Introduction to SQL

CMPT 354
Jian Pei
jpei@cs.sfu.ca
Outline

• Overview of the SQL Query Language
• SQL Data Definition
• Basic Query Structure of SQL Queries
• Additional Basic Operations
• Set Operations
• Null Values
• Aggregate Functions
• Nested Subqueries
• Modification of the Database
Where Did SQL Come From?

• IBM Sequel language developed as part of the System R project at the IBM San Jose Research Laboratory (70s)
• Renamed Structured Query Language (SQL) (1981)
• ANSI and ISO standard SQL
  • SQL-86 – ANSI standard
  • SQL-89 – added integrity constraints
  • SQL-92 – major revision, adding new data types, character sets, scalar/set operations, conditional expressions, ...
  • SQL:1999 (language name became Y2K compliant!) – added regular expression matching, recursive queries (e.g. transitive closure), triggers, ...
  • SQL:2003 – introduced XML related features, window function, ...
• Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features
• Not all examples here may work on your particular system
SQLite

• We use SQLite in this course <https://www.sqlite.org/index.html>
  • a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine
  • the most used database engine in the world
• Download it and install it in your computer in this week, please
• Alternatively, use the online SQL interpreter based on SQLite/sql.js <https://www.db-book.com/db7/university-lab-dir/sqljs.html>
• The database used in the textbook is available at <https://www.db-book.com/db7/university-lab-dir/univdb-sqlite.db>
• Many useful documents/tutorials online
• If you have any questions about setting up or using SQLite in your computer, please come to our office hours
SQL Parts

• DML – provide the ability to query information from a database and to insert tuples into, delete tuples from, and modify tuples in the database
• Integrity – the DDL includes commands for specifying integrity constraints
• View definition – the DDL includes commands for defining views
• Transaction control – commands for specifying the beginning and ending of transactions
• Embedded SQL and dynamic SQL – define how SQL statements can be embedded within general-purpose programming languages
• Authorization – commands for specifying access rights to relations and views
Data Definition Language (DDL)

- The SQL data-definition language (DDL) allows the specification of information about relations, including
  - The schema for each relation
  - The type of values associated with each attribute
  - The Integrity constraints
  - The set of indices to be maintained for each relation
  - Security and authorization information for each relation
  - The physical storage structure of each relation on disk
Domain Types in SQL

• **char(n)** – fixed length character string, with user-specified length $n$
• **varchar(n)** – variable length character strings, with user-specified maximum length $n$
• **int** – integer (a finite subset of the integers that is machine-dependent)
• **smallint** – small integer (a machine-dependent subset of the integer domain type)
• **numeric(p,d)** – fixed point number, with user-specified precision of $p$ digits, with $d$ digits to the right of decimal point. (ex., `numeric(3,1)`, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
• **real, double precision** – floating point and double-precision floating point numbers, with machine-dependent precision
• **float(n)** – floating point number, with user-specified precision of at least $n$ digits
• More to come later
Create Table Construct

• An SQL relation is defined using the `create table` command
  
  ```sql
  create table r
  (A_1 D_1, A_2 D_2, ..., A_n D_n,
   (integrity-constraint_1),
   ...
   (integrity-constraint_k))
  ```

  • `r` is the name of the relation
  • each `A_i` is an attribute name in the schema of relation `r`
  • `D_i` is the data type of values in the domain of attribute `A_i`

• Example:

  ```sql
  create table instructor ( 
  ID               char(5),
  name             varchar(20),
  dept_name        varchar(20),
  salary           numeric(8,2));
  ```
Integrity Constraints in Create Table

• Types of integrity constraints
  • **primary key** \((A_1, \ldots, A_n)\)
  • **foreign key** \((A_m, \ldots, A_n)\) references \(r\)
  • **not null**

• SQL prevents any update to the database that violates an integrity constraint

• Example

```sql
create table instructor (
    ID char(5),
    name varchar(20) not null,
    dept_name varchar(20),
    salary numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department);
```
More Relation Definitions

create table student (  
  ID varchar(5),  
  name varchar(20) not null,  
  dept_name varchar(20),  
  tot_cred numeric(3,0),  
  primary key (ID),  
  foreign key (dept_name) references department);

create table takes (  
  ID varchar(5),  
  course_id varchar(8),  
  sec_id varchar(8),  
  semester varchar(6),  
  year numeric(4,0),  
  grade varchar(2),  
  primary key (ID, course_id, sec_id, semester, year) ,  
  foreign key (ID) references student,  
  foreign key (course_id, sec_id, semester, year) references section);
create table course (
    course_id      varchar(8),
    title          varchar(50),
    dept_name      varchar(20),
    credits        numeric(2,0),
    primary key (course_id),
    foreign key (dept_name) references department);

To-Do List

• Suppose we want to create two tables
  student (stud-id, name, address, supervisor-id);
  supervisor(supervisor-id, name, address, student-id);

• Foreign key constraints
  • Attribute supervisor-id in table student is the foreign key referencing table supervisor
  • Attribute student-id in table supervisor is the foreign key referencing table student

• How to create those two tables?
• What difficulties may those two tables lead to?
• Is this a good design? If yes, why? If not, how to improve it?
Updates to Tables

- **Insert**
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`

- **Delete**
  - Remove all tuples from the *student* relation
    - `delete from student`

- **Drop Table**
  - `drop table r`

- **Alter**
  - `alter table r add A D`
    - *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*
    - All existing tuples in the relation are assigned *null* as the value for the new attribute
  - `alter table r drop A`
    - *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases
Basic Query Structure

• A typical SQL query has the form:

```
select A_1, A_2, ..., A_n
from r_1, r_2, ..., r_m
where P
```

• $A_i$ represents an attribute
• $R_i$ represents a relation
• $P$ is a predicate
• The result of an SQL query is a relation
The select Clause

• The select clause lists the attributes desired in the result of a query
  • Corresponds to the projection operation of the relational algebra

• Example: find the names of all instructors: \( \prod_{\text{name}} (\text{instructor}) \)
  
  select name 
  from instructor

• NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters)
  • E.g., Name \(\equiv\) NAME \(\equiv\) name
  • Some people use upper case wherever we use bold font
The select Clause

- SQL allows duplicates in relations as well as in query results
  - Does relational algebra allow duplicates?
- To force the elimination of duplicates, use the keyword `distinct` after `select`
- Find the department names of all instructors, and remove duplicates
  
  ```
  select distinct dept_name
  from instructor
  ```
- The keyword `all` specifies that duplicates should NOT be removed
  
  ```
  select all dept_name
  from instructor
  ```
- Relational algebra: \( \Pi_{dept\_name} (instructor) \)

J. Pei: CMPT 354 -- Introduction to SQL
The select Clause

• An asterisk in the select clause denotes “all attributes”

```sql
select *
from instructor
```

• An attribute can be a literal with no `from` clause

```sql
select '437'
```
• Results is a table with one column and a single row with value “437”
• Can give the column a name using:

```sql
select '437' as FOO
```

• An attribute can be a literal with `from` clause

```sql
select 'A'
from instructor
```
• Result is a table with one column and $N$ rows (number of tuples in the `instructors` table), each row with value “A”
To-Do List

• What is the output of the following query? Why?
  `select 'a' 'A', 'b' 'B'`

• What is the output of the following query? Why?
  `select 'a' 'A', 'b' 'B'
  from instructor`

• What is the output of the following query? Why?
The select Clause

- The **select** clause can contain arithmetic expressions involving the operation, +, −, *, and /, and operating on constants or attributes of tuples
  - Query
    ```sql
    select ID, name, salary/12
    from instructor
    ```
  - Result: a relation that is the same as the `instructor` relation, except that the value of the attribute `salary` is divided by 12
  - Can rename “salary/12” using the **as** clause:
    ```sql
    select ID, name, salary/12 as monthly_salary
    ```
The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra
- To find all instructors in Comp. Sci. dept: \( \sigma_{\text{dept\_name}=\text{"Comp. Sci."} } (\text{instructor}) \)
  
  \[
  \begin{align*}
  &\text{select name} \\
  &\text{from instructor} \\
  &\text{where dept\_name = 'Comp. Sci.'}
  \end{align*}
  \]
- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 70000: \( \sigma_{\text{dept\_name}=\text{"Physics"} \land \text{salary} > 70000 } (\text{instructor}) \)
  
  \[
  \begin{align*}
  &\text{select name} \\
  &\text{from instructor} \\
  &\text{where dept\_name = 'Comp. Sci.'\ and salary > 70000}
  \end{align*}
  \]
The from Clause

• The **from** clause lists the relations involved in the query
  • Corresponds to the Cartesian product operation of the relational algebra
• Find the Cartesian product *instructor X teaches*
  
  ```
  select *
  from instructor, teaches
  ```
  
  • Generate every possible instructor-teaches pair, with all attributes from both relations
  • (depending on specific SQL implementation) For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
• Cartesian product is not very useful directly, but is useful combined with where-clause condition (selection operation in relational algebra)
Examples

- Find the names of all instructors who have taught some course and the course_id
  
  ```
  select name, course_id
  from instructor, teaches
  where instructor.ID = teaches.ID
  ```

- Find the names of all instructors in the Art department who have taught some course and the course_id
  
  ```
  select name, course_id
  from instructor, teaches
  where instructor.ID = teaches.ID
  and instructor.dept_name = 'Art'
  ```
The Rename Operation

- The SQL allows renaming relations and attributes using the as clause:
  
  \[
  \text{old-name as new-name}
  \]

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  
  \[
  \text{select distinct T.name from instructor as T, instructor as S where T.salary > S.salary and S.dept_name = 'Comp. Sci.'}
  \]

- Keyword as is optional and may be omitted
  
  \[
  \text{instructor as T \equiv instructor T}
  \]
To-Do List

• Relation \textit{emp-super}
• Find the supervisor of “Bob”
• Find the supervisor of the supervisor of “Bob”
• Can you find ALL the supervisors (direct and indirect) of “Bob”?  

\begin{center}
\begin{tabular}{|l|l|}
\hline
\text{person} & \text{supervisor} \\
\hline
Bob & Alice \\
Mary & Susan \\
Alice & David \\
David & Mary \\
\hline
\end{tabular}
\end{center}
To-Do List

• Find the department names which has at least one instructor whose salary is at least 80000
• Find the pair of instructor names \((x, y)\) such that \(x\) and \(y\) work in the same department
• Find the names of instructors who teaches the same course twice
String Operations

• SQL includes a string-matching operator for comparisons on character strings
• The operator `like` uses patterns that are described using two special characters:
  • The `%` character matches any substring, including empty
  • The `_` character matches any single character
• Find the names of all instructors whose name includes the substring “dar”
  ```sql
  select name
  from instructor
  where name like '%dar%'
  ```
• Use backslash (\) as the escape character
  • Match the string “100%”
  ```sql
  like '100\%' escape '\'
  ```
String Operations

- Patterns are case sensitive
- Pattern matching examples:
  - 'Intro%' matches any string beginning with “Intro”
  - '%Comp%' matches any string containing “Comp” as a substring
  - '_ _ _' matches any string of exactly three characters
  - '_ _ _ %' matches any string of at least three characters
- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.
Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  
  ```sql
  select distinct name
  from instructor
  order by name
  ```

- We may specify `desc` for descending order or `asc` for ascending order, for each attribute; ascending order is the default
  
  - Example: `order by name desc`

- Can sort on multiple attributes
  
  - Example: `order by dept_name, name`
  
  - First by dept_name, sort all tuples having the same dept_name by name
Where Clause Predicates

- SQL includes a `between` comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, $\geq 90,000$ and $\leq 100,000$)

```sql
select name
from instructor
where salary between 90000 and 100000
```

- Tuple comparison

```sql
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```
Set Operations

• Find courses that ran in Fall 2017 or in Spring 2018
  \[
  \text{(select course_id from section where sem = 'Fall' and year = 2017)}
  \text{union}
  \text{(select course_id from section where sem = 'Spring' and year = 2018)}
  \]

• Find courses that ran in Fall 2017 and in Spring 2018
  \[
  \text{(select course_id from section where sem = 'Fall' and year = 2017)}
  \text{intersect}
  \text{(select course_id from section where sem = 'Spring' and year = 2018)}
  \]

• Find courses that ran in Fall 2017 but not in Spring 2018
  \[
  \text{(select course_id from section where sem = 'Fall' and year = 2017)}
  \text{except}
  \text{(select course_id from section where sem = 'Spring' and year = 2018)}
  \]
Set Operations

• Set operations **union**, **intersect**, and **except**
  • Each of the above operations automatically eliminates duplicates
• To retain all duplicates, use the
  • **union all**,  
  • **intersect all**  
  • **except all**
To-Do List

• Find all instructors and their ids who did not teach any courses
Null Values

• It is possible for tuples to have a null value, denoted by null, for some of their attributes
  • null signifies an unknown value or that a value does not exist
• The result of any arithmetic expression involving null is null
  • Example: 5 + null returns null
• The predicate is null can be used to check for null values
  • Example: Find all instructors whose salary is null
    
    ```sql
    select name
    from instructor
    where salary is null
    ```
• The predicate is not null succeeds if the value on which it is applied is not null
Null Values

• SQL treats as unknown the result of any comparison involving a null value (other than predicates is null and is not null).
  • Example: 5 < null or null <> null or null = null

• The predicate in a where clause can involve Boolean operations (and, or, not); thus the definitions of the Boolean operations need to be extended to deal with the value unknown.
  • and: (true and unknown) = unknown,
    (false and unknown) = false,
    (unknown and unknown) = unknown
  • or: (unknown or true) = true,
    (unknown or false) = unknown
    (unknown or unknown) = unknown

• Result of where clause predicate is treated as false if it evaluates to unknown
To-Do List

• What is the output of the following query? Why?
  
  ```sql
  select 5+null '5+null'
  ```

• What is the output of the following query? Why?
  
  ```sql
  select 5+null '5+null'
  from instructor
  ```
Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
  
  **avg**: average value
  **min**: minimum value
  **max**: maximum value
  **sum**: sum of values
  **count**: number of values
Aggregate Functions Examples

• Find the average salary of instructors in the Computer Science department
  
  ```sql
  select avg(salary)
  from instructor
  where dept_name = 'Comp. Sci.';
  ```

• Find the total number of instructors who teach a course in the Spring 2018 semester
  
  ```sql
  select count (distinct ID)
  from teaches
  where semester = 'Spring' and year = 2018;
  ```

• Find the number of tuples in the course relation
  
  ```sql
  select count (*)
  from course;
  ```
To-Do List

• Find the name of the instructor and the salary who has the highest salary

• What is the output of the follow query? Why?

    select name, avg(salary)
    from instructor
Aggregate Functions – Group By

- Find the average salary of instructors in each department

```sql
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>
Aggregation

• Attributes in select clause outside of aggregate functions must appear in group by list

/* erroneous query */
select dept_name, ID, avg(salary)
from instructor
group by dept_name;
To-Do List

• What is the output of the following query? Why?
  
  ```sql
  select dept_name, name, avg(salary)
  from instructor
  group by dept_name
  ```
Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000
  ```sql
  select dept_name, avg(salary) as avg_salary
  from instructor
  group by dept_name
  having avg(salary) > 42000;
  ```

- Predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups
To-Do List

• Find all instructors who taught at least 2 sections
• Can you use aggregate function count() to find all instructors who did not teach any courses?
Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries
- A subquery is a select-from-where expression that is nested within another query
- The nesting can be done in the following SQL query

```
select A_1, A_2, ..., A_n
from r_1, r_2, ..., r_m
where P
```

as follows:
- **From clause:** \( r_i \) can be replaced by any valid subquery
- **Where clause:** \( P \) can be replaced with an expression of the form:
  \[ B \text{ <operation>} (\text{subquery}) \]
  \( B \) is an attribute and <operation> to be explained later
- **Select clause:**
  \( A_i \) can be replaced by a subquery that generates a single value
Set Membership

• Find courses offered in Fall 2017 and in Spring 2018
  
  ```
  select distinct course_id
  from section
  where semester = 'Fall' and year= 2017 and
  course_id in (select course_id
                 from section
                 where semester = 'Spring' and year= 2018);
  ```

• Find courses offered in Fall 2017 but not in Spring 2018
  
  ```
  select distinct course_id
  from section
  where semester = 'Fall' and year= 2017 and
  course_id not in (select course_id
                     from section
                     where semester = 'Spring' and year= 2018);
  ```
Set Comparison – “some” Clause

• Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department

\[
\text{select distinct } T.\text{name} \\
\text{from instructor as } T, \text{ instructor as } S \\
\text{where } T.\text{salary} > S.\text{salary} \text{ and } S.\text{dept_name} = 'Biology';
\]

• Same query using > some clause

\[
\text{select name} \\
\text{from instructor} \\
\text{where salary} > \text{some (select salary} \\
\text{from instructor} \\
\text{where dept_name} = 'Biology');
\]
Definition of “some” Clause

• \( F <\text{comp}> \text{some} \ r \iff \exists t \in r \text{ such that } (F <\text{comp}> t) \)
  
  • \( <\text{comp}> \) can be: \(<, \leq, >, =, \neq\)

  \[
  \begin{array}{c|c}
    5 & 6 \\
    \hline
    5 & 0 \\
  \end{array}
  \]

  (read: \( 5 < \text{some tuple in the relation} \))

  \[
  \begin{array}{c|c}
    5 & 0 \\
    \hline
    0 & 5 \\
  \end{array}
  \]

  (\( 5 < \text{some} \) ) = false

  \[
  \begin{array}{c|c}
    5 & 0 \\
    \hline
    0 & 5 \\
  \end{array}
  \]

  (\( 5 = \text{some} \) ) = true

  \[
  \begin{array}{c|c}
    5 & 0 \\
    \hline
    0 & 5 \\
  \end{array}
  \]

  (\( 5 \neq \text{some} \) ) = true (since \( 0 \neq 5 \))

(\( = \text{some} \) \( \equiv \) in

However, (\( \neq \text{some} \) \( \neq \) not in)
Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department

```sql
select name
from instructor
where salary > all (select salary
                     from instructor
                     where dept name = 'Biology');
```
Definition of “all” Clause

• \( F \ <\text{comp}> all \ r \Longleftrightarrow \forall t \in r \ (F <\text{comp}> t) \)

\[
\begin{array}{c|c}
(5 < all) & = false \\
\hline
0 & 5 \\
5 & 6 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
(5 < all) & = true \\
\hline
6 & 10 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
(5 = all) & = false \\
\hline
4 & 5 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
(5 \not= all) & = true \ (\text{since } 5 \not= 4 \text{ and } 5 \not= 6) \\
\hline
4 & 6 \\
\hline
\end{array}
\]

\( (\not= all) \equiv \text{not in} \)

However, \((= all) \not\equiv in\)
Test for Empty Relations

• The **exists** construct returns the value **true** if the argument subquery is nonempty.

  • **exists**  $r \iff r \neq \emptyset$

  • **not exists**  $r \iff r = \emptyset$
Use of “exists” Clause

• Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```sql
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
exists (select *
    from section as T
    where semester = 'Spring' and year = 2018
    and S.course_id = T.course_id);
```

• **Correlation name** – variable S in the outer query

• **Correlated subquery** – the inner query
Use of “not exists” Clause

• Find all students who have taken all courses offered in the Biology department
  
  ```sql
  select distinct S.ID, S.name
  from student as S
  where not exists ( (select course_id
  from course
  where dept_name = 'Biology')
  except
  (select T.course_id
  from takes as T
  where S.ID = T.ID));
  ```

• The first nested query lists all courses offered in Biology
• The second nested query lists all courses a particular student took

• $X - Y = \emptyset \iff X \subseteq Y$

• Cannot write this query using = all and its variants
Test for Absence of Duplicate Tuples

• The **unique** construct tests whether a subquery has any duplicate tuples in its result.

• The **unique** construct evaluates to “true” if a given subquery contains no duplicates.

• Find all courses that were offered at most once in 2017

```sql
select T.course_id
from course as T
where unique ( select R.course_id
from section as R
where T.course_id = R.course_id
and R.year = 2017 );
```
To-Do List

• Find the course titles and the students take the courses
  • Can you write it using join and no subqueries?
  • Can you write it using subqueries?
Subqueries in the Form Clause

- SQL allows a subquery expression to be used in the `from` clause
- Find the average instructors’ salaries of those departments where the average salary is greater than $42,000

  ```sql
  select dept_name, avg_salary
  from ( select dept_name, avg(salary) as avg_salary
          from instructor
          group by dept_name
        ) as dept_avg
  where avg_salary > 42000;
  ```
- We do not need to use the `having` clause
- Another way to write the above query

  ```sql
  select dept_name, avg_salary
  from ( select dept_name, avg(salary) from instructor
          group by dept_name
        ) as dept_avg
  where avg_salary > 42000;
  ```
With Clause

• The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs

• Find all departments with the maximum budget

  ```sql
  with max_budget (value) as
  (select max(budget) from department)
  select department.name
  from department, max_budget
  where department.budget = max_budget.value;
  ```
Complex Queries using With Clause

• Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```
To-Do List

• Find the students who has the highest scores in at least 2 courses
Scalar Subquery

• Scalar subquery is one which is used where a single value is expected
• List all departments along with the number of instructors in each department

```sql
select dept_name,
    ( select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
    as num_instructors
from department;
```

• Runtime error if subquery returns more than one result tuple
Ranking

- Ranking is done in conjunction with an order by specification
- Suppose we are given a relation `student_grades(ID, GPA)` giving the grade-point average of each student
- Find the rank of each student
  \[
  \text{select } ID, \text{ rank()} \over (\text{order by } GPA \text{ desc}) \text{ as } s_{\text{rank}} \\
  \text{from } student\_grades
  \]
- An extra `order by` clause is needed to get them in sorted order
  \[
  \text{select } ID, \text{ rank()} \over (\text{order by } GPA \text{ desc}) \text{ as } s_{\text{rank}} \\
  \text{from } student\_grades \\
  \text{order by } s_{\text{rank}}
  \]
- Ranking may leave gaps: if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - `dense_rank` does not leave gaps, so next dense rank would be 2
Ranking through Basic SQL Aggregation

• Ranking can be done using basic SQL aggregation, but the query is very inefficient

```sql
select ID, (1 + (select count(*)
    from student_grades B
    where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```
Ranking within Partitions

- Ranking can be done within partition of the data
- Find the rank of students within each department

```sql
select ID, dept_name,
    rank () over (partition by dept_name order by GPA desc)
    as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

- Multiple `rank` clauses can occur in a single `select` clause
- Ranking is done `after` applying `group by` clause/aggregation
- Can be used to find top-n results
  - More general than the `limit n` clause supported by many databases, since it allows top-n within each partition
Other Ranking Functions

- Other ranking functions:
  - `percent_rank` (within partition, if partitioning is done)
  - `cume_dist` (cumulative distribution)
    - fraction of tuples with preceding values
  - `row_number` (non-deterministic in presence of duplicates)

- SQL:1999 permits a user to specify **nulls first** or **nulls last**
  
  ```sql
  select ID,
  rank ( ) over (order by GPA desc nulls last) as s_rank
  from student_grades
  ```
Windowing

- Smooth out random variations

- **moving average**: Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day

- **Window specification** in SQL:
  - Given relation \textit{sales}(date, value)

    \begin{verbatim}
    select date, sum(value) over 
    (order by date between rows 1 preceding and 1 following) 
    from sales
    \end{verbatim}
Windowing within Partitions

- Given a relation `transaction (account_number, date_time, value)`, where value is positive for a deposit and negative for a withdrawal
- Find total balance of each account after each transaction on the account

```sql
select account_number, date_time, 
    sum (value) over 
    (partition by account_number 
     order by date_time 
     rows unbounded preceding) 
as balance
from transaction
order by account_number, date_time
```
Modification of the Database

• Deletion of tuples from a given relation.
• Insertion of new tuples into a given relation
• Updating of values in some tuples in a given relation
Deletion

• Delete all instructors
  \[\texttt{delete from instructor}\]

• Delete all instructors from the Finance department
  \[\texttt{delete from instructor} \quad \texttt{where dept\_name} = 'Finance';\]

• Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building
  \[\texttt{delete from instructor} \quad \texttt{where dept\_name} \in (\texttt{select dept\_name} \quad \texttt{from department} \quad \texttt{where building} = 'Watson');\]
Deletion

• Delete all instructors whose salary is less than the average salary of instructors
  \[\text{delete from instructor where salary < (select avg (salary) from instructor)};\]

• Problem: as we delete tuples from instructor, the average salary changes

• Solution used in SQL:
  1. First, compute \text{avg} (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing \text{avg} or retesting the tuples)
Insertion

• Add a new tuple to course

\[
\text{insert into course values (}'CS-437', 'Database Systems', 'Comp. Sci.', 4);\]

• or equivalently

\[
\text{insert into course (course_id, title, dept_name, credits) values (}'CS-437', 'Database Systems', 'Comp. Sci.', 4);\]

• Add a new tuple to student with tot_creds set to null

\[
\text{insert into student values (}'3003', 'Green', 'Finance', \text{null});\]
Insertion

• Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000

  ```sql
  insert into instructor
  select ID, name, dept_name, 18000
  from student
  where dept_name = 'Music' and total_cred > 144;
  ```

• The `select from where` statement is evaluated fully before any of its results are inserted into the relation

  Otherwise queries like

  ```sql
  insert into table1 select * from table1
  ```

  would cause problem
Updates

• Give a 5% salary raise to all instructors
  \[ \text{update instructor} \]
  \[ \text{set salary} = \text{salary} * 1.05 \]

• Give a 5% salary raise to those instructors who earn less than 70000
  \[ \text{update instructor} \]
  \[ \text{set salary} = \text{salary} * 1.05 \]
  \[ \text{where salary} < 70000; \]

• Give a 5% salary raise to instructors whose salary is less than average
  \[ \text{update instructor} \]
  \[ \text{set salary} = \text{salary} * 1.05 \]
  \[ \text{where salary} < (\text{select avg (salary) from instructor}); \]
Updates

• Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  • Write two **update** statements:
    ```sql
    update instructor
    set salary = salary * 1.03
    where salary > 100000;
    
    update instructor
    set salary = salary * 1.05
    where salary <= 100000;
    ```

• The order is important
Case Statement for Conditional Updates

• Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%

```
update instructor
    set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
    end
```
Updates with Scalar Subqueries

• Recompute and update tot_creds value for all students

\[
\text{update student } S \\
\text{set tot_cred = (select sum(credits)} \\
\text{from takes, course} \\
\text{where takes.course_id = course.course_id and} \\
S.ID= \text{takes.ID and takes.grade <> 'F' and} \\
\text{takes.grade is not null);}
\]

• Sets tot_creds to null for students who have not taken any course

• Instead of sum(credits), use:

\[
\text{case} \\
\text{when sum(credits) is not null then sum(credits)} \\
\text{else 0} \\
\text{end}
\]
Summary

• Overview of the SQL Query Language
• SQL Data Definition
• Basic Query Structure of SQL Queries
• Additional Basic Operations
• Set Operations
• Null Values
• Aggregate Functions
• Nested Subqueries
• Modification of the Database