Welcome!

Goal for the Day

- Basic Schema Definition of SQL
- Basic Structure of SQL Queries
- Aggregation functions
- Nested Subqueries
- Modification of Database

Motivation

- Last week, we discussed the mathematical and analytical background that built DBMS
- We also discussed how to organize tables to make querying effective
- Now, we get into the actually querying
- That was the prequel to SQL

Overview of SQL

- Originated by IBM as Sequel
- Has since been standardized
- There are several parts to SQL

Overview of SQL

- Data-definition language (DDL): SQL provides commands for defining relation schemas, deleting relations, and modifying relation schemas
- Data-manipulation language (DML) the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database
- Integrity: SQL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate the constraints are not allowed

Overview of SQL

- View definition: The SQL DDL includes commands for defining views
- Transaction control : SQL includes commands for specifying the beginning and ending of transactions
- Embedded and Dynamic SQL Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java
- Authorization: SQL DDL includes commands for specifying access rights to relations and views

SQL Data Definition

- The set of relations in a database must be specified to the system by means of a DDL
- The SQL DDL allows specifications of relations and also information about each relation including
  - schema, types of values, integrity constraints, indices to be maintained, security and authorization, physical storage structure

Basic Types

SQL supports a variety of built-in types, including

- char(n) fixed-length character string with user-specified length n
- varchar(n) a variable-length character string with user-specified maximum length n (recommended)
- An integer (finite subset)
- smallint small integer
- numeric(p, d) fixed point number with p digits and d digits allowed to the right of a decimal
- real, double precision: floating point and double precision floating-point numbers with machine-dependent precision
- float(n) a floating-point number, with precision of at least n digits
- Each type can be NULL

Basic Schema Definition

We define a SQL relation by using the `create table` command.

Basic Schema Definition

`create table` r
  (A_1, D_1,
  ... ...,
  A_n D_n,
  integrity-constraint$_1$
  ...
  integrity-constraint$_k$);

Basic Schema Definition

Many integrity constraints are allowed

- primary key $(A_{j1}, A_{j2}, ...A_{jm})$
- primary keys must be non-null and unique
- optional, but good idea to keep track of them

Basic Schema Definition

Many integrity constraints are allowed

- foreign key $(A_{k1}, A_{k2}, ...A_{kl})$
- foreign keys correspond to the primary keys of another relation
- Without this constraint, you may end up with tuples that don't relate back to anything

Basic Schema Definition

Many integrity constraints are allowed

- not null
- forces a value for each member of the attribute

Basic Schema Definition

```
create table department
  (dept_name      varchar(20),
  building varchar(15),
  budget numeric(12,2),
  primary key(dept_name));
```

Basic Schema Definition

```
create table r
  (A1     D1,
  A2     D2,
  A3     D3,
  A4    (D4));
```

Basic Schema Definition

- A newly created table will be empty
- We use the insert command to to add data to it
- `insert into` *instructor*
  `values`(10211, ‘Smith’, ‘Biology’, 66000);
- We can delete from a relation (`delete from`) or drop the relation all together (`drop table`)
- We can add relations with `alter table` like `alter table` *r* `add` A, D

Basic Structure of Queries

- Three main clauses: `select`, `from`, and `where`
- Bad accident of history: select in relational algebra is the same as WHERE in SQL!
- `select` in SQL is like Π in relational algebra

Basic Structure of Queries

Table 1: Teacher

Name	**Teacher ID**	**Salary**
John Smith	101	$50,000$
Jane Doe	102	$55,000$
Mark Johnson	103	$48,000$

Basic Structure of Queries

`select` Name
`from` teacher

Table 2: Teacher

Name
John Smith
Jane Doe
Mark Johnson

Or

`select distinct` Name
`from` teacher

Or

`select all` Name
`from` teacher

all is the default so we don't need to say it

Basic Structure of Queries

We can include aritmetic operations in a select statement as follows:

`select` Name, Teacher ID, Salary*1.1
from teacher

Name	**Teacher ID**	**Salary**
John Smith	101	55, 000
Jane Doe	102	60, 500
Mark Johnson	103	52, 800

Basic Structure of Queries

A where clause is just like σ

`select` Name, Teacher ID, Salary
from teacher
`where` Salary > 50000

Name	**Teacher ID**	**Salary**
Jane Doe	102	$55,000$

SQL allows the use of logical connectives and, or and not in a where clause.
Also, $<, <=, >, >=, =, <>$

Basic Structure of Queries

How to think about it

- select: lists the attributes desired in the result of the query
- from: lists the relations to be accessed to evaluate the query
- where: a predicate involving attributes of the relation in the from clause
- Generally easiest to understand a query by reading from first, then where, and then select

Queries on Multiple Relations

We can generalize a generic query as follows
`select` A_1, A_2,...,A_n `from` r_1, r_2,...,r_m `where` P;

Each A_i is an attribute, each r_i a relation, and P is a predicate with default TRUE.

Queries on Multiple Relations

- Consider the schema
- Department(dept_name, building, budget)
- Instructor(ID, name, dept_name, salary)
- What if we wanted to know the names of all instructors, their department name, and the building of their department?
- We need to combine relations
- We need to make sure each tuple in the instructor relation is matched with the tuple in the department relation whose dept_name value matches the dept_name in the instructor relation

Queries on Multiple Relations

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name=department.dept_name
```

Queries on Multiple Relations

- `from` defines a Cartesian product of the relations listed in the clause
- Recall our relational algebra example with Pets, or our lossless join dependency example with wine to think about what Cartesian products look like
- Instructor $\times$ Department $=$
- (Department.dept_name, Department.building, Department.budget, Instructor.ID, Instructor.name, Instructor.dept_name, Instructor.salary)
- (Department.dept_name, building, budget, ID, name, Instructor.dept_name, salary)

Quick Question

Consider the query

```
    select p.a1
    from p, r1, r2
    where p.a1=r1.a1 or p.a1=r2.a1
```

Under what conditions does the query select values of $p.a1$ that are either in $r1$ or in $r2$. Hint: consider the case where either r_i is empty.

Quick Answer

- The query selects values of $p.a1$ that are equal to some value of $r1a1$ or $r2a2$ if and only if both are non-empty.
- Why? Recall $A \times \emptyset = \emptyset$

Queries on Multiple Relations

- `where` restricts our attention to combinations that meet a certain predicate
- In our example, we only care about combinations of Instructor and Department where the instructor was actually within the department
- Hence the argument
- `where` instructor.dept_name=department.dept_name

Queries on Multiple Relations

- Just like in relational algebra, we may not want to have to specify a where clause each time we want to combine relations because we are concerned about anomalies from the Cartesian product
- SQL supports the natural join
- Instead of
  `select` name
  `from` instructor, department
  `where` instructor.dept_name=department.dept_name
- `select`
  `from` instructor `natural join` department

Queries on Multiple Relations

Consider the schema

Instructor(ID, name, dept_name, salary)

Teaches(ID, course_id, sec_id)

Course(course_id, title, dept_name, credits)

Now consider the following queries

-
  ```
  select name
  from instructor natural join teaches, course
  where teaches.course_id=course.course_id
  ```

-
  ```
  select name
  from instructor natural join teachesnatural join course
  ```

Do they produce the same result?

Queries on Multiple Relations

No

- The natural join of instructor and teaches will contain the attributes (ID, name, dept_name, salary, course_id, sec_id)
- Course shares the attributes dept_name and course_id
- The natural join will force the Course.dept_name and Course.course_id to match the natural join of instructor and teaches
- The Cartesian product will not
- The second query will omit names of instructors who teach courses outside of their department
- We may not want these omitted!

Queries on Multiple Relations

- We can specify exactly which columns should be equated to write better join queries
- `select` name
  `from` (instructor `natural join` teaches) `join` course `using` (course_id)
- `join...using` requires a list of attribute names to be specified
- `join`..using will allow teaches.dept_name and course.dept_name to be different

Queries on Multiple Relations

- Consider any two relations r_1 and r_2 with attributes A_i $i \in \{1, 2, 3\}$ so A_1, A_2, A_3 and $r_1(A_1, A_2, A_3)$ and $r_2(A_1, A_2, A_3)$
- r_1 `join` r_2 `using` $(A_1,\ A_2)$ will require a pair of tuples t_1 from r_1 and t_2 from r_2 to match if $t_1.A_1 = t_2.A_2 \wedge t_1.A_2 = t_2.A_2$.
- The natural join would require a third equality $t_1.A_3 = t_2.A_3$

Other Basic Operations

- Rename
- String
- Attribute specification select
- Ordering
- Where clause predicates

Rename

We may need to change names for many reasons

- Two relations in the from clause may have the same name, producing duplicates
- If we use an arithmetic operation in our select clause, we won't have a name, and need to assign one
- We may want to have an easier to read name
- old-name as new-name

Rename

- `select` name as instructor_name, course_id `from` instructor, teaches `where` instructor.ID=teaches.ID
- `select` T.name, S.course_id
  `from` instructor as T, teaches as S
  `where` T.ID = S.ID
- `select distinct` T.name
  instructor as T, instructor as S
  `where` T.salary > S.salary `and` S.dept_name = 'Biology'
- Common names for what we did with T and S are correlation name, table alias, tuple variable

String Operations

- Strings are enclosed in single quotes ‘Computer’
- Standard for equality is case sensitivity
- `upper` and `s` change the cases
- `trim` remove space
- _ The character matches any char
- % the char matches any substring

String Operations

- 'Intro%' matches any string beginning with “Intro”
- %Comp% matches any string containing “Comp” as a substring
- ‘___’ any string of three chars
- ‘___%’ any string of at least three chars

String Operations

Find the names of all departments whose building names includes the substring 'Watson'

```
select dept_name
from department
where building like '%Watson%'
```

String Operations

- 'Intro%' matches any string beginning with “Intro”
- %Comp% matches any string containing “Comp” as a substring
- ‘___’ any string of three chars
- ‘___%’ any string of at least three chars

String Operations

- Escape can be used to deal with special chars
- `like` 'ab \%cd% `escape` '\' matches all strings with "ab%cd"

Attribute Specification

- What if we want all attributes from a relation?
- We argue `select*`

Order Display

- `order by` lets us list items in ascending order
- Specify `desc` for descending order
- We can combine them
- `select`*
  `from` instructor
  `order by`
  salary desc, name `asc`

Where Clause Predicates

- Say we want people with salaries between 35,000 and 42,000
- SQL supports a between comparison to make this easier
- `select` name
  `from` instructor
  `where` salary `between` 35000 `and` 42000
- [select] name
  `from` instructor
  `where` salary <= 42000 `and` salary >= 35000
- we could also say `not between`

Where Clause Predicates

- Say we want people with salaries between 35,000 and 42,000
- SQL supports a between comparison to make this easier
- `select` name
  `from` instructor
  `where` salary `between` 35000 `and` 42000
- `select` name
  `from` instructor
  `where` salary <= 42000 `and` salary >= 35000

Where Clause Predicates

- We can compare sets of values $(v_1, v_2, ..., v_n)$
- Ordering is defined lexiographically
- $(a_1, a_2) <= (b_1, b_2)$ is true when $a_1 <= b_1$ and $a_2 <= b_2$
- `select` name, course_id
  `from` instructors, teaches
  `where` (instructor.ID, dept_name) = (teaches.ID, 'Biology')
- Result: instructor names and the courses they taught for all instructors in the Bio department who have taught some course

Set Operators

For all set operators, duplicates are automatically deleted, so we must declare `all` if we don't want this to happen

- Union operator
- Intersect operator
- Except operator

Union operator

Remember: duplicates automatically deleted. If we don't want this behavior, add 'all' after the set clause

```
(select course_id
from section
where semester = ‘Fall’ and year = 2009)
union
(select course_id
from section
where semester = ‘Spring’ and year = 2010)
```

Intersect operator

Remember: duplicates automatically deleted. If we don't want this behavior, add 'all' after the set clause

```
(select course_id
from section
where semester = ‘Fall’ and year = 2009)
intersect
(select course_id
from section
where semester = ‘Spring’ and year = 2010)
```

Except operator

Remember: duplicates automatically deleted. If we don't want this behavior, add 'all' after the set clause

```
(select course_id
from section
where semester = ‘Fall’ and year = 2009)
except
(select course_id
from section
where semester = ‘Spring’ and year = 2010)
```

Aggregate Functions

SQL offers five built in aggregating functions

- Average: `avg`
- Minimum: `min`
- Maximum: `max`
- Total: `sum`
- Count: `count`

`sum` and `avg` only accept numbers as inputs, but others can use strings or nonnumeric data

Aggregate Functions

- select avg (salary)
  from instructor
  where dept_name = ‘Comp.Sci’
- select avg (salary) as avg_salary
  from instructor
  where dept_name = ‘Comp.Sci’

Aggregate Functions

- The default is to include duplicates (think about calculating averages)
- We can override this with `distinct`, which sometimes we must do for our result to be meaningful
- `select count` (`distinct` ID)
  `from` teaches
  `where` semester = 'Spring' `and` year = 2010
- We can use `count(*)` to count all tuples in a relation

Aggregate Functions

Sometimes we would like to apply aggregate functions to a group of sets of tuples

- we use the `group by` clause
- Anything after `group by` is put into a group
- "Find the average salary in each department"
- `select` dept_name, `avg`(salary) `as` salary
  `from` instructor
  `group by` dept_name
- You can think of `group by` as always being there, but the default being that the tuple is a group

Aggregate Functions

Instructor(ID, name, dept_name, salary)
Teaches(ID, course_id, sec_id), semester, year)
Course(course_id, title, dept_name, credits)

- Find the number of instructors in each department who teach a course in the Spring 2010 semester
- What do we need to do?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Start with our `from` statement
- Do we have enough information from one relation alone?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- No, we need instructor and teaches
- How do we want to combine them?
- `from` instructor `natural join` teaches

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Think about our where statement
- Will we have to broad of a relation?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Think about our where statement
- Will we have to broad of a relation?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Yes, let's make it smaller
- `where` semester = 'Spring' `and` year = 2010

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Think about grouping
- Do we treat this as just one group or do we want something more specific?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- More specific
- `group by` dept_name

Aggregate Functions

So far we have `from` instructor `natural join` teaches
`where` semester = ‘Spring’ `and` year = 2010
`group by` dept_name
What’s missing?

Aggregate Functions

Find the number of instructors in each department who teach a course in the Spring 2010 semester

- Let's apply an aggregation function and have it print out without an arbitrary name
- What will we say?

Aggregate Functions

- `select` dept_name, `count distinct`(ID) as instructor_count

Aggregate Functions

```
select dept_name, count distinct(ID) as instructor_count
from instructor natural join teaches
where semester = ‘Spring’ and year = 2010
group by dept_name
```

Aggregate Functions

- Important note: when aggregating, the select statement should only include the grouping attribute and the aggregated attribute (the one having the function performed on it)
- Otherwise, SQL will treat the query as an error
- Example: `select` dept_name, ID, `avg`(salary)
  `from` instructor
  `group by` dept_name
- What is wrong?

Aggregate Functions

Another great clause for aggregation, `having`

- What if we wanted to make a statement that applies to entire groups?
- Ex: We only care about departments within a particular field (STEM or Humanities) or with a particular budget size
- We use `having`, which will apply a predicate after groups are formed

Example

- What is the average salary among departments where the average salary is greater than 42,000?

-
  ```
  select dept_name, avg(salary) as avg_salary
  from instructor
  group by dept_name
  having avg(salary) > 42000
  ```
- Having clause can only include attributes being grouped or aggregated, or the query is erroneous

Example

Consider the schema

- Takes(ID, course_id, sec_id, semester, year, grade)
- Course(course_id, title, dept_name, credits)

For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

Example

Question: For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

Questions we ask ourselves to write the query?

- Does any one relation give us the answer?
- Do we need to reduce the number of tuples we are looking at?

Example

Question: For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

- No. `from` takes `natural join` student
- Yes. `where` year = 2009

Example

Question: For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

- How are we grouping?
- What are our group conditions?

Example

Question: For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

- `group by` course_id
- `having count`(ID) $>= 2$

Example

Question: For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least two students

- What are we trying to print?
- `select` course_id, `avg`(total_cred) `as` avg_cred

Example

Our final query:

```
select course_id, avg(total_cred) as avg_cred
from takes natural join student
where year = 2009
group by course_id, seemster, year, sec_id
having count(ID)>=2
```

Aggregation with Null Values

- The SQL standard says the function ignores null values
- All aggregation functions except *count*$(*)$ ignore null values
- Count will include the null value because it will want to add up the total number of tuples in a group

Nested Subqueries

We can nest a query inside a larger query. By nesting, we can combing questions to increase our efficiency. We can use nested subqueries for several things

- Set Membership
- Set Comparison
- Testing for Empty Relations
- Testing for Duplicates
- Augmenting `from` and `with` clauses

Nested Subqueries

- A subquery is a `select-from-where` expression nested in another query.
  The nesting can be done in the following SQL query:

  ```
  select  A1, A2, ..., An
  from  r1, r2, ..., rm
  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:

    $$B < operation > \text{(subquery)}$$

    where B is an attribute and <operation> is to be defined later.
  - **Select clause:** A_i can be replaced by a subquery that generates a single value.

Nested Subqueries

- Say we want to know what courses were both taught in the fall and the spring in years 2017 and 2018 respectively
- One way of doing this is with set intersection
- Another way is to use a nested subquery

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- Step 1: Write our nested query
- Step 2: Write the outer query

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- Here, we want courses that appear in both the fall and the spring
- We can obtain list 1, and say in a larger query we want a list that includes list 1 as a condition
- In this instance, it would be generating the list of courses in the spring 2018, and then saying we want the list of courses in the fall in 2017

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- What is our first query?

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- What is our first query?
- (`select` course_id
  `from` section
  `where` semester = 'Spring' and year = 2018)

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- What query goes on the outside?

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- What query goes on the outside?
- select distinct course_id
  from section
  where semester = 'Fall' and year = 2017

Example: What courses were both taught in the fall and the spring in years 2017 and 2018

- Let's connect them with `and` course_id `in`

Example: What courses were both taught in the fall and the spring in years 2017 and 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);
```

Example 2: Name all instructors whose name is neither “Mozart” nor “Einstein”

- We can also use `in` and `not in` to test set membership

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

```
select count(distinct ID)
from takes
where (course_id, sec_id, semester, year) in
  (select course_id, sec_id, semester, year
  from teaches
  where teaches.ID = 10101);
```

Set Comparison

Earlier, we wanted to know the names of all instructors whose salary is greater than at least one instructor in the Biology department. We wrote the following code to do so:

```
select distinct T.name
from instructor as T, instructor as S
where T.Salary > S.salary and S.dept_name = 'Biology';
```

- Instead, we can use **some**
- Some indicates set membership

Set Comparison

- $(f < comp > t)\texttt{some}r \leftrightarrow \exists t \in r : (f < comp > t)$
- $(f < comp > t) \in \{<, >, \leq, \geq, =, \neq\}$
- You can think about it as at least, there only needs to exist a t

Set Comparison

- $5 < \textit{some}\{0, 5, 6\}$
- $5 < \textit{some}\{0, 5\}$
- $5 = \textit{some}\{0, 5\}$
- $5 \neq \textit{some}\{0, 5\}$

Set Comparison

- $5 < \textit{some}\{0, 5, 6\}$ True
- $5 < \textit{some}\{0, 5\}$ False
- $5 = \textit{some}\{0, 5\}$ True
- $5 \neq \textit{some}\{0, 5\}$ True
- $(= \textit{some}) \equiv \textit{in}$
- $\neq \textit{some} \not\equiv$ not in

Set Comparison

We may also use an all clause instead of the some clause

- $(f < comp > t)\texttt{some}r \overset{\forall}{\leftrightarrow} t \in r : (f < comp > t)$
- The statement must be true for all t, rather just at least 1 t

Set Comparison

- $5 < \textit{all}\{0,5,6\}$ False
- $5 < \textit{all}\{6,10\}$ True
- $5 = \textit{all}\{0,5\}$ False
- $5 \neq \textit{all}\{4,6\}$ True
- $(= \textit{some}) \equiv \textit{in}$
- $\neq \textit{all} \equiv$ not in
- $= \textit{all} \not\equiv$ in

Tests for Empty Relations

- exists returns the value true of the subquery is nonempty

Query: Courses in Both Fall 2017 and Spring 2018

```
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.

Tests for Empty Relations

- We can write “relation A contains relation B” as “B except A”

Tests for Empty Relations

- A scoping rule applies: local definitions apply, and a local definition can only be used locally

Example: Find all students who have taken all courses offered in the Bio department

```
    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
);
```

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

Example: Find all students who have taken all courses offered in the Bio department

- Outer query: all courses
- Inner query 1: the set of all courses offered in Biology
- Inner query 2: the set of all courses student S.ID has taken
- Overall, the query tests whether there exists a student who has taken all courses in the biology department because it attempts to see if the set of courses students has taken contains the set of all biology classes

Example: Find all students who have taken all courses offered in the Bio department

- Why is this the case?
- Let's think back to set theory
- $X - Y = \emptyset \longleftrightarrow X \subseteq Y$
- We would only get nothing if everything in X is also in Y, and that is the definition of subet.

Testing for Duplicate Tuples

We can test for whether a subquery has duplicate tuples in several ways. One way, not widely implemented, is UNIQUE. Consider the question: Find all courses that were offered at most once in 2009

select T.courseid
from course as T
where unique (**select** R.courseid
from section as R
where T.courseid=R.courseid
and R.year = 2017)

Unique formally tests if there are any two tuples t_1 and t_2 s.t. $t_1 = t_2$. Test fails if any fields are null.

Testing for Duplicate Tuples

We can also do it this way:

```
select T.course_id
from course as T
where 1 <= (select count(R.course_id)
    from section as R
    where year = 2017 and T.course_id=R.course_id)
```

Subqueries in the From Clause

- We may not want to use the having clause
- select dept_name, avg_salary
  from(select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name)
  where avg_salary > 42000
- SELECT dept_name, avg_salary
  FROM(SELECT * FROM (
      SELECT dept_name, AVG(salary) as avg_salary
      FROM instructor
      GROUP BY dept_name
) AS dept_avg)
  where avg_salary > 42000

The With Clause

- The with clause creates a temporary relation with definitions that are only available to the queries within the with clause
- Find the department with the maximum budget
- **with** max_budget (value) as
      (**select** max(budget)
      from department)
  select dept_name
  from department, max_budget
  where budget = max_budget.value

The With Clause

- We defined a new relation that was temporary, and that relation selected the max budget value from departments. We named it max_budget
- Then, we selected department names from the relations department and max_budget, only looking at cases where the budget in department as equal to the max we found in our nested query
- Our final result is then given

Finding Departments with Total Salary Above Average

- Let's find all departments where the total salary is greater than the average of the total salary at all departments.

Finding Departments with Total Salary Above Average

- We want to find departments where the total salary is greater than the average of the total salary across all departments.

Step 1: Create ‘dept_total‘ Table

```
dept_total (dept_name, value) as
  (select dept_name, sum(salary)
   from instructor
   group by dept_name)
```

- This creates a table with department names and their total salaries.
- The ‘sum(salary)‘ is calculated for each department using the ‘GROUP BY‘ clause.

Step 2: Create ‘**dept**$_t$*otal*$_a$*vg*‘ *Table*

```
dept_total_avg(value) as
  (select avg(value)
   from dept_total)
```

- This creates a table with a single column containing the average of the ‘value‘ column from the ‘dept_total‘ table.

Step 3: Select Departments Above Average

```
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

- Selects department names where the total salary (‘value‘ in ‘dept_total‘) is greater than the average total salary (‘value‘ in ‘dept_total_avg‘).

Putting it all Together

- We want to find departments where the total salary is greater than the average of the total salary across 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;
```

Scalar Subquery

Definition: A scalar subquery is used where a single value is expected.

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

Scalar Subquery

Definition: A scalar subquery is used where a single value is expected.

- **Example:** List all departments along with the number of instructors in each department.

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

Scalar Subquery

Definition: A scalar subquery is used where a single value is expected.

- **Example:** List all departments along with the number of instructors in each department.

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

- **Note:** Runtime error if the subquery returns more than one result tuple.

Modification of the Database

- Deletion
- Insertion
- Update

Delete

- General Structure

  ```
  delete from r
  where P
  ```

P(t) finds all tuples *t* for which statement *P* is true

We delete whole tuples, not values within attributes

DELETE Statements

1. **Delete All Instructors:**

```
DELETE FROM instructor;
```

2. **Delete Instructors from the Finance Department:**

```
DELETE FROM instructor
WHERE dept_name = 'Finance';
```

3. **Delete Tuples for Instructors in Watson Building:**

```
DELETE FROM instructor
WHERE dept_name IN (SELECT dept_name
                    FROM department
                    WHERE building = 'Watson');
```

DELETE Statements

- Delete all instructors whose salary is less than the average salary of instructors
- We must be careful to compute the average first, because it will change as we delete

```
DELETE FROM instructor
WHERE salary < (SELECT AVG (salary)
                FROM instructor);
```

Adding a Tuple to the Course Table

- We want to add a new tuple to the ‘course‘ table.

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

```
insert into course (course_id, title, dept_name, credits)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

Adding a Tuple to the Course Table

- We want to add a new tuple to the ‘course‘ table.

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Alternatively, we can use the explicit column list:

```
insert into course (course_id, title, dept_name, credits)
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

Adding a Tuple to the Student Table

- Now, let's add a new tuple to the ‘student‘ table with ‘tot_creds‘ set to null.

```
insert into student
values ('3003', 'Green', 'Finance', null);
```

Adding a Tuple to the Student Table

- Now, let's add a new tuple to the ‘student‘ table with ‘tot_creds‘ set to null.

```
insert into student
values ('3003', 'Green', 'Finance', null);
```

- This SQL query inserts a new student with null total credits (‘tot_creds‘).

Inserting Students as Instructors

- We want to 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.

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

Inserting Students as Instructors

- We want to 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.

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

- This SQL query inserts students into the ‘instructor‘ table, assigning a salary of 18,000.

Inserting Students as Instructors

- We want to 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.

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

- This SQL query inserts students into the ‘instructor‘ table, assigning a salary of 18,000.
- The condition ensures that only Music department students with more than 144 credit hours are selected.

Updates

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

Updates

- We can use an update statement if we want to change the values of some tupels without changing all values in the tuple
- Increase the salary by 5%

-
  ```
  update instructor
  set salary = salary*1.05
  ```
- Say we wanted to do something more complicated: give a 3% raise for people earning over 100000 and 5% otherwise

Updates

Using the case construct: general form

```
case
    when pred_1 then result_1
    when pred_2 then result_2
    ...
    else result_0
end
```

In our example:

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