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 n digits to the right of decimal point.
- **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 are covered in Chapter 4.**

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Create Table Construct

An SQL relation is defined using the `create table` command:

```
create table r (A₁, D₁, A₂, D₂, ... , Aᵣ, Dᵣ,
   (integrity-constraint₁),
   ... , (integrity-constraintᵣ))
```

- `r` is the name of the relation
- Each `Aᵢ` is an attribute name in the schema of relation `r`
- `Dᵢ` is the data type of values in the domain of attribute `Aᵢ`

Example:

```
create table instructor (ID char(5), name varchar(20) not null, dept_name varchar(20), salary numeric(9,2))
insert into instructor values ('10211', 'Smith', 'Biology', 66000);
insert into instructor values ('10211', null, 'Biology', 66000);
```
And a Few More Relation Definitions

- **create table** student (id varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  salary numeric(3,5),
  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);

- Note: sec_id can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester.

Basic Query Structure

The SQL data-manipulation language (DML) provides the ability to query information, and insert, delete and update tuples.

- A typical SQL query has the form:
  ```sql
  select A_1, A_2, ..., A_n
  from T_1, T_2, ..., T_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:
  ```sql
  select name from instructor
  ```

- NOTE: SQL keywords are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. Name ≠ NAME's name
  - Some people use upper case wherever we use bold font.
The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword `distinct` after select.
- Find the names of all departments with instructor, and remove duplicates
  
  ```sql
  SELECT DISTINCT dept_name
  FROM instructor
  ```

- The (redundant) keyword `all` specifies that duplicates not be removed.
  
  ```sql
  SELECT all dept_name
  FROM instructor
  ```

- An asterisk in the select clause denotes "all attributes"
  
  ```sql
  SELECT * FROM instructor
  ```

- The select clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.
  - Most systems also support additional functions
    - E.g., substring
  - Most systems allow user defined functions (UDFs)

- The query:
  
  ```sql
  SELECT ID, name, salary/12
  FROM instructor
  ```

  would return a relation that is the same as the instructor relation, except that the value of the attribute salary is divided by 12.

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
  
  ```sql
  SELECT * FROM instructor, teaches
  ```

  generates every possible instructor – teaches pair, with all attributes from both relations

- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

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 by salary > 80000
  
  ```sql
  SELECT name
  FROM instructor
  WHERE dept_name = 'Comp. Sci.' AND salary > 80000
  ```

- Comparison results can be combined using the logical connectives and, or, and not.

- Comparisons can be applied to results of arithmetic expressions.

- SQL standard: any valid expression that returns a boolean result
  - Vendor specific restrictions may apply!

Cartesian Product: instructor X teaches

<table>
<thead>
<tr>
<th>Instructor</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
</tr>
<tr>
<td>1001</td>
<td>Boris Glavic</td>
</tr>
<tr>
<td>1002</td>
<td>John Doe</td>
</tr>
<tr>
<td>1003</td>
<td>Jane Smith</td>
</tr>
<tr>
<td>1004</td>
<td>Peter King</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>CS-101</td>
<td>Comp. Sci</td>
<td>50000</td>
</tr>
<tr>
<td>1020</td>
<td>CS-201</td>
<td>Comp. Sci</td>
<td>60000</td>
</tr>
<tr>
<td>1030</td>
<td>CS-301</td>
<td>Comp. Sci</td>
<td>70000</td>
</tr>
</tbody>
</table>

Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.
  
  ```sql
  SELECT name, course_id
  FROM instructor, teaches
  WHERE instructor.ID = teaches.ID
  ```

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department
  
  ```sql
  SELECT section.course_id, semester, year, title
  FROM section, course
  WHERE course.department = 'Comp. Sci.'
  ```
Try Writing Some Queries in SQL

- Suggest queries to be written.....

Joined Relations

- Join operations take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join.
- The join operations are typically used as subquery expressions in the from clause.

Join operations – Example

- Relation course

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

- Relation prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

- Observe that prereq information is missing for CS-315 and course information is missing for CS-437.

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- This is the natural join from relational algebra.
- select * from instructor natural join teaches;

Natural Join (Cont.)

- Danger in natural join; beware of unrelated attributes with same name which get equated incorrectly.
- List the names of instructors along with the title of courses that they teach.

- Incorrect version (makes course dept name = instructor dept name)
  - select name, title from instructor natural join teaches;

- Correct version
  - select name, title from instructor natural join teaches course;
  - Another correct version
  - select name, title from (instructor natural join teaches) join course using(course_id);
  - select name, title from (instructor natural join teaches) join course using(course_id);

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.

- Incorrect version (makes course dept name = instructor dept name)
  - select name, course_id from instructor, teaches where instructor.ID = teaches.ID;

- Correct version
  - select name, course_id from instructor natural join teaches;
Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses null values.

Left Outer Join

- course natural left outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>GameDesign</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>

Right Outer Join

- course natural right outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>GameDesign</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the from clause.
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<table>
<thead>
<tr>
<th>Join types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner join</td>
<td>natural on &lt;predicate&gt;</td>
</tr>
<tr>
<td>left outer join</td>
<td>using (A₁, A₂, ..., Aₙ)</td>
</tr>
<tr>
<td>right outer join</td>
<td></td>
</tr>
<tr>
<td>full outer join</td>
<td></td>
</tr>
</tbody>
</table>

Full Outer Join

- course natural full outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>GameDesign</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Joined Relations – Examples

- course inner join prereq on course.course_id = prereq.course_id

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
<th>course_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>CS-190</td>
<td>CS-190</td>
</tr>
<tr>
<td>CS-190</td>
<td>GameDesign</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

- What is the difference between the above, and a natural join?
- course left outer join prereq on course.course_id = prereq.course_id

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>null</td>
</tr>
<tr>
<td>CS-190</td>
<td>GameDesign</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>null</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>CS-101</td>
<td>null</td>
</tr>
</tbody>
</table>
Joined Relations – Examples

- course natural right outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

- course full outer join prereq using (course_id)

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3 null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

The Rename Operation

- The SQL allows renaming relations and attributes using the `as` clause: `old-name as new-name`
- E.g.
  - `select ID, name, salary/12 as monthly_salary from instructor`
- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.
  - `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
  - `instructor as T`
  - `Keyword as` must be omitted in Oracle

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:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring ‘dar’.
  - `select name from instructor where name like '%dar%'`
- Match the string ‘100 %’
  - `like '100 % escape %'`

String Operations (Cont.)

- 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.

Case Construct

- Like case, if, and ? Operators in programming languages

```sql
case
  when c1 then e1
  when c2 then e2
  ...
  [else e0]
end
```
- Each c is a condition
- Each e is an expression
- Returns the first e for which c evaluates to true
  - If none of the c is true, then return e0 (else)
  - If there is no else return null

Case Construct Example

- Like case, if, and ? Operators in programming languages

```sql
select
  name,
  case
    when salary > 100000 then 'premium'
    else 'standard'
  end as customer_group
from customer
```
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.
  ```sql
  Example: order by name desc
  ```
- Can sort on multiple attributes
  ```sql
  Example: order by dept_name, name
  ```
- Order is not expressible in the relational model!

Where Clause Predicates

- SQL includes a between comparison operator
  ```sql
  Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)
  ```
- 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 2009 and in Spring 2010
  ```sql
  (select course_id from section where sem = 'Fall' and year = 2009)
  union
  (select course_id from section where sem = 'Spring' and year = 2010)
  ```
- Find courses that ran in Fall 2009 and Spring 2010
  ```sql
  (select course_id from section where sem = 'Fall' and year = 2009)
  intersect
  (select course_id from section where sem = 'Spring' and year = 2010)
  ```
- Find courses that ran in Fall 2009 but not in Spring 2010
  ```sql
  (select course_id from section where sem = 'Fall' and year = 2009)
  except
  (select course_id from section where sem = 'Spring' and year = 2010)
  ```

Set Operations

- Set operations union, intersect, and except
  ```sql
  Each of the above operations automatically eliminates duplicates
  ```
  ```sql
  To retain all duplicates use the corresponding multiset versions
  union all, intersect all and except all.
  ```
- Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  ```sql
  m + n times in r union all s
  min(m,n) times in r intersect all s
  max(0, m - n) times in r except all s
  ```

Null Values

- It is possible for tuples to have a null value, denoted by null, for some of their attributes
  ```sql
  null signifies an unknown value or that a value does not exist.
  ```
- The result of any arithmetic expression and comparisons involving null evaluates to null
  ```sql
  Example: 5 + null returns null
  null > 5 returns null
  null = null returns null
  ```
- The predicate is null can be used to check for null values.
  ```sql
  Example: Find all instructors whose salary is null.
  ```
  ```sql
  select name
  from instructor
  where salary is null
  ```

Null Values and Three Valued Logic

- Any comparison with null returns null
  ```sql
  Example: 5 < null or null <= null or null = null
  ```
- Three-valued logic using the truth value null:
  ```sql
  OR: (null or true) = true, (null or false) = null (null or null) = null
  ```
  ```sql
  AND: (true and null) = null, (false and null) = false, (null and null) = null
  ```
  ```sql
  NOT: (not null) = null
  ```
  ```sql
  "P is null" evaluates to true if predicate P evaluates to null.
  ```
  ```sql
  Result of where clause predicate is treated as false if it evaluates to null.
  ```
Aggregation 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
- Most DBMS support user defined aggregation functions.

Aggregation Functions (Cont.)

- 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 2010 semester:
  ```sql
  select count(distinct ID)
  from teaches
  where semester = 'Spring' and year = 2010
  ```
- Find the number of tuples in the course relation:
  ```sql
  select count(*)
  from course;
  ```

Aggregation Functions – Group By

- Find the average salary of instructors in each department:
  ```sql
  select dept_name, avg(salary)
  from instructor
  group by dept_name;
  ```
- Where: departments with no instructor will not appear in result.

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>Dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>7676</td>
<td>Cric</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>4036</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>79000</td>
</tr>
<tr>
<td>1010</td>
<td>Srivikasam</td>
<td>Comp. Sci.</td>
<td>60000</td>
</tr>
<tr>
<td>1038</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>9341</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>1212</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>3041</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>3234</td>
<td>El Sol</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>5855</td>
<td>Californi</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>3333</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>3436</td>
<td>Gobli</td>
<td>Physica</td>
<td>87000</td>
</tr>
<tr>
<td>2222</td>
<td>Einstein</td>
<td>Physics</td>
<td>90000</td>
</tr>
</tbody>
</table>

Aggregation 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)
  from instructor
  group by dept_name
  having avg(salary) > 42000;
  ```
- Note: predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups.

Aggregation (Cont.)

- Attributes in select clause outside of aggregate functions must appear in group by list:
  ```sql
  /* erroneous query */
  select dept_name, ID, avg(salary)
  from instructor
  group by dept_name;
  ```
- Predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups.

Null Values and Aggregates

- Total all salaries:
  ```sql
  select sum(salary)
  from instructor;
  ```
- Above statement ignores null amounts
- Result is null if there is no non-null amount
- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null
Empty Relations and Aggregates

- What if the input relation is empty
- Conventions:
  - \texttt{sum}: returns null
  - \texttt{avg}: returns null
  - \texttt{min}: returns null
  - \texttt{max}: returns null
  - \texttt{count}: returns 0

Multiset Relational Algebra

- Pure relational algebra operates on set-semantics (no duplicates allowed)
  - E.g., after projection
- Multiset (bag-semantics) relational algebra retains duplicates, to match SQL semantics
  - SQL duplicate retention was initially for efficiency, but is now a feature
- Multiset relational algebra defined as follows
  - \texttt{selection}: has as many duplicates of a tuple as in the input, if the tuple satisfies the selection
  - \texttt{projection}: one tuple per input tuple, even if it is a duplicate
  - \texttt{cross product}: If there are \( m \) copies of \( t_1 \) in \( r \), and \( n \) copies of \( t_2 \) in \( s \), there are \( m \times n \) copies of \( t_1 \times t_2 \) in \( r \times s \)
  - Other operators similarly defined
    - E.g., union: \( m + n \) copies
    - \texttt{intersection}: \( \min(m, n) \) copies
    - \texttt{difference}: \( \max(0, m-n) \) copies

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset (bag semantics)** versions of some of the relational algebra operators – given multiset relations \( r_1 \) and \( r_2 \):
  1. \( \sigma_P(r_1) \): If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \), and \( t_1 \) satisfies selection \( \sigma_P(r_1) \), then there are \( c_1 \) copies of \( t_1 \) in \( \sigma_P(r_1) \).
  2. \( \Pi_A(r) \): For each copy of tuple \( t_1 \) in \( r_1 \), there is a copy of \( \Pi_A(t_1) \) in \( \Pi_A(r_1) \) where \( \Pi_A(t_1) \) denotes the projection of the single tuple \( t_1 \).
  3. \( r_1 \times r_2 \): If there are \( c_1 \) copies of tuple \( t_1 \) in \( r_1 \), and \( c_2 \) copies of tuple \( t_2 \) in \( r_2 \), there are \( c_1 \times c_2 \) copies of the tuple \( t_1 \times t_2 \) in \( r_1 \times r_2 \).

Empty Relations and Aggregates (Cont.)

- Example: Suppose multiset relations \( r_1(A, B) \) and \( r_2(C) \) are as follows:
  \( r_1 = \{(1, 3), (2, 3), (3, 3)\} \)
  \( r_2 = \{(2, 3), (3, 3)\} \)
- Then \( \Pi_B(r_1) \) would be \( \{(a, b), (a, b)\} \), while \( \Pi_B(r_1) \times \Pi_B(r_2) \) would be
  \( \{(a, b, c), (a, b, c), (a, b, c), (a, b, c), (a, b, c)\} \)
- SQL duplicate semantics:
  \( \text{select } A_1, A_2, \ldots, A_n \)
  \( \text{from } r_1, r_2, \ldots, r_m \)
  \( \text{where } P \)
  - is equivalent to the multiset version of the expression:
  \( \Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m)) \)

SQL and Relational Algebra

- \( \text{select } A_1, A_2, \ldots, A_n \)
  \( \text{from } r_1, r_2, \ldots, r_m \)
  \( \text{where } P \)
  \( \text{is equivalent to the following expression in multiset relational algebra} \)
  \[ \prod_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m)) \]

Duplicates (Cont.)

- More generally, the non-aggregated attributes in the \texttt{select} clause may be a subset of the \texttt{group by} attributes, in which case the equivalence is as follows:
  \( \text{select } A_1, \text{sum}(A_2) \) \text{AS} \text{sumA3} \)
  \( \text{from } r_1, r_2, \ldots, r_m \)
  \( \text{where } P \)
  \( \text{group by } A_1, A_2 \)
  \( \text{is equivalent to the following expression in multiset relational algebra} \)
  \[ \prod_{A_1, \text{sumA3}}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m)) \]
Subqueries in the From 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)
where avg_salary > 42000;
```

- Note that we do not need to use the having clause.
- Another way to write above query

```sql
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
where avg_dept(avg (dept_name, avg_salary)) > 42000;
```

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.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find courses offered in Fall 2009 and in Spring 2010

```sql
select distinct course_id
from section
where semester = 'Fall' and year=2009 and
      course_id in (select course_id
                     from section
                     where semester = 'Spring' and year=2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```sql
select distinct course_id
from section
where semester = 'Fall' and year=2009 and
      course_id not in (select course_id
                         from section
                         where semester = 'Spring' and year=2010);
```

Example Query

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

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

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

Quantification

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

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

- Same query using > some clause

```sql
select name
from instructor
where salary > some (select salary
                    from instructor
                    where dept_name = 'Biology');
```

Definition of Some Clause

- F <comp> r ³ t ² r such that (F <comp> r)
Where <comp> can be: <, ≤, >, =

(F < some 0 5 6)
(read: 5 < some tuple in the relation)

(F < some 0 5 6) = true
(F ≥ some 0 5 6) = false
(F = some 0 5 6) = true
(F ≠ some 0 5 6) = true (since 0 ≠ 5)
(≥ some) = in
However, (= some) / not in
Example Query

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

```
select name
from instructor
where salary > all (select salary
  from instructor
  where dept_name = 'Biology');
```

Definition of all Clause

- \( F < \text{comp} > \) all \( r \) if \( \forall t \in r (F < \text{comp} > t) \)

- \( (5 < \text{all} ) = \text{false} \)
- \( (5 < \text{all} ) = \text{true} \)
- \( (5 = \text{all} ) = \text{false} \)
- \( (5 < \text{all} ) = \text{true} \) (since 5 \( \neq \) 4 and 5 \( \neq \) 6)
- \( (+ \text{all} ) \) is not in
- However, \( (\text{all} ) \) \( \notin \) in

Test for Empty Relations

- The \( \exists \) construct returns the value \text{true} if the argument subquery returns a nonempty result.
- \( \exists r \iff r = \emptyset \)
- \( \nexists r \iff r = \emptyset \)

Correlation Variables

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester":

```
select course_id
from section as S
where semester = 'Fall' and year= 2009
and exists (select *
  from section as T
  where semester = 'Spring' and year= 2010
  and S.course_id = T.course_id);
```

- Correlated subquery
- Correlation name or correlation variable

Not Exists

- Find all students who have taken all courses offered in the Biology 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 course_id
    from takes as T
    where S.ID = T.ID)));
```

- Note that \( X - Y = \emptyset \iff X \subseteq Y \)
- Note: Cannot write this query using \( \text{all} \) and its variants

Test for Absence of Duplicate Tuples

- The \( \text{unique} \) construct tests whether a subquery has any duplicate tuples in its result.
- \( (\text{Evaluates to "true" on an empty set}) \)
- Find all courses that were offered at most once in 2009

```
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 = 2009);
```
With Clause

- The with clause provides a way of defining a temporary view 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 budget
  from department, max_budget
  where department.budget = max_budget.value;
  ```

Complex Queries using With Clause

- With clause is very useful for writing complex queries
- Supported by most database systems, with minor syntax variations
- Find all departments where the total salary is greater than the average of the total salary at all departments
  
  ```sql
  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

- Scalar subquery is one which is used where a single value is expected
  
  ```sql
  E.g. select dept_name, (select count(*)
  from instructor
  where department.dept_name = instructor.dept_name)
  as num_instructors
  from department;
  ```

- E.g. select name
  from instructor
  where salary * 10 > (select budget 
  from department
  where department.dept_name = instructor.dept_name)

- Runtime error if subquery returns more than one result tuple

Query Features Recap - Syntax

- Almost all clauses are optional
- Examples:
  - SELECT * FROM r;
  - SELECT 1;
  - Conventional: returns single tuple
  - SELECT * FROM accounts HAVING sum(balance) = 0;
  - SELECT a.BY GROUP BY 1;
  - SELECT HAVING true;
  - Let r be a relation with two attributes a and b
    - SELECT a.b FROM r
    - WHERE a in (SELECT a FROM r) AND b IN (SELECT b FROM r)
    - GROUP BY a.b HAVING count(*) > 0;
  - Note: Not all systems support all of this “non-sense”

- An SQL query is either a Select-from-where block or a set operation
- An SQL query block is structured like this:
  - `SELECT [DISTINCT] select_list`
  - `[FROM from_list]`
  - `[WHERE where_condition]`
  - `[GROUP BY group_by_list]`
  - `[HAVING having_condition]`
  - `[ORDER BY order_by_list]`

- Set operations
  - `| Query Block | set_op | | Query Block`
  - `set_op`: `ALL | UNION | INTERSECT | EXCEPT`
**Syntax – ORDER BY**

- ORDER BY order_by_list
- order_by_list
  - Like select_list minus renaming
  - Optional [ASC | DESC] for each item

Examples:
- SELECT * FROM r ORDER BY a;
- SELECT * FROM r ORDER BY b, a;
- SELECT * FROM r ORDER BY a * 2;
- SELECT * FROM r ORDER BY a * 2, a;
- SELECT * FROM r ORDER BY a + (SELECT count(*) FROM s);

**Query Semantics**

- Evaluation Algorithm (you can do it manually – sort of)
  1. Compute FROM clause
     - Compute cross product of all items in the FROM clause
     - Relations: nothing to do
     - Subqueries: use this algorithm to recursively compute the result of subqueries first
     - Join expressions: compute the join
  2. Compute WHERE clause
     - For each tuple in the result of 1. evaluate the WHERE clause condition
  3. Compute GROUP BY clause
     - Group the results of step 2. on the GROUP BY expressions
  4. Compute HAVING clause
     - For each group (if any) evaluate the HAVING condition

**Query Semantics (Cont.)**

5. Compute SELECT clause
   - Project each result tuple from step 4 on the SELECT expressions
6. Compute ORDER BY clause
   - Order the result of step 5 on the ORDER BY expressions

If the WHERE, SELECT, GROUP BY, HAVING, ORDER BY clauses have any nested subqueries:
- For each tuple t in the result of the FROM clause
- Substitute the correlated attributes with values from t
- Evaluate the resulting query
- Use the result to evaluate the expression in the clause the subquery occurs in

**Query Semantics (Cont.)**

- Equivalent relational algebra expression
  - ORDER BY has no equivalent, because relations are unordered
  - Nested subqueries: need to extend algebra (not covered here)
- Each query block is equivalent to
  \[
  \pi(G(\pi(F_1 \times \ldots F_n)))
  \]
- Where \( F_i \) is the translation of the \( i \) FROM clause item
- Note: we leave out the arguments

**Modification of the Database**

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation
Modification of the Database – Deletion

- Delete all instructors
  
  ```
  delete from instructor
  ```

- Delete all instructors from the Finance department
  
  ```
  delete from instructor
  where dept_name = 'Finance';
  ```

- Delete all tuples in the `instructor` relation for those instructors associated with a department located in the Watson building.
  
  ```
  delete from instructor
  where dept_name in (select dept_name from department
  where building = 'Watson');
  ```

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors
  
  ```
  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 `avg` salary and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing `avg` or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to `course`
  
  ```
  insert into course
  values ('CS-437', 'Database Systems', 'comp. Sci.', 4);
  ```

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

Insertion (Cont.)

- Add all instructors to the `student` relation with `tot_creds` set to 0
  
  ```
  insert into student
  select ID, name, dept_name, 0
  from instructor
  ```

  - The `select from where` statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like `insert into table1 select * from table1` would cause problems, if `table1` did not have any primary key defined.

Modification of the Database – Updates

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

    ```
    update instructor
    set salary = salary * 1.05
    where salary <= 100000;
    ```

  - The order is important
  - Can be done better using the `case` statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement
  
  ```
  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
  ```sql
  update student S
  set tot_creds = (select sum(credits) from takes natural join course
  where S.ID = takes.ID and takes.grade <> 'F' and takes.grade is not null);
  ```
- Sets tot_creds to null for students who have not taken any course
- Instead of sum(credits), use:
  ```sql
  case
    when sum(credits) is not null then sum(credits)
  else 0
  end
  ```
- Or COALESCE(sum(credits),0)

Recap

- SQL queries
  - Clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY
  - Nested subqueries
  - Equivalence with relational algebra
  - SQL update, inserts, deletes
  - Semantics of referencing updated relation in WHERE
  - SQL DDL
    - Table definition: CREATE TABLE

Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL - Intermediate
- Database Design
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization

Advanced SQL Features**

- Create a table with the same schema as an existing table:
  ```sql
  create table temp_account like account
  ```