

Find the names of students who are taking a course taught by Slim and a course taught by Haytham.

```
SELECT S.name
FROM   Student S, Enroll E, Course C
WHERE  S.sid = E.sid AND E.cid = C.cid AND C.teacher = 'Slim' AND
       EXISTS (SELECT S2. sname
               FROM   Student S2, Enroll E2, Course C2
               WHERE  S2.sid = E2.sid AND E2.cid = C2.cid AND
                      C2.teacher = 'Haytham' AND S2.sid = S.sid)
```

- **Correlation:** `S2.sid = S.sid`
- In general, **subquery** must be re-computed for each `Student` tuple `S` – **Nested loop**.
- `NOT EXISTS`: Empty Set Testing
- **Example:** Find the names of students who are not taking database.

Set Comparison Operations

- `op ANY`, `op ALL`, where `op` is `>, <, =, <>, >=, <=`.
- `ANY`: There exists some, **existential**.
- `ALL`: For All (every), **universal**.
- Find the names of students whose GPAs are higher than that of some student called Ahmed.

```
SELECT S.sname
FROM   Student S
WHERE  S.gpa < ANY (SELECT S2.gpa
                    FROM   Student S2
                    WHERE  S2.sname = 'Ahmed')
```

- What if there is no student called `Ahmed`?
  - `S.gpa < ANY ...`: returns false.
  - `S.gpa < ALL ...`: returns true.

Division – Universal Quantification

Find the names of students who are taking all courses.

- Given a student `S`: Compute the `cid`s of the courses that `S` is not taking:

```
SELECT C.cid
FROM   Course C
EXCEPT
SELECT E.cid
FROM   Enroll E
WHERE  S.sid = E.sid
```

- `S` is put in the answer if and only if the set is empty!

```
SELECT S.sname
FROM   Student S
WHERE  NOT EXISTS (SELECT C.cid
                   FROM   Course C
                   EXCEPT
                   SELECT E.cid
                   FROM   Enroll E
                   WHERE  S.sid = E.sid)
```

Using Expressions as Relation Names

- Find the names of students who are taking databases.

```
SELECT S.sname
FROM   Student S, (SELECT E.sid
                   FROM   Enroll E
                   WHERE  E.cid = 501)  AS temp
WHERE  S.sid = temp.sid
```

- Naming temporary (intermediate) relation: FROM clause can also contain subquery.

- How to rename attributes?

```
SELECT S.sname  AS name          // Or name = S.sname
FROM   Student S, (SELECT E.sid
                   FROM   Enroll E
                   WHERE  E.cid = 501)  AS temp
WHERE  S.sid = temp.sid
```

Aggregate Functions – Non-Algebraic Operators

- Significant **extension** of relational algebra:
  - `COUNT(*)`, `COUNT([DISTINCT](A))`
  - `SUM([DISTINCT](A))`
  - `AVG([DISTINCT](A))`
  - `MAX(A)`, `MIN(A)`
- Here `A` is an **attribute**.
- **Examples:**
  - ```
    SELECT COUNT(*)
    FROM   Student
    ```
  - ```
    SELECT MAX(S.gpa)
    FROM   Student S
    ```
  - ```
    SELECT AVG(S.gpa)
    FROM   Student S
    ```
    ```

Aggregate Functions – Non-Algebraic Operators (Examples)

- Find the number of students with distinct names who are taking databases:

```
SELECT COUNT(DISTINCT (S.sname))
FROM   Student S, Enroll E
WHERE  S.sid = E.sid AND E.cid = 501
```

- Find the name and GPA of the student(s) with the highest GPAs.

```
SELECT S.sname, S.gpa
FROM   Student S
WHERE  S.gpa = (SELECT MAX(S2.gpa)
                FROM   Student S2)
```

Aggregate Functions – Non-Algebraic Operators (Examples)

- Aggregate functions provide an **alternative** to `ANY` and `ALL`.
- **Example:** Find the names of students who are older than the oldest student with a gpa of 1.0.
- Using **aggregate function**:

```
SELECT S.sname
FROM   Student S
WHERE  S.age > (SELECT MAX(S2.age)
                FROM   Student S2
                WHERE  S2.gpa = 1.0)
```

- Using `All`:

```
SELECT S.sname
FROM   Student S
WHERE  S.age > ALL (SELECT S2.age
                    FROM   Student S2
                    WHERE  S2.gpa = 1.0)
```

Aggregate Functions in `SELECT` Clause – `GROUP BY`

- This query is illegal:

```
SELECT S.sname, MAX(S.gpa)
FROM   Student S
```

- Sometimes we want to apply aggregate function to each of several **groups**.
- **Example:** Find the number of students taking databases for each grade.
- For each grade (`A+, A, ..., F`), we have to write a query that looks like:

```
SELECT COUNT(E.sid)
FROM   Enroll E
WHERE  E.cid = 501 AND E.grade = 'A'
```

- **But in general, we do not know how many values (groups) we may have.**

- For each grade, find the number of students receiving that grade

```
SELECT E.grade, COUNT(E.sid)
FROM   Enroll E
WHERE  E.cid = 'database'
GROUP BY E.grade
```

- For each grade higher than 'F', find the number of databases students receiving that grade.

```
SELECT E.grade, COUNT(E.sid)
FROM   Enroll E
WHERE  E.cid = 'database'
GROUP BY E.grade
HAVING E.grade < 'F'
```

Queries with `GROUP BY`

```
SELECT [DISTINCT] target-list
FROM   relation-list
WHERE  condition
GROUP BY grouping-list
HAVING group-qualifications
```

- `target-list` contains
  - attribute lists
  - terms with aggregate functions, e.g. `MAX(S.gpa)`
- `grouping-list` is a list of attributes used to determine groups.
- Attributes in attribute list **MUST** be also in `grouping-list`.
- A **group** is a set of tuples that have the same values for all attributes in `grouping-list`.
- `group-qualifications` restrict what groups we want. It is optional.
- An attribute appears in `group-qualifications` **MUST** be also in `grouping-list`.

Conceptual Evaluation Strategy

```
SELECT [DISTINCT] target-list
FROM   relation-list
WHERE  condition
GROUP BY grouping-list
HAVING group-qualifications
```

- **Compute the cross-product** of `relation-list`.
- **Discard resulting tuples** if they do not satisfy `condition`.
- **Delete attributes** that are not in `attribute-list`.
- **Divide the remaining tuples** into groups by the value of attributes of `grouping-list`.
- **Eliminate some groups** by applying `group-qualifications`.

- Find the names and grades of database students ordered by their grades.

```
SELECT S.sname, E.grade
FROM   Student S, Enroll E
WHERE  S.sid = E.sid AND E.cid = 501
ORDER BY E.grade
```

- Find the names and grades of database students ordered first by their grades and within each grade ordered by names.

```
SELECT S.sname, E.grade
FROM   Student S, Enroll E
WHERE  S.sid = E.sid AND E.cid = 501
ORDER BY E.grade, S.sname
```

- `ASC` and `DESC`: Default is `ASC`

Impact of Null Values on SQL Constructs

- `WHERE` clause eliminates rows in which the condition does not evaluate to `true`.
- **Duplicates**: Two rows are duplicates if the corresponding columns are either equal or both contain null values.
- **Comparison**: Two null values when compared using `=` is `unknown`.
- Arithmetic operations all return `null` if one of their operands is `null`.
- `COUNT(*)` treats null values like other values (included in the count).
- All other operators `COUNT, SUM, MIN, MAX, AVG` and variations using `DISTINCT` discard null values.
- When applied to only null values, then the result is `null`.

More on SQL DDL – Complex Integrity Constraints I

- Table constraint over a single table using `CHECK`.
- **Example:** To ensure that the gpa must be a value in the range `0.7` to `5`.

```
CREATE TABLE Student(sid INTEGER,
                     sname CHAR(10),
                     gpa REAL,
                     CHECK (gpa >= 0.7 AND gpa < 5.0))
```

Complex Integrity Constraints II

- To enforce that a student can not be enrolled in `'database'`, we use the following:

```
CREATE TABLE Enroll(sid INTEGER,
                    cid INTEGER,
                    grade CHAR(1),
                    PRIMARY KEY (sid, cid),
                    FOREIGN KEY (sid) REFERENCES Student,
                    FOREIGN KEY (cid) REFERENCES Course,
                    CONSTRAINT noDatabase
                    CHECK ('Database' <>
                              (SELECT C.cname
                               FROM   Course C
                               WHERE  Enroll.cid = C.cid)))
```

- When a row is inserted into `Enroll` or an existing row is modified, the expression in the `CHECK` condition is evaluated and the command is rejected if the expression evaluates to false.

Specifying General Constraints as Assertions

- **Assertions:** Integrity Constraints over several tables.
- **Example:** The number of courses and number of teachers should be less than 50.
- **Assertion:**

```
CREATE ASSERTION smallUniversity
CHECK ((    SELECT COUNT(C.cid) FROM Course C)
        + ( SELECT COUNT(T.tid) FROM Teacher T)
        < 50 )
```

More on SQL DDL – Views: Virtual Tables

- A **view** is a relation but we store a **definition** instead of a set of tuples.
- **Example:** Student names and grades of all students who are attending the database course:

```
CREATE VIEW CourseDB(sname, grade)
        AS SELECT S.sname, E.grade
                       FROM   Student S, Enroll E
                       WHERE  S.sid = E.sid AND E.cid = 501
```

- Views are a way of specifying a table that we need to reference frequently rather than specifying the join every time.
- If a view is not needed anymore, we can use the DROP command to dispose of it.

```
DROP VIEW CourseDB
```