CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu

Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Today’s Plan

- Discuss reading assignment topics
  - Keys, Writing SQL queries, Different types of Joins, Aggregates
- New topics to discuss
  - Outerjoins, Views, Formal Semantics of SQL, Integrity Constraints, Triggers, Transactions

- Other things
  - Grading of reading homeworks (ignore ELMS score)
  - Computing environment setup
  - Small change in the late days policy (max 4 days for any assignment instead of 5, so I can discuss in class on Wed)
  - Let me know if you are not okay with using your answers (without attribution) to illustrate a point

Keys

- Let $K \subseteq R$
- $K$ is a superkey of $R$ if values for $K$ are sufficient to identify a unique tuple of any possible relation $r(R)$
  - Example: $\{ID\}$ and $\{ID, name\}$ are both superkeys of instructor.
- Superkey $K$ is a candidate key if $K$ is minimal (i.e., no subset of it is a superkey)
  - Example: $\{ID\}$ is a candidate key for Instructor
- One of the candidate keys is selected to be the primary key
  - Typically one that is small and immutable (doesn’t change often)
  - Primary key typically highlighted (e.g., underlined)
Your questions

- How to choose a primary key?
  - Identify candidate keys → pick one that is more immutable
  - Add one if no natural candidate key

- Difference between candidate keys and super keys?
  - Superkeys are not “minimal” – we can remove some attributes
  - e.g., (ID, name), (ID, salary), etc. for the instructor table

- Using “randomly” generated primary keys
  - Nothing specifically wrong – you see that sometimes
  - No real benefit over using a SERIAL (next slide)

- Does primary key need to be the first attribute in the schema? NO

Keys

- Auto-generated primary keys
  - CREATE TABLE abc (ID SERIAL);

- Equivalent to:
  - CREATE SEQUENCE abc_id_seq;
  - CREATE TABLE abc (ID integer NOT NULL DEFAULT nextval('abc_id_seq'));
  - ALTER SEQUENCE abc_id_seq OWNED BY abc.ID;

- Later (explicit) representation allows sharing of a sequence by multiple tables

- Problems with auto-generated keys
  - Only use if needed -- many tables already have a primary key
  - Perfectly fine to have compound primary keys – using separate ID may lead to problems
    - E.g., Imagine an ID column added to the “borrower” table
Keys

For the instructor table, choose all superkeys under the assumptions that it is possible for there to be two instructors with the same name in the same department, but they would not have the same salaries.

Consider the banking database depicted in Figure 2.15. What is the primary key for the borrower table?

.Keys

In the University database: let’s say for some of the courses, we allow the students to pair up (i.e., form groups of two) and do the assignments together. We would like to keep track of this information in a separate table:

\[
\text{course_pairs}\langle \text{s_id1, s_id2, course_id, sec_id, semester, year}\rangle,
\]

indicating that those two students formed a group for that particular course. For different courses, students may form different groups, but for any specific offering, each student can only participate in at most one such group.

Which of the following could be a primary key for this table?

- s_id1, s_id2
- s_id1, course_id, sec_id, semester, year
- s_id2, course_id, sec_id, semester, year
- s_id1, s_id2, course_id, sec_id, semester, year
- s_id1, s_id2, course_id, semester, year
- s_id1, course_id, semester, year
Keys

- Movies(title, year, length, director, studio)

- UBER Example:
  - customer(name, address, credit_card)
  - driver(SSN, name, address, date_started)
  - trip(driver_SSN, customer_name, address_picked_up, address_dropped_off, datetime_picked_up, datetime_dropped_off, fare)
Your questions

- Can a primary key consisting of multiple attributes be used as a foreign key?
  - Sure. See “Takes” in the University schema. Fairly common.

- Is it possible for a relation to consist of only foreign keys?
  - Sure. See “Advisor” or “Prereq” tables. This is common for what are called “relationship” tables which simply record the relationship between two types of entities (e.g., students having advisors, etc).

- If NULL in database creates so many headaches, why don’t we just use a default value for each data type such as empty string for a string?
  - This is a good question, but complicated to answer. Default value is still a value, whereas NULL indicates missing/unknown values. Both have their uses, and you have to choose which is more appropriate.

Your questions

- Is there an advantage of using a WITH clause in an SQL rather running it as a subquery in the WHEN clause?
  - WITH leads to cleaner queries, and also allows reuse (see ex below). Either is fine – they get evaluated similarly at the end.
  - with temp as (select * from R) select ... from temp t1, temp t2 where ...
    - This would be more complicated using subqueries because of repetition

- I cannot seem to understand why natural join is not preferable as mentioned in the videos very well.
  - There is a higher chance of missing some common attributes which can lead to unintended results especially for large schemas, so I personally prefer to avoid them – less headache
Your questions

- When joining two tables can we select all of the columns from one table and only a subset of the columns of the second? If we can, how do we select those columns? Do we need to list them out?
  - "select friends.*, users.name from friends natural join users;"
- In which case would intersect be useful?
  - e.g., to find instructors who are teaching in both Spring 2010 and Fall 2010
- What is "T.salary S.salary" in "select distinct T.name from instructor as T, instructor as S where T.salary S.salary and S.dept_name = 'Biology'"? although the book gives the description of the query, but it seems the clause seems to conflict each other since T and S are aliasing the same table.
  - It is simpler to think of aliasing as creating copies of the table in such cases, so the two references are indeed uniquely defined

Your questions

- If one attribute of a table is a reference to another table, I don't understand why the name of the attributes do not have to match. How does PSQL know that the foreign key references that attribute?
  - You explicitly tell PSQL in the create table statement (using "references")
- Do any SQL implementations support "on change" actions, similar to "on delete"? For instance, if the value of a primary variable in one relation were to be changed, is there support for automatically updating all foreign relations / references in all other relations?
  - "on update"
- How would you search for an instructor's name that is of at length 4 but not greater then 12?
  - SQL probably has a length() function that you can use to get the length
- Is a natural join on two tables with no shared columns just a cartesian product of the two tables?
  - Correct
Find the names of all instructors:

```
select name
from instructor
```

More complex filters:
```
select name
from instructor
where (dept_name != 'Finance' and salary > 75000)
or (dept_name = 'Finance' and salary > 85000);
```

A filter with a subquery:
```
select name
from instructor
where dept_name in (select dept_name from department where budget < 100000);
```
Multi-table Queries

Cartesian product:
```
select *
from instructor, department
```

Use predicates to only select “matching” pairs:
```
select *
from instructor i, department d
where i.dept_name = d.dept_name;
```

Identical (in this case) to using a natural join:
```
select *
from instructor natural join department;
```

Natural join does an equality on common attributes – doesn’t work here:
```
select *
from instructor natural join advisor;
```

Instead can use “on” construct (or where clause as above):
```
select *
from instructor join advisor on (i_id = id);
```

3-Table Query to get a list of instructor-teaches-course information:
```
select i.name as instructor_name, c.title as course_name
from instructor i, course c, teaches
where i.ID = teaches.ID and c.course_id = teaches.course_id;
```

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn’t
```
select name, title
from instructor natural join course natural join teaches;
```

I prefer avoiding “natural joins” for that reason

Note: On the small dataset, the above two have the same answer, but not on the large dataset. Large dataset has cases where an instructor teaches a course from a different department.
Aggregates: Group By

Split the tuples into groups, and compute the aggregate for each group

\[
\text{select } \text{dept\_name, avg(salary)} \\
\text{from instructor} \\
\text{group by dept\_name} \\
\text{having avg(salary) > 42000;}
\]

<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>

SQL: Nulls

Boolean Operations with Null

\[n < \text{NULL} = \text{UNKNOWN} \quad \text{(similarly for all boolean ops: } >, \leq, \geq, <\text{, }>\text{, }=, \ldots)\]

\[
e.g. \quad \text{branch} = \\
\begin{array}{|c|c|}
\hline
\text{bname} & \text{bcity} \\
\hline
\text{Downtown} & \text{Boston} \\
\hline
\text{Perry} & \text{Horseneck} \\
\hline
\text{Mianus} & \text{Horseneck} \\
\hline
\text{Waltham} & \text{Boston} \\
\hline
\end{array}
\]

\[
\text{SELECT *} \\
\text{FROM branch} \\
\text{WHERE assets IS NULL}
\]

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waltham</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
SQL: Unknown

Boolean Operations with Unknown

\[ n < \text{NULL} = \text{UNKNOWN} \quad (\text{similarly for all boolean ops:} \, >, \, <=, \, >=, \, <>, \, =, \, ... ) \]

\begin{align*}
\text{FALSE OR UNKNOWN} & = \text{UNKNOWN} \\
\text{TRUE AND UNKNOWN} & = \text{UNKNOWN}
\end{align*}

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

\begin{align*}
\text{UNKNOWN OR UNKNOWN} & = \text{UNKNOWN} \\
\text{UNKNOWN AND UNKNOWN} & = \text{UNKNOWN} \\
\text{NOT (UNKNOWN)} & = \text{UNKNOWN}
\end{align*}

\text{Can write:}
\begin{align*}
\text{SELECT} & \ldots \\
\text{FROM} & \ldots \\
\text{WHERE booleanexp IS UNKNOWN}
\end{align*}

UNKNOWN tuples are not included in final result

Outline

- More SQL
  - Outerjoins
  - Views
  - Formal semantics of SQL
  - Integrity Constraints
  - Triggers
  - Transactions
Consider the query:

```sql
select course_id, title, count(*)
from course
natural join prereq
group by course_id, title;
```

You expect:
- A list of course id/titles with number of prereqs for each

But:
- What about courses that don’t have any prereq?
- We would like to get “(course_id, title, 0)” for them

### Outer Joins

<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>

<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>

prereq information is missing for CS-315 and course information is missing for CS-437 (wouldn’t be allowed by referential integrity constraint, but let’s say it is)

<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>
</tbody>
</table>
### Outer Joins

#### 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>

#### 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>

#### 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>Game Design</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>

#### 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>Robotics</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

#### 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>Game Design</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>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>
Additional Operators: Joins

- **Equi-join**
  - A join that only has equality conditions

- **Theta-join** ($\bowtie_{\theta}$)
  - $r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$

- **Left outer join** ($\bowland$s)
  - Say $r(A, B)$, $s(B, C)$
  - We need to somehow find the tuples in $r$ that have no match in $s$
  - Consider: $(r - \pi_{r.A, r.B}(r \bowland s))$
  - We are done:

$$ (r \bowland s) \cup \rho_{temp(A, B, C)} \left( (r - \pi_{r.A, r.B}(r \bowland s)) \times \{\text{NULL}\} \right)$$

Additional Operators: Join Variations

- **Tables**: $r(A, B)$, $s(B, C)$

<table>
<thead>
<tr>
<th>name</th>
<th>Symbol</th>
<th>SQL Equivalent</th>
<th>RA expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross product</td>
<td>$\times$</td>
<td>select * from $r$, $s$;</td>
<td>$r \times s$</td>
</tr>
<tr>
<td>natural join</td>
<td>$\bowland$</td>
<td>natural join</td>
<td>$\pi_{r.A, r.B, s.C} \sigma_{r.B = s.B}(r \times s)$</td>
</tr>
<tr>
<td>theta join</td>
<td>$\bowland_{\theta}$</td>
<td>from .. where $\theta$;</td>
<td>$\sigma_{\theta}(r \times s)$</td>
</tr>
<tr>
<td>equi-join</td>
<td>$\bowland_{\theta}$</td>
<td>(theta must be equality)</td>
<td></td>
</tr>
<tr>
<td>left outer join</td>
<td>$r \bowland s$</td>
<td>left outer join (with “on”);</td>
<td>(see previous slide)</td>
</tr>
<tr>
<td>full outer join</td>
<td>$r \bowland s$</td>
<td>full outer join (with “on”);</td>
<td>–</td>
</tr>
<tr>
<td>(left) semijoin</td>
<td>$r \bowland s$</td>
<td>none</td>
<td>$\pi_{r.A, r.B}(r \bowland s)$</td>
</tr>
<tr>
<td>(left) antijoin</td>
<td>$r \bowland s$</td>
<td>none</td>
<td>$r - \pi_{r.A, r.B}(r \bowland s)$</td>
</tr>
</tbody>
</table>
Outline

- More SQL
  - Outerjoins
  - Views
  - Formal semantics of SQL
  - Integrity Constraints
  - Triggers
  - Transactions

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

  ```sql
  create view v as <query expression>
  where:
  <query expression> is any legal expression
  The view name is represented by v
  ```

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it
Example Queries

- A view of instructors without their salary
  ```sql
  create view faculty as
  select ID, name, dept_name
  from instructor
  ```

- Find all instructors in the Biology department
  ```sql
  select name
  from faculty
  where dept_name = 'Biology'
  ```

- Create a view of department salary totals
  ```sql
  create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum(salary)
  from instructor
  group by dept_name;
  ```

Views

- Is it different from DBMS’s side?
  - Yes; a view may or may not be *materialized*
  - Pros/Cons?

- Updates into views have to be treated differently
  - In most cases, disallowed.
**Views vs Tables**

<table>
<thead>
<tr>
<th>Creating</th>
<th>Create view V as (select * from A, B where …)</th>
<th>Create table T as (select * from A, B where …)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used</td>
<td>In any select query. Only some update queries.</td>
<td>It's a new table. You can do what you want.</td>
</tr>
<tr>
<td>Maintained as</td>
<td>1. Evaluate the query and store it on disk as if a table. 2. Don’t store. Substitute in queries when referenced.</td>
<td>It's a new table. Stored on disk.</td>
</tr>
<tr>
<td>What if a tuple inserted in A?</td>
<td>1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.</td>
<td>T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.</td>
</tr>
</tbody>
</table>

**Views vs Tables**

- Views strictly supercede “create a table and define a trigger to keep it updated”
- Two main reasons for using them:
  - Security/authorization
  - Ease of writing queries
- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change
Update of a View

- Add a new tuple to faculty
  
  \texttt{insert into faculty values ('30765', 'Green', 'Music');}

- This insertion must be represented by the insertion of the tuple
  \texttt{('30765', 'Green', 'Music', null)}
  into the \texttt{instructor} relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed, e.g.:
  - \texttt{create view instructor\_info as}
    \begin{verbatim}
    select ID, name, building
    from instructor, department
    where instructor.dept\_name= department.dept\_name;
    \end{verbatim}
  - \texttt{insert into instructor\_info values ('69987', 'White', 'Taylor');}

- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation