Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: $\sigma$
  - project: $\Pi$
  - union: $\cup$
  - set difference: $-$
  - Cartesian product: $\times$
  - rename: $\rho$
Select Operation

- The **select** operation selects tuples that satisfy a given predicate.
- Notation: \( \sigma_p(r) \)
- \( p \) is called the **selection predicate**
- Example: select those tuples of the `instructor` relation where the instructor is in the “Physics” department.
  - Query
    \[ \sigma_{dept\_name=\text{"Physics"}}(\text{instructor}) \]
  - Result

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
</tbody>
</table>

Select Operation (Cont.)

- We allow comparisons using
  \( =, \neq, >, \geq, <, \leq \)
in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives:
  \( \land \) (and), \( \lor \) (or), \( \neg \) (not)
- Example: Find the instructors in Physics with a salary greater $90,000, we write:
  \[ \sigma_{dept\_name=\text{"Physics"} \land salary > 90,000}(\text{instructor}) \]
- The select predicate may include comparisons between two attributes.
  - Example, find all departments whose name is the same as their building name:
  - \( \sigma_{dept\_name=\text{building}}(\text{department}) \)
Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:
  \[ \Pi_{A_1, A_2, A_3 \ldots A_k} (r) \]
  where \( A_1, A_2, \ldots, A_k \) are attribute names and \( r \) is a relation name.
- The result is defined as the relation of \( k \) columns obtained by erasing the columns that are not listed.
- Duplicate rows removed from result, since relations are sets.

Project Operation Example

- Example: eliminate the \textit{dept\_name} attribute of \textit{instructor}
- Query:
  \[ \Pi_{ID, name, salary} (instructor) \]
- Result:

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>65000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>90000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>40000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>60000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>75000</td>
</tr>
<tr>
<td>58583</td>
<td>Calieri</td>
<td>62000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>72000</td>
</tr>
</tbody>
</table>
Composition of Relational Operations

- The result of a relational-algebra operation is a relation and therefore of relational-algebra operations can be composed together into a relational-algebra expression.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{\text{name}}(\sigma_{\text{dept\_name} = \text{"Physics"}}(\text{instructor}))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

- Select name from instructor where dept\_name = 'Physics';

Cartesian-Product Operation

- The Cartesian-product operation (denoted by X) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations instructor and teaches is written as:

  $$\text{instructor} \ X \ \text{teaches}$$

- We construct a tuple of the result out of each possible pair of tuples: one from the instructor relation and one from the teaches relation (see next slide)
- Since the instructor ID appears in both relations we distinguish between these attributes by attaching to the attribute the name of the relation from which the attribute originally came.
  - instructor.ID
  - teaches.ID
Join Operation

- The Cartesian-Product 
  \( \text{instructor} \times \text{teaches} \)
  associates every tuple of instructor with every tuple of teaches.
  - Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of \( \text{instructor} \times \text{teaches} \) that pertain to instructors and the courses that they taught, we write:
  \[ \sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor \times teaches}) \]
  - We get only those tuples of \( \text{instructor} \times \text{teaches} \) that pertain to instructors and the courses that they taught.
- The result of this expression, shown in the next slide

### The instructor X teaches table

<table>
<thead>
<tr>
<th>instructor.ID</th>
<th>name</th>
<th>dept.name</th>
<th>salary</th>
<th>teaches.ID</th>
<th>course.ID</th>
<th>sec.id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>12121</td>
<td>FEN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>12121</td>
<td>FEN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>12121</td>
<td>FEN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>12121</td>
<td>FEN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Join Operation (Cont.)

- The table corresponding to:

\[ \sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \times \text{teaches}) \]

Select *
from instructor,teaches
where instructor.id=teaches.id

<table>
<thead>
<tr>
<th>instructor.ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
<th>teaches.ID</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>12121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
<td>32343</td>
<td>HIS-351</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>45565</td>
<td>CS-101</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>45565</td>
<td>CS-319</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>76766</td>
<td>BIO-101</td>
<td>1</td>
<td>Summer</td>
<td>2017</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>76766</td>
<td>BIO-301</td>
<td>1</td>
<td>Summer</td>
<td>2018</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-190</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2018</td>
</tr>
</tbody>
</table>

Join Operation (Cont.)

- The join operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations \( r(\mathcal{R}) \) and \( s(\mathcal{S}) \)
- Let “theta” be a predicate on attributes in the schema \( \mathcal{R} \) “union” \( \mathcal{S} \). The join operation \( r \bowtie_{\theta} s \) is defined as follows:

\[ r \bowtie_{\theta} s = \sigma_{\theta} (r \times s) \]

- Thus

\[ \sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \times \text{teaches}) \]

- Can equivalently be written as

\[ \text{instructor} \bowtie_{\text{instructor.id} = \text{teaches.id}} \text{teaches}. \]
Union Operation

- The union operation allows us to combine two relations
- Notation: \( r \cup s \)
- For \( r \cup s \) to be valid.
  1. \( r, s \) must have the same arity (same number of attributes)
  2. The attribute domains must be compatible (example: 2\(^{nd}\) column of \( r \) deals with the same type of values as does the 2\(^{nd}\) column of \( s \))
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

\[
\begin{align*}
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \land \text{year}=\text{2017}} (\text{section})) \cup \\
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \land \text{year}=\text{2018}} (\text{section}))
\end{align*}
\]

Union Operation (Cont.)

- Result of:

\[
\begin{align*}
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \land \text{year}=\text{2017}} (\text{section})) \cup \\
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \land \text{year}=\text{2018}} (\text{section}))
\end{align*}
\]

<table>
<thead>
<tr>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
</tr>
<tr>
<td>CS-319</td>
</tr>
<tr>
<td>CS-347</td>
</tr>
<tr>
<td>FIN-201</td>
</tr>
<tr>
<td>HIS-351</td>
</tr>
<tr>
<td>MU-199</td>
</tr>
<tr>
<td>PHY-101</td>
</tr>
</tbody>
</table>
Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: \( r \cap s \)
- Assume:
  - \( r, s \) have the same arity
  - attributes of \( r \) and \( s \) are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.
  \[
  \Pi_{\text{course_id}} (\sigma_{\text{semester} = "Fall" \land \text{year} = 2017} (\text{section})) \cap \\
  \Pi_{\text{course_id}} (\sigma_{\text{semester} = "Spring" \land \text{year} = 2018} (\text{section}))
  \]
- Result

\[
\begin{array}{l}
\text{course_id} \\
\text{CS-101}
\end{array}
\]

Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation \( r - s \)
- Set differences must be taken between compatible relations.
  - \( r \) and \( s \) must have the same arity
  - attribute domains of \( r \) and \( s \) must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester
  \[
  \Pi_{\text{course_id}} (\sigma_{\text{semester} = "Fall" \land \text{year} = 2017} (\text{section})) - \\
  \Pi_{\text{course_id}} (\sigma_{\text{semester} = "Spring" \land \text{year} = 2018} (\text{section}))
  \]
- Result

\[
\begin{array}{l}
\text{course_id} \\
\text{CS-347} \\
\text{PHY-101}
\end{array}
\]
The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by \( \leftarrow \) and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

\[
\begin{align*}
\text{Physics} & \leftarrow \sigma_{\text{dept.name}=\text{“Physics”}}(\text{instructor}) \\
\text{Music} & \leftarrow \sigma_{\text{dept.name}=\text{“Music”}}(\text{instructor}) \\
\text{Physics} & \cup \text{Music}
\end{align*}
\]

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, \( \rho \), is provided for that purpose.
- The expression:

\[
\rho_x(E)
\]

returns the result of expression \( E \) under the name \( x \).
- Another form of the rename operation:

\[
\rho_{x(\bar{A}_1, \bar{A}_2, .. \bar{A}_n)}(E)
\]
Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1
  \[ \sigma_{\text{dept}_\text{name} = \text{"Physics"}} (\text{instructor} \bowtie \text{teaches}) \]
- Query 2
  \[ (\sigma_{\text{dept}_\text{name} = \text{"Physics"}} (\text{instructor})) \bowtie \text{teaches} \]
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1
  \[ \sigma_{\text{dept}_\text{name} = \text{"Physics"} \land \text{salary} > 90,000} (\text{instructor}) \]
- Query 2
  \[ \sigma_{\text{dept}_\text{name} = \text{"Physics"}} (\sigma_{\text{salary} > 90,000} (\text{instructor})) \]
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.
Extended Projection

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

• Example

• \( R = \pi_{A+B, A, A}(R) = \begin{array}{c|c} A & B \\ \hline 1 & 2 \\ 3 & 4 \end{array} \)


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:

• \( R = \begin{array}{c|c} A & B \\ \hline 1 & 2 \\ 3 & 4 \end{array} \)

• \( \pi_{\text{Max}(A), \text{Min}(B)}(R) = \begin{array}{c|c|c} \text{Max}(A) & \text{Min}(B) \\ \hline 3 & 2 \end{array} \)

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

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department:
  \[ \Pi \text{avg(salary)}(\sigma_{\text{dept\_name} = 'Comp. Sci.'}(\text{instructor})) \]
  
  - select \text{avg} (\text{salary})
    from \text{instructor}
    where \text{dept\_name} = 'Comp. Sci.';

- Find the total number of instructors who teach a course in the Spring 2018 semester:
  
  - select \text{count} (\text{distinct ID})
    from \text{teaches}
    where \text{semester} = 'Spring' and \text{year} = 2018;

- Find the number of tuples in the \text{course} relation:
  
  - select \text{count} (*)
    from \text{course};
Grouping Operator

\[ \gamma_L(R), \text{ where } L \text{ is a list of elements that are either} \]
\[ \text{a) Individual (grouping) attributes or} \]
\[ \text{b) Of the form } \theta(A), \text{ where } \theta \text{ is an aggregation operator} \]
\[ \text{and } A \text{ the attribute to which it is applied,} \]
\[ \text{is computed by:} \]
\[ \text{1. Group } R \text{ according to all the grouping attributes on list } L. \]
\[ \text{2. Within each group, compute } \theta(A), \text{ for each element } \theta(A) \text{ on list } L. \]
\[ \text{3. Result is the relation whose columns consist of one tuple for} \]
\[ \text{each group. The components of that tuple are the values} \]
\[ \text{associated with each element of } L \text{ for that group.} \]

Aggregate Functions – Group By

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

  - \textbf{select} dept\_name, avg(salary) as avg_salary
  - \textbf{from} instructor
  - \textbf{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;
    ```

Aggregate Functions – Having Clause

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

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

as follows:

- **From clause**: $r_i$ can be replaced by any valid subquery.
- **Where clause**: $P$ can be replaced with an expression of the form:

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

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

“Breaking” the Model

- Some SQL constructs break the traditional relational model
  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

σ SELECT
π PROJECT
X CARTESIAN PRODUCT
∪ UNION
– SET-DIFFERENCE
∩ SET-INTERSECTION
⋈ THETA-JOIN
⋈ NATURAL JOIN
÷ DIVISION or QUOTIENT

UNARY
FUNDAMENTAL
BINARY

CAN BE DEFINED
IN TERMS OF
FUNDAMENTAL OPS

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'.
  
  ```sql
  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
  
  ```sql
  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\%'
  ```
  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_k))
  ```

  - \( r \) is the name of the relation
  - each \( A_i \) is an attribute name in the schema of relation \( r \)
  - \( D_i \) is the data type of values in the domain of attribute \( A_i \)

- Example:

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

- Types of integrity constraints
  - **primary key** \((A_1, ..., A_n)\)
  - **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),  
department 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

\[
\text{delete from instructor}
\]

- Delete all instructors from the Finance department

\[
\text{delete from instructor}
\quad \text{where} \quad \text{dept\_name} = \text{'Finance'};
\]

- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.

\[
\text{delete from instructor}
\quad \text{where} \quad \text{dept\_name in (select dept\_name}
\quad \text{from department}
\quad \text{where} \quad \text{building = 'Watson')};
\]
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)

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.)

- 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:
    ```sql
    update instructor
    set salary = salary * 1.03
    where salary > 100000;
    update instructor
    set salary = salary * 1.05
    where salary <= 100000;
    ```
  - The order is important
  - Can be done better using the `case` statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement
  ```sql
  update instructor
  set salary = case
    when salary <= 100000 then salary * 1.05
    else salary * 1.03
  end
  ```
Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

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

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

- Instead of sum(credits), use:

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