Extended Projection

- Allow the columns in the projection to be functions of one or more columns in the argument relation.

- Example

\[ R = \begin{array}{cc}
A & B \\
1 & 2 \\
3 & 4 \\
\end{array} \]

\[ \pi_{A+B,A,A}(R) = \begin{array}{ccc}
A+B & A1 & A2 \\
3 & 1 & 1 \\
7 & 3 & 3 \\
\end{array} \]
Aggregation Operators

- Summarize a column in some way.
  - Operate over multiple tuples
- Five standard operators: Sum, Average, Count, Min, and Max.
  - Use with grouping (see next slide) or shorthand as “special” projection:

\[
R = \pi_{\text{Max}(A), \text{Min}(B)}(R) = \text{Max}(A) \quad \text{Min}(B)
\]

- Remember: Aggregations return a single row – can’t combine with non-aggregates in projection

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department:
  \[
  \Pi_{\text{avg}(\text{salary})} (\sigma_{\text{dept}_\text{name} = '\text{Comp. Sci.'}} (\text{instructor})),
  \]
  \[
  \text{select avg}(\text{salary})
  \text{from} \text{instructor}
  \text{where} \text{dept}_\text{name} = '\text{Comp. Sci.'};
  \]
- Find the total number of instructors who teach a course in the Spring 2018 semester:
  \[
  \text{select count}(\text{distinct ID})
  \text{from} \text{teaches}
  \text{where} \text{semester} = '\text{Spring}' \text{ and year} = 2018;
  \]
- Find the number of tuples in the course relation:
  \[
  \text{select count}(*)
  \text{from} \text{course};
  \]
Grouping Operator

\[ \gamma_L(R) \], where \( L \) is a list of elements that are either

a) Individual (grouping) attributes or

b) Of the form \( \theta(A) \), where \( \theta \) is an aggregation operator and \( A \) the attribute to which it is applied,

is computed by:

1. Group \( R \) according to all the grouping attributes on list \( L \).
2. Within each group, compute \( \theta(A) \), for each element \( \theta(A) \) on list \( L \).
3. Result is the relation whose columns consist of one tuple for each group. The components of that tuple are the values associated with each element of \( L \) for that group.

Aggregate Functions – Group By

- Find the average salary of instructors in each department: \( \gamma_{\text{dept.name, avg(salary)}}(\text{instructor}) \)

  - \( \text{select dept.name, avg(salary) as avg_salary from instructor group by dept.name;} \)

<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 (Cont.)

- Attributes in `select` clause outside of aggregate functions must appear in `group by` list
  - /* erroneous query */
  
  ```sql
  select dept_name, ID, avg(salary)
  from instructor
  group by dept_name;
  ```

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) as 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
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 (Cont.)

- 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.
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:

```sql
SELECT A1, A2, ..., An
FROM r1, r2, ..., rm
WHERE P
```

as follows:
- **From clause**: r<sub>i</sub> 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 defined later.
- **Select clause**: A<sub>i</sub> can be replaced by a subquery that generates a single value.

“Breaking” the Model

- Some SQL constructs break the traditional relational model:

  ```sql
  SELECT bar
  FROM sells
  WHERE beer IN
  (SELECT favorite_beer FROM drinkers);
  ```
- What is the equivalent relational algebra?
  - Why does it break the model?
Relational Algebra

\[ \begin{align*}
\sigma & \text{ SELECT} \\
\pi & \text{ PROJECT} \\
\times & \text{ CARTESIAN PRODUCT} \\
\cup & \text{ UNION} \\
\setminus & \text{ SET-DIFFERENCE} \\
\cap & \text{ SET-INTERSECTION} \\
\bowtie & \text{ THETA-JOIN} \\
\bowtie & \text{ NATURAL JOIN} \\
\div & \text{ DIVISION or QUOTIENT}
\end{align*} \]

\[ \begin{align*}
\text{UNARY} & \quad \text{BINARY} \\
\text{FUNDAMENTAL} & \quad \text{CAN BE DEFINED} \\
& \quad \text{IN TERMS OF} \\
& \quad \text{FUNDAMENTAL OPS}
\end{align*} \]

SQL
SQL History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- 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.

SQL Parts

- DML -- provides the ability to query information from the 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 – includes 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 – includes commands for specifying access rights to relations and views.
The Rename Operation (SQL)

- 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’.
  
  - `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 ≡ instructor T`

Self Join Example

- Relation `emp-super`

<table>
<thead>
<tr>
<th>person</th>
<th>supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Alice</td>
</tr>
<tr>
<td>Mary</td>
<td>Susan</td>
</tr>
<tr>
<td>Alice</td>
<td>David</td>
</tr>
<tr>
<td>David</td>
<td>Mary</td>
</tr>
</tbody>
</table>

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Can you find ALL the supervisors (direct and indirect) of “Bob”?
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”.
  ```sql
  select name
  from instructor
  where name like '%dar%'
  ```
- Match the string “100%”
  ```sql
  like '100\%' escape '\'
  ```
  in that above we use backslash (`\`) as the escape character.

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`
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
    ) as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

Data Definition Language

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 stores 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 are covered in Chapter 4.

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_n))
  ```
  
  - \( r \) 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** \( (A_1, ..., A_n) \)
  - **foreign key** \( (A_{m'}, ..., A_{n'}) \) 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);
```

And a Few More Relation Definitions

- **create table** student (

```sql
    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 (  

```sql
    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);
```
And more still

- **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);

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`
    - where `A` is the name of the attribute to be added to relation `r` and `D` is the domain of `A`.
    - All exiting tuples in the relation are assigned `null` as the value for the new attribute.
  - `alter table r drop A`
    - where `A` is the name of an attribute of relation `r`
    - Dropping of attributes not supported by many databases.
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
  
  `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
  
  ```sql
  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)

**Insertion**

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

  - or equivalently
  
  ```sql
  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
  
  ```sql
  insert into student
  values ('3003', 'Green', 'Finance', null);
  ```
Insertion (Cont.)

- 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
  
  ```sql
  update instructor
  set salary = salary * 1.05
  ```

- Give a 5% salary raise to those instructors who earn less than 70000
  
  ```sql
  update instructor
  set salary = salary * 1.05
  where salary < 70000;
  ```

- Give a 5% salary raise to instructors whose salary is less than average
  
  ```sql
  update instructor
  set salary = salary * 1.05
  where salary < (select avg (salary)
                  from instructor);
  ```
Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - 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
  
  \[
  \text{update student } S \\
  \text{set } \text{tot_creds} = (\text{select sum(credits)} \text{ from takes, course} \\
  \text{where takes.course_id} = \text{course.course_id} \text{ and} \\
  \text{S.ID} = \text{takes.ID and} \\
  \text{takes.grade} \neq 'F' \text{ and} \text{takes.grade is not null});
  \]

- Sets tot_creds to null for students who have not taken any course
- Instead of sum(credits), use:
  
  \[
  \text{case} \\
  \text{when sum(credits) is not null then sum(credits)} \\
  \text{else 0} \\
  \text{end}
  \]