Advanced Data Management (CSCI 680/490)

Databases

Dr. David Koop
Python Features

- Iterators: for loops use to go through elements
  - `it = iter(d.values()); next(it)`
- Comprehensions: succinct computations over collections (map & filter)
  - `squares = [i**2 for i in range(10) if i % 3 != 1]`
- Exceptions: deal with errors when desired, allow aggregation
  - `try-except-else-finally`
- Object-Oriented Programming:
  - Class definitions (`__init__`, `self`)
  - Using object `obj`: `obj.field`, `obj.function()`
Databases & DBMSes

• Database:
  - Basically, just structured data/information stored on a computer
  - Very generic, doesn't specify specific way that data is stored
  - Can be single-file (or in-memory) or much more complex

• Database Management System (DBMS):
  - Software to manage databases
  - Instead of each program writing its own methods to manage data, abstract data management to the DBMS
  - Specify structure of the data (schema)
  - Provide query capabilities
Data Models

• The data model specifies:
  - what data can be stored (and sometimes how it is stored)
  - associations between different data values
  - what constraints can be enforced
  - how to access and manipulate the data

• Relational model
• Entity-Relationship data model (mainly for database design)
• Object-based data models (Object-oriented and Object-relational)
• Semistructured data model (XML)
• Network Model

[A. Silberschatz et al.]
Assignment 1

• Due Monday
• Using Python for data analysis on MoMA data
• Use basic python for now to work on language knowledge
• Use Anaconda or a hosted Python environment
• Turn .ipynb file in via Blackboard
Relational Model History

• Invented by Edgar F. Codd in early 1970s
• Focus was data independence
  - Previous data models required physical-level design and implementation
  - Changes to a database schema were very costly to applications that accessed the database
• IBM, Oracle were first implementers of relational model (1977)
  - Usage spread very rapidly through software industry
  - SQL was a particularly powerful innovation
Relations

- Relations are basically tables of data
  - Each row represents a **tuple** in the relation
- A relational database is an **unordered** set of relations
  - Each relation has a unique name in the database
- Each row in the table specifies a relationship between the values in that row
  - The account ID “A-307”, branch name “Seattle”, and balance “275” are all related to each other

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Relations and Attributes

- Each relation has some number of **attributes**
  - Sometimes called “columns”
- Each attribute has a **domain**
  - Set of valid values for the attribute (+ null)
  - Values are usually **atomic**
- The account relation has 3 attributes
  - Domain of balance is the set of nonnegative integers
  - Domain of branch_name is the set of all valid branch names in the bank

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Database Schema

- Database schema: the logical structure of the database.
- Database instance: a snapshot of the data at a given instant in time.
- Example Schema
  - instructor
    - (ID, name, dept_name, salary)

<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>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</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>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Keys

- Let $K \subseteq R$
- $K$ is a **superkey** of $R$ if values for $K$ are sufficient to identify a unique tuple of each possible relation $r(R)$
  - Example: $\{ID\}$ and $\{ID, name\}$ are both superkeys of instructor.
- Superkey $K$ is a **candidate key** if $K$ is **minimal**
  - Example: $\{ID\}$ is a candidate key for Instructor
- One of the candidate keys is selected to be the **primary key**.
  - Which one?
Foreign Key Constraints

- Foreign key constraint: Value in one relation must appear in another
  - Referencing relation
  - Referenced relation
  - Example: dept_name in instructor is a foreign key from instructor referencing department
Schema Diagram with Keys

Diagram showing relationships between tables such as `student`, `course`, `teaches`, `section`, `time_slot`, `classroom`, `prereq`, `instructor`, and `department`. Each table has attributes such as `ID`, `course_id`, `sec_id`, `semester`, `year`, `title`, `dept_name`, `credits`, `building`, `room_number`, `capacity`, `time_slot_id`, `day`, `start_time`, `end_time`, `prereq_id`, `name`, `salary`, and more.

[Reference: A. Silberschatz et al.]
Relational Query Languages

• Procedural versus non-procedural, or declarative
• “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
• The above 3 pure languages are equivalent in computing power
• Concentrate on relational algebra
  - Not Turing-machine equivalent
  - 6 basic operations
Relational Algebra

• Definition: 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: $\sigma_{\text{dept\_name}="Physics"}(\text{instructor})$
  - 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>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Select Operation Comparisons

- 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 than $90,000$:
  - $\sigma_{\text{dept}_\text{name}=\text{“Physics”}} \land \text{salary} > 90,000 \ (\text{instructor})$
- The select predicate may include comparisons between two attributes.
  - Example: departments whose name is the same as their building name:
    - $\sigma_{\text{dept}_\text{name}=\text{building}} \ (\text{department})$
Project Operation

• A unary operation that returns its argument relation, with certain attributes left out.

• Notation: $\prod_{A_1, A_2, A_3, \ldots, A_k} (r)$
  where $A_1, A_2, A_3, \ldots, A_k$ 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 `dept_name` attribute of instructor
- Query: $\prod_{ID, \text{name}, \text{salary}} (\text{instructor})$
Composition of Relational Operations

• The result of a relational-algebra operation is a relation
• … so relational-algebra operations can be composed together into a relational-algebra expression.

Example: 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.
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: *instructor* X *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

• Since the instructor ID appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.

  - *instructor*.ID and *teaches*.ID
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>FIN-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>...</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>FIN-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>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
</tbody>
</table>
Join Operation

- The Cartesian-Product instructor \( X \) 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 instructor \( X \) teaches that pertain to instructors and the courses that they taught, we write:
  \[ \sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \times \text{teaches}) \]
  - We get only those tuples of instructor \( X \) teaches that pertain to instructors and the courses that they taught.

[A. Silberschatz et al.]
### Join Operation (Cont.)

The table corresponding to \( \sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \bowtie \text{teaches}) \)

<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>2017</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-319</td>
<td>2</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
<td>98345</td>
<td>EE-181</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
</tr>
</tbody>
</table>
The join operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.

Consider relations $r(R)$ and $s(S)$.

Let $\theta$ be a predicate on attributes in the schema $R \cup S$. The join operation is:

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

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.
  - \( r, s \) must have the same arity (same number of attributes)
  - The attribute domains must be **compatible** (example: 2nd column of \( r \) deals with the same type of values as does the 2nd column of \( s \))
Union Example

- Find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both:

\[
\prod_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"}} \land \text{year}=2017 (\text{section})) \cup \\
\prod_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"}} \land \text{year}=2018 (\text{section}))
\]
Set-Intersection Operation

- The **set-intersection** operation allows us to find tuples that are in both the input relations.

- **Notation:** \( r \cap s \)

- Same requirements as union:
  - \( r, s \) 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.

- \( \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"}} \land \text{year}=2017 (\text{section})) \cap \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"}} \land \text{year}=2018 (\text{section})) \)
Set Difference Operation

- The **set-difference** operation allows us to find tuples that are in one relation but are not in another.

- Notation $r - s$

- Same requirements as union and set-intersection: 
  - $r$ and $s$ must have the same arity
  - attribute domains of $r$ and $s$ must be compatible

- Example: Find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{\text{course_id}} (\sigma_{\text{semester}="Fall" \land \text{year}=2017} (\text{section})) - \Pi_{\text{course_id}} (\sigma_{\text{semester}="Spring" \land \text{year}=2018} (\text{section}))$$
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: $\sigma_{\text{dept\_name}=\text{"Physics"}} \land \text{salary} > 90,000 (\text{instructor})$
  - Query 2: $\sigma_{\text{dept\_name}=\text{"Physics"}} (\sigma_{\text{salary} > 90.000} (\text{instructor}))$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.
Equivalent Queries

• Example: Find information about courses taught by instructors in the Physics department

• Query 1:

$$\sigma_{\text{dept\_name}=\text{"Physics"}} (\text{instructor} \Join \text{instructor.ID = teaches.ID})$$

• Query 2

$$(\sigma_{\text{dept\_name}=\text{"Physics"}} (\text{instructor})) \Join \text{instructor.ID = teaches.ID}$$

• The order of joins is one focus of some of the work on query optimization
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)


• Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

• Not all examples work on all systems
Components of SQL

- **Data Definition Language (DDL):** the specification of information about relations, including schema, types, integrity constraints, indices, storage

- **Data Manipulation Language (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.

- **Also:** Transaction control, embedded and dynamic SQL, authorization
Create Table

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

\[
\text{create table } r (A_1 D_1, A_2 D_2, \ldots, A_n D_n, (C_1), \ldots, (C_k))
\]

- \( r \) is the \textbf{name} of the relation
- each \( A_i \) is an \textbf{attribute name} in the schema of relation \( r \)
- \( D_i \) is the \textbf{data type} of values in the domain of attribute \( A_i \)
- \( C_i \) are integrity constraints

- Example:

\[
\text{create table instructor (}
\begin{align*}
\text{ID} & \quad \text{char}(5), \\
\text{name} & \quad \text{varchar}(20), \\
\text{dept\_name} & \quad \text{varchar}(20), \\
\text{salary} & \quad \text{numeric}(8,2));
\end{align*}
\]
**Create Table**

- 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, (C_1), ..., (C_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 $A_i$
  - $C_i$ are integrity constraints

- Example:
  
  ```sql
  create table instructor(
    ID char(5),
    name varchar(20),
    dept_name varchar(20),
    salary numeric(8,2));
  ```
Create Table

• 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, (C_1), ..., (C_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))
  ```

\( C_i \) are integrity constraints
Create Table

• 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, (C_1), ..., (C_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))
```

C_i are integrity constraints
Integrity Constraints in Create Table

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

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

- 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);
Updates to tables

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

- Delete: `delete from student;` -- remove all tuples from student

- Drop Table: `drop table r`

- Alter: `alter table r add A D; alter table r drop A`
  - `A` is the name of the attribute to be added to relation `r`
  - `D` is the domain of `A`
  - All exiting tuples are assigned `null` for the new attribute's value
  - Dropping of attributes not widely supported

[A. Silberschatz et al.]
Basic Query Structure

• A typical SQL query has the form:

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

   - $A_i$ represents an **attribute**
   - $r_i$ represents a **relation**
   - $P$ is a **predicate**.

• The result of an SQL query is a **relation**
Select

• 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
  - **select** name
    from instructor;

• Note: SQL names are **case insensitive**
  - Name and NAME and name are equivalent
  - Some people use upper case for language keywords (e.g. **SELECT**)
Select

- SQL allows **duplicates** in relations as well as in query results.
- To eliminate duplicates, put the keyword `distinct` after `select`.
- Example: Find the department names of all instructors (no duplicates)
  
  - `select distinct dept_name
    from instructor;`

- The keyword **all** specifies that duplicates should not be removed

  - `select all dept_name
    from instructor;`
Select

- An asterisk (*) in the select clause denotes “all attributes”
  - `select * from instructor;

- An attribute can be a literal with no from clause (select '437')
  - Result is a table with one column and a single row with value '437'
  - Can give the column a name using as: `select '437' as FOO`

- An attribute can be a literal with from clause:
  - `select 'A' from instructor`
  - Result is a table with one column and $N$ rows (number of tuples in the instructors table), each row with value “A”
Select "Math"

- The select clause can contain **arithmetic expressions** involving the operation, +, −, *, and /, and operating on constants or attributes of tuples.

- 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.

- Can rename expressions using the **as** clause:

  ```sql
  - select ID, name, salary/12 as monthly_salary
  ```
Where

- The **where** clause specifies conditions that the result must satisfy
  - Confusingly corresponds to the **selection** predicate in relational algebra
- Example: Find all instructors in Comp. Sci. dept
  - **select** name
    **from** instructor
    **where** dept_name = 'Comp. Sci.'
Where

- The operands can be expressions with operators $<, \leq, >, \geq, =,$ and $\neq$
- SQL allows the use of the logical connectives $\text{and}$, $\text{or}$, and $\text{not}$
- Comparisons can be applied to results of arithmetic expressions
- Example: Find all instructors in Comp. Sci. with salary $> 70000$

```sql
- select name
  from instructor
  where dept_name = 'Comp. Sci.' and salary > 70000
```

[A. Silberschatz et al.]
From

- The `from` clause lists the relations involved in the query
  - Corresponds to the **Cartesian Product** operation in relational algebra
- Find the Cartesian product `instructor X teaches`
  - `select *`
    - `from instructor, teaches;`
  - All possible `instructor - teaches` pairs, with all attributes from both
  - Shared attributes (e.g., ID) are renamed (e.g., instructor.ID)
- Not very useful directly but useful combined with where clauses.
From

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

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

<table>
<thead>
<tr>
<th>name</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Srinivasan</td>
<td>CS-101</td>
</tr>
<tr>
<td>Srinivasan</td>
<td>CS-315</td>
</tr>
<tr>
<td>Srinivasan</td>
<td>CS-347</td>
</tr>
<tr>
<td>Wu</td>
<td>FIN-201</td>
</tr>
<tr>
<td>Mozart</td>
<td>MU-199</td>
</tr>
<tr>
<td>Einstein</td>
<td>PHY-101</td>
</tr>
<tr>
<td>El Said</td>
<td>HIS-351</td>
</tr>
<tr>
<td>Katz</td>
<td>CS-101</td>
</tr>
<tr>
<td>Katz</td>
<td>CS-319</td>
</tr>
<tr>
<td>Crick</td>
<td>BIO-101</td>
</tr>
<tr>
<td>Crick</td>
<td>BIO-301</td>
</tr>
<tr>
<td>Brandt</td>
<td>CS-190</td>
</tr>
<tr>
<td>Brandt</td>
<td>CS-190</td>
</tr>
<tr>
<td>Brandt</td>
<td>CS-319</td>
</tr>
<tr>
<td>Kim</td>
<td>EE-181</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
The Rename Operation

- SQL allows renaming relations and attributes using the `as` clause:
  - `old-name as new-name`

- Example: 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` is equivalent to `instructor T`
Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018
  - (select course_id from section where sem = 'Fall' and year = 2017)
    union
  - (select course_id from section where sem = 'Spring' and year = 2018)

- Find courses that ran in Fall 2017 and in Spring 2018
  - (select course_id from section where sem = 'Fall' and year = 2017)
    intersect
  - (select course_id from section where sem = 'Spring' and year = 2018)

- Find courses that ran in Fall 2017 but not in Spring 2018
  - (select course_id from section where sem = 'Fall' and year = 2017)
    except
  - (select course_id from section where sem = 'Spring' and year = 2018)
Aggregate Functions

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

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

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