Announcements

• Assignment 2: Due on June 20\textsuperscript{th}
  • Late policy: 5\% penalty per 24 hours

• Project Milestone 1: Due on June 22\textsuperscript{nd}
  • Late policy: 25\% penalty per 24 hours

• Midterm: On June 26\textsuperscript{th}
  • Everything until lecture 10 (except lecture 6 on advance SQL)
Relational data model

• A database is a collection of relations (or tables)
• Each relation has a set of attributes (or columns)
• Each attribute has a unique name and a domain (or type)
  • The domains are required to be atomic

• Each relation contains a set of tuples (or rows)
  • Each tuple has a value for each attribute of the relation
  • Duplicate tuples are not allowed
    • Two tuples are duplicates if they agree on all attributes

☞ Simplicity is a virtue!
Types of integrity constraints

• Tuple-level
  • Domain restrictions, attribute comparisons, etc.
    • E.g. age cannot be negative
    • E.g. for flights table, arrival time > take off time

• Relation-level
  • Key constraints (focus in this lecture)
    • E.g. uid should be unique in the User relation
  • Functional dependencies (Textbook, Ch. 7)

• Database-level
  • Referential integrity – foreign key (focus in this lecture)
    • uid in Member must refer to a row in User with the same uid
Key (Candidate Key)

Def: A set of attributes $K$ for a relation $R$ if

- **Condition 1:** In no instance of $R$ will two different tuples agree on all attributes of $K$
  - That is, $K$ can serve as a “tuple identifier”
- **Condition 2:** No proper subset of $K$ satisfies the above condition
  - That is, $K$ is minimal

- Example: $User (uid, name, age, pop)$
  - $uid$ is a key of $User$
  - $age$ is not a key (not an identifier)
  - $\{uid, name\}$ is not a key (not minimal), but a superkey

- One candidate key is assigned to be primary key
Relational algebra

• A language for querying relational data based on “operators”
• Not used in commercial DBMSs (SQL)

- Core operators:
  • Selection, projection, cross product, union, difference, and renaming

- Additional, derived operators:
  • Join, natural join, intersection, etc.

- Compose operators to make complex queries

Output or intermediate result tables are transient
Operators can only be applied one row at a time

• You must be able to evaluate the condition over each single row of the input table!
  • Example: the most popular user

\[ \sigma_{\text{pop} \geq \text{every pop in User}} \text{ User} \]
Summary of operators

Core Operators
1. Selection: $\sigma_p R$
2. Projection: $\pi_L R$
3. Cross product: $R \times S$
4. Union: $R \cup S$
5. Difference: $R - S$
6. Renaming: $\rho_S(A_1 \rightarrow A_1', A_2 \rightarrow A_2', \ldots) R$

Derived Operators
1. Join: $R \bowtie_p S$
2. Natural join: $R \bowtie S$
3. Intersection: $R \cap S$

Note: Only use these operators for assignments & exams
Why do we need core operator $X$?

- Difference
  - The only non-monotone operator
- Projection
  - The only operator that removes columns
- Cross product
  - The only operator that adds columns
- Union
  - The only operator that adds rows
- Selection
  - The only operator that conditionally removes rows
Expression tree notation

IDs of users who belong to at least two groups

\[ \pi_{\text{uid}_1} \triangleleft_{\text{uid}_1 = \text{uid}_2 \land \text{gid}_1 \neq \text{gid}_2} \]

\[ \rho(\text{uid} \rightarrow \text{uid}_1, \text{gid} \rightarrow \text{gid}_1) \]

\[ \rho(\text{uid} \rightarrow \text{uid}_2, \text{gid} \rightarrow \text{gid}_2) \]

Member
A trickier example

• Who are the most popular?
  • Who do NOT have the highest $pop$ rating?
  • Whose $pop$ is lower than somebody else’s?

User ($uid$ int, $name$ string, $age$ int, $pop$ float)
Group ($gid$ string, $name$ string)
Member ($uid$ int, $gid$ string)
A trickier example

- Who are the most popular?
  - Who do NOT have the highest pop rating?
  - Whose pop is lower than somebody else’s?

A deeper question:
When (and why) is “—” needed?
Non-monotone operators

Add more rows to the input...

• If some old output rows may become invalid → the operator is non-monotone

• Example: difference operator $R - S$

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>901</td>
<td>edf</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>

= | uid | gid |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>

This old row becomes invalid because the new row added to $S$
Non-monotone operators

If some old output rows may become invalid (causing some row removal) \(\rightarrow\) the operator is non-monotone

Otherwise (old output rows always remain “correct”) \(\rightarrow\) the operator is monotone

\[
\begin{array}{cc}
\text{uid} & \text{gid} \\
123 & \text{gov} \\
857 & \text{abc} \\
189 & \text{abc}
\end{array}
\quad - \quad 
\begin{array}{cc}
\text{uid} & \text{gid} \\
123 & \text{gov} \\
901 & \text{edf}
\end{array}
\quad = \quad 
\begin{array}{cc}
\text{uid} & \text{gid} \\
857 & \text{abc} \\
189 & \text{abc}
\end{array}
\]

This old row is always valid no matter what rows are added to \( R \)
Classification of relational operators

- Selection: $\sigma_p R$  
  Monotone
- Projection: $\pi_L R$  
  Monotone
- Cross product: $R \times S$  
  Monotone
- Join: $R \bowtie_p S$  
  Monotone
- Natural join: $R \bowtie S$  
  Monotone
- Union: $R \cup S$  
  Monotone
- Difference: $R - S$  
  Monotone w.r.t. $R$; non-monotone w.r.t $S$
- Intersection: $R \cap S$  
  Monotone
SQL (lectures 3-6)
CREATE TABLE User (uid INT, name VARCHAR(30), age INT, pop DECIMAL(3,2));
CREATE TABLE Group (gid CHAR(10), name VARCHAR(100));
CREATE TABLE Member (uid INT, gid CHAR(10));

DROP TABLE User;
DROP TABLE Group;
DROP TABLE Member;

Drastic action: deletes ALL info about the table, not just the contents
Basic queries for DML: SFW statement

• SELECT $A_1, A_2, \ldots, A_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE condition;

• Also called an SPJ (select-project-join) query

• Corresponds to (but not really equivalent to) relational algebra query:
  \[ \pi_{A_1,A_2,\ldots,A_n}(\sigma_{\text{condition}}(R_1 \times R_2 \times \cdots \times R_m)) \]
## SQL set and bag operations

- **Set:** UNION, EXCEPT, INTERSECT
  - Exactly like set $\cup$, $-$, and $\cap$ in relational algebra
  - Duplicates in input tables, if any, are first eliminated
  - Duplicates in result are also eliminated (for UNION)

### Example

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
<th>(SELECT * FROM Bag1) UNION (SELECT * FROM Bag2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>fruit</td>
<td>fruit</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(SELECT * FROM Bag1) EXCEPT (SELECT * FROM Bag2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
</tr>
<tr>
<td>apple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(SELECT * FROM Bag1) INTERSECT (SELECT * FROM Bag2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
</tr>
<tr>
<td>orange</td>
</tr>
</tbody>
</table>
SQL set and bag operations

• Set: UNION, EXCEPT, INTERSECT
  • Exactly like set \( \cup, - \), and \( \cap \) in relational algebra

• Bag: UNION ALL, EXCEPT ALL, INTERSECT ALL
  • Think of each row as having an implicit count (the number of times it appears in the table)

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>fruit</td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>apple: 2</td>
<td>apple: 1</td>
</tr>
<tr>
<td>orange: 1</td>
<td>orange: 2</td>
</tr>
</tbody>
</table>

(SELECT * FROM Bag1) EXCEPT ALL (SELECT * FROM Bag2);

proper-subtract the two counts

apple: 1
orange: 0
Set versus bag operations

Poke (uid1, uid2, timestamp)
- uid1 poked uid2 at timestamp

Question: How do these two queries differ?

Q1: 
(SELECT uid1 FROM Poke) 
EXCEPT 
(SELECT uid2 FROM Poke);

Users who poked others but never got poked by others

Q2: 
(SELECT uid1 FROM Poke) 
EXCEPT ALL 
(SELECT uid2 FROM Poke);

Users who poked others more than others poked them
Table subqueries

• Use query result as a table
  • In set and bag operations, FROM clauses, etc.

• Example: names of users who poked others more than others poked them

```
SELECT DISTINCT name
FROM User,
    (SELECT uid1 as uid FROM Poke)
  EXCEPT ALL
    (SELECT uid2 as uid FROM Poke) AS T
WHERE User.uid = T.uid;
```
WITH clause

- The WITH clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs.

```sql
WITH max_pop AS (SELECT max(pop) AS popVal FROM user)
SELECT uid, name FROM user, max_pop WHERE user.pop = max_pop.popVal
```

- Supported by many but not all DBMSs
- Can be written using subqueries
IN subqueries

• $x$ IN *(subquery)* checks if $x$ is in the result of *subquery*

• Example: users at the same age as (some) Bart

```sql
SELECT *
FROM User,
WHERE age IN (SELECT age
FROM User
WHERE name = 'Bart');
```
EXISTS subqueries

• EXIST (subquery) checks if the result of subquery is non-empty

• Example: users at the same age as (some) Bart

```
SELECT *
FROM User AS u,
WHERE EXISTS (SELECT * FROM User
  WHERE name = 'Bart'
  AND age = u.age);
```

• This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries
Aggregates

• Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

• Example: number of users under 18, and their average popularity
  • COUNT(*) counts the number of rows

```
SELECT COUNT(*), AVG(pop) FROM User WHERE age < 18;
```

<table>
<thead>
<tr>
<th>COUNT (*)</th>
<th>AVG (pop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.625</td>
</tr>
</tbody>
</table>
Grouping

• SELECT ... FROM ... WHERE ...
  GROUP BY list_of_columns;

• Example: compute average popularity for each age group

  SELECT age, AVG(pop)
  FROM User
  GROUP BY age;
Example of computing GROUP BY

```
SELECT age, AVG(pop) FROM User GROUP BY age;
```

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Compute GROUP BY: group rows according to the values of GROUP BY columns

Compute SELECT for each group

<table>
<thead>
<tr>
<th>age</th>
<th>avg_pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.55</td>
</tr>
<tr>
<td>8</td>
<td>0.50</td>
</tr>
</tbody>
</table>
HAVING examples

• Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)

• List the average popularity for each age group with more than a hundred users

```
SELECT age, AVG(pop) 
FROM User 
GROUP BY age 
HAVING COUNT(*)>100;
```

• Can be written using WHERE and table subqueries

```
SELECT T.age, T.apop 
FROM (SELECT age, AVG(pop) AS apop, COUNT(*) AS gsize 
FROM User 
GROUP BY age) AS T 
WHERE T.gsize>100;
```
ORDER BY example

• List all users, sort them by **popularity (descending)** and **name (ascending)**

```
SELECT uid, name, age, pop
FROM User
ORDER BY pop DESC, name;
```

• **ASC** is the **default** option
• Strictly speaking, only **output** columns can appear in ORDER BY clause (although some DBMS support more)
Three-valued logic to handle NULL

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

- $x \text{ AND } y = \min(x, y)$
- $x \text{ OR } y = \max(x, y)$
- NOT $x = 1 - x$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \text{ AND } y$</th>
<th>$x \text{ OR } y$</th>
<th>NOT $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

- Comparing a NULL with another value (including another NULL) using $=,$ $>,$ etc., the result is NULL

- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - NULL is not enough

- Aggregate functions ignore NULL, except COUNT(*)
Unfortunate consequences

• Q1a = Q1b?

Q1a. SELECT AVG(pop) FROM User;

Q1b. SELECT SUM(pop)/COUNT(*) FROM User;

• Q2a = Q2b?

Q2a. SELECT * FROM User;

Q2b SELECT * FROM User WHERE pop=pop;

• Be careful: NULL breaks many equivalences
• Use **IS NULL** or **NOT NULL** for null comparisons
### Outerjoin examples

#### Group

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
</tr>
</tbody>
</table>

#### Member

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
<tr>
<td>857</td>
<td>gov</td>
</tr>
<tr>
<td>789</td>
<td>foo</td>
</tr>
</tbody>
</table>

#### Group \times Member

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
<td>NULL</td>
</tr>
<tr>
<td>foo</td>
<td>NULL</td>
<td>789</td>
</tr>
</tbody>
</table>

A **full outerjoin** between $R$ and $S$:

- All rows in the result of $R \bowtie S$, plus
- “Dangling” $R$ rows (those that do not join with any $S$ rows) padded with NULL’s for $S$’s columns
- “Dangling” $S$ rows (those that do not join with any $R$ rows) padded with NULL’s for $R$’s columns
# Outerjoin examples

**Group ⋈ Member**

- A left outerjoin ($R ⋈ S$) includes rows in $R ⋈ S$ plus dangling $R$ rows padded with NULL’s

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>nuk</td>
<td>United Nuclear Workers</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Group ⋈ Member**

- A right outerjoin ($R ⋈ S$) includes rows in $R ⋈ S$ plus dangling $S$ rows padded with NULL’s

<table>
<thead>
<tr>
<th>gid</th>
<th>name</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>Book Club</td>
<td>857</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>123</td>
</tr>
<tr>
<td>gov</td>
<td>Student Government</td>
<td>857</td>
</tr>
<tr>
<td>dps</td>
<td>Dead Putting Society</td>
<td>142</td>
</tr>
<tr>
<td>foo</td>
<td>NULL</td>
<td>789</td>
</tr>
</tbody>
</table>
Outerjoin syntax

SELECT * FROM Group LEFT OUTER JOIN Member
ON Group.gid = Member.gid;

SELECT * FROM Group RIGHT OUTER JOIN Member
ON Group.gid = Member.gid;

SELECT * FROM Group FULL OUTER JOIN Member
ON Group.gid = Member.gid;

A similar construct exists for regular ("inner") joins:

SELECT * FROM Group JOIN Member
ON Group.gid = Member.gid;

Theta join: gid is repeated

Natural join: gid appears once

For natural joins, add keyword NATURAL; don’t use ON

SELECT * FROM Group NATURAL JOIN Member;
Insert/Delete/Update

• Insert one row
  • User 789 joins Dead Putting Society
    
    ```sql
    INSERT INTO Member VALUES (789, 'dps');
    ```

• Delete **everything** from a table
  
  ```sql
  DELETE FROM Member;
  ```

• Delete according to a **WHERE** condition
  • Example: User 789 leaves Dead Putting Society
    
    ```sql
    DELETE FROM Member WHERE uid=789 AND gid='dps';
    ```

• Update: User 142 changes name to “Barney”
  
  ```sql
  UPDATE User
  SET name = 'Barney'
  WHERE uid = 142;
  ```
Types of SQL constraints

• NOT NULL
• Key
• Referential integrity (foreign key)
• General assertion
• Tuple- and attribute-based CHECK’s
NOT NULL & Key constraint examples

CREATE TABLE User
(uid INT NOT NULL,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL,
 age INT,
 pop DECIMAL(3,2));

CREATE TABLE User
(uid INT NOT NULL PRIMARY KEY,
 name VARCHAR(30) NOT NULL,
 twitterid VARCHAR(15) NOT NULL UNIQUE,
 age INT,
 pop DECIMAL(3,2));

CREATE TABLE Member
(uid INT NOT NULL,
 gid CHAR(10) NOT NULL,
 PRIMARY KEY(uid,gid));

At most one primary key per table
Any number of UNIQUE keys per table
This form is required for multi-attribute keys
Referential integrity in SQL

• Referenced column(s) must be **PRIMARY KEY**
• Referencing column(s) form a **FOREIGN KEY**
• Example

```sql
CREATE TABLE Member
(uid INT NOT NULL REFERENCES User(uid),
gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

Some system allow them to be non-PK but must be UNIQUE

```sql
CREATE TABLE MemberBenefits
(.....
FOREIGN KEY (uid,gid) REFERENCES Member(uid,gid));
```

This form is required for multi-attribute foreign keys
Enforcing referential integrity

Example: *Member.uid* references *User.uid*

- Delete or update a *User* row whose *uid* is referenced by some *Member* row
  - Reject or ON DELETE CASCADE or ON DELETE SET NULL

```
CREATE TABLE Member
  (uid INT NOT NULL REFERENCES User(uid)
   ON DELETE CASCADE,
   ...
  );
```

User

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td></td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td></td>
</tr>
<tr>
<td>789</td>
<td>Nelson</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Option 1: Reject

<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
<tr>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>

Member

```
CREATE TABLE Member
  (uid INT NOT NULL REFERENCES User(uid)
   ON DELETE CASCADE,
   ...
  );
```

Option 2: Cascade (ripple changes to all referring rows)
General assertion

- CREATE ASSERTION \textit{assertion\_name}
  CHECK \textit{assertion\_condition};

- \textit{assertion\_condition} is checked for each modification that could potentially violate it.

- Example: \textit{Member.uid} references \textit{User.uid}

  ```sql
  CREATE ASSERTION MemberUserRefIntegrity
  CHECK (NOT EXISTS
    (SELECT * FROM Member
     WHERE uid NOT IN
     (SELECT uid FROM User)));
  ```
Triggers

• A **trigger** is an event-condition-action (ECA) rule
  • When **event** occurs, test **condition**; if condition is satisfied, execute **action**

```sql
CREATE TRIGGER PickyPopGroup
AFTER UPDATE OF pop ON User
REFERENCING NEW ROW AS newUser
FOR EACH ROW
WHEN (newUser.pop < 0.5)
  AND (newUser.uid IN (SELECT uid
           FROM Member
           WHERE gid = 'popgroup'))
DELETE FROM Member
WHERE uid = newUser.uid AND gid = 'popgroup';
```
Trigger options

• Possible events include:
  - INSERT ON table; DELETE ON table; UPDATE [OF column] ON table

• Timing—action can be executed:
  - AFTER or BEFORE the triggering event
  - INSTEAD OF the triggering event on views (lecture 5)

• Granularity—trigger can be activated:
  - FOR EACH ROW modified
  - FOR EACH STATEMENT that performs modification
# Transition variables/tables

- **OLD ROW**: the modified row before the triggering event
- **NEW ROW**: the modified row after the triggering event
- **OLD TABLE**: a read-only table containing all old rows modified by the triggering event
- **NEW TABLE**: a table containing all modified rows after the triggering event

<table>
<thead>
<tr>
<th>Event</th>
<th>Row</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>old r; old t</td>
<td>old t</td>
</tr>
<tr>
<td>Insert</td>
<td>new r; new t</td>
<td>new t</td>
</tr>
<tr>
<td>Update</td>
<td>old/new r; old/new t</td>
<td>old/new t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Row</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>old/new r</td>
<td>-</td>
</tr>
<tr>
<td>Insert</td>
<td>new r</td>
<td>-</td>
</tr>
<tr>
<td>Delete</td>
<td>old r</td>
<td>-</td>
</tr>
</tbody>
</table>

AFTER Trigger | BEFORE Trigger
Certain triggers are only possible at statement level

CREATE TRIGGER MaintainAvgPop
AFTER UPDATE OF pop ON User
REFERENCING NEW TABLE AS newUsers
OLD TABLE AS oldUsers
FOR EACH STATEMENT
    WHEN (0.5 > (SELECT AVG(pop) FROM User))
    BEGIN
        DELETE FROM User WHERE uid IN (SELECT uid FROM newUsers)
        INSERT INTO User (SELECT * FROM oldUsers)
    END
Views

- A **view** is like a “virtual” table
  - Defined by a query, which describes **how to compute the view contents on the fly**
  - Stored as a query by DBMS instead of query contents
  - Can be used in queries just like a regular table

```sql
CREATE VIEW PopGroup AS
  SELECT *
  FROM User
  WHERE uid IN (SELECT uid
                FROM Member
                WHERE gid = 'popgroup');

SELECT AVG(pop) FROM PopGroup;
SELECT MIN(pop) FROM PopGroup;
SELECT ... FROM PopGroup;

DROP VIEW popGroup;
```
DB Design (lectures 7-10): E/R models Design theory
E/R basics

- **Entity**: a “thing,” like an object
- **Entity set**: a collection of things of the same type, like a relation of tuples or a class of objects
  - Represented as a rectangle
- **Relationship**: an association among entities
- **Relationship set**: a set of relationships of the same type (among same entity sets)
  - Represented as a diamond
- **Attributes**: properties of entities or relationships, like attributes of tuples or objects
  - Represented as ovals
General cardinality constraints

- General cardinality constraints determine **lower and upper** bounds on the number of relationships of a given relationship set in which a component entity may participate.

**Example:**

- **Student**
  - Takes
    - Course
  - (3,5)
  - (6,100)
Weak entity sets

• If entity E’s existence depends on entity F, then
  • F is a dominant entity
  • E is a subordinate entity
  • Example: Rooms inside Buildings are partly identified by Buildings’ name

• Weak entity set: containing subordinate entities
  • Drawn as a double rectangle
  • The relationship sets are called supporting relationship sets, drawn as double diamonds
  • A weak entity set must have a many-to-one or one-to-one relationship to a distinct entity set

• Strong entity set: containing no subordinate entities
Specialization or ISA relationships

- Similar to the idea of subclasses in object-oriented programming: subclass = special case, fewer entities, and possibly more properties
  - Represented as a triangle (direction is important)
- Example: paid users are users, but they also get avatars (yay!)

```
Users
  uid
  name
  ISA
  PaidUsers

Groups
  gid
  name

IsMemberOf
  fromDate

Automatically “inherits” key, attributes, relationships
Can participate in other relationships
```
Composite and multi-valued attributes

- Composite attributes: composed of fixed number of other attributes
  - E.g. Address
- Multi-valued attributes: attributes that are set-valued
  - e.g. Hobbies (double edges)
Translating entity sets

• An entity set translates directly to a table
  • Attributes → columns
  • Key attributes → key columns

User \((uid, \text{name})\)  

Group \((gid, \text{name})\)
Translating weak entity sets

• Remember the “borrowed” key attributes
• Watch out for attribute name conflicts
Translating double diamonds?

• No need to translate because the relationship is implicit in the weak entity set’s translation

Relationship

RoomInBuilding

(room_building_name, room_number,)

is subsumed by entity

Room (building_name, room_number, capacity)
Comparison of three approaches of translating subclasses & ISA

• Entity-in-all-superclasses
  • User (uid, name), PaidUser (uid, avatar)
  • Pro: All users are found in one table
  • Con: Attributes of paid users are scattered in different tables

• Entity-in-most-specific-class
  • User (uid, name), PaidUser (uid, name, avatar)
  • Pro: All attributes of paid users are found in one table
  • Con: Users are scattered in different tables

• All-entities-in-one-table
  • User (uid, [type, ]name, avatar)
  • Pro: Everything is in one table
  • Con: Lots of NULL’s; complicated if class hierarchy is complex
Translating composite and multi-valued attributes

Composite:  
Employee(eld,...,Street, City, Province,..)

Multi-valued:  
EmployeeHobbies(eld, hobby)

Foreign key: **eld references Employee**

Employee join EmployeeHobbies to get all info
Functional dependencies

• A **functional dependency (FD)** is a constraint between two sets of attributes in a relation

• FD has the form $X \rightarrow Y$, where $X$ and $Y$ are sets of attributes in a relation $R$

• $X \rightarrow Y$ means that whenever two tuples in $R$ agree on all the attributes in $X$, they must also agree on all attributes in $Y$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>?</td>
</tr>
</tbody>
</table>

Must be $b$  
Could be anything

• If $X$ is a superkey of $R$, then $X \rightarrow R$ (all the attributes)
Implied FDs: Armstrong’s Axioms

- A set of fds can imply other fds via 3 intuitive rules: Armstrong’s Axioms

1. Reflexivity: If \( Y \subseteq X \), then \( X \rightarrow Y \) (trivially)
   - iID, name \( \rightarrow \) iID
   - English: Each iID and name value determine a unique iID value

2. Augmentation: if \( X \rightarrow Y \), then \( XZ \rightarrow YZ \) (trivially)
   - If iID \( \rightarrow \) salary then iID, name \( \rightarrow \) salary, name
   - English: if each iID determines a unique salary value, then each (iID, name) value pair determines a unique (salary, name) value

<table>
<thead>
<tr>
<th>iID</th>
<th>name</th>
<th>salary</th>
<th>depName</th>
<th>bldng</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Alice</td>
<td>5000</td>
<td>CS</td>
<td>DC</td>
<td>20000</td>
</tr>
<tr>
<td>222</td>
<td>Bob</td>
<td>4000</td>
<td>Physics</td>
<td>PHY</td>
<td>30000</td>
</tr>
<tr>
<td>333</td>
<td>Carl</td>
<td>5200</td>
<td>CS</td>
<td>DC</td>
<td>20000</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
3. Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- Suppose each instructor can be in a single department and each dep has a single budget
- FD1: $iID \rightarrow depName$ FD2: $depName \rightarrow budget$, then $iID \rightarrow budget$
- English: If each $iID$ value determines a unique $depName$ value, which in turn determines a unique budget value, then each $iID$ value determines a unique budget value.

<table>
<thead>
<tr>
<th>InstDep</th>
</tr>
</thead>
<tbody>
<tr>
<td>$iID$</td>
</tr>
<tr>
<td>111</td>
</tr>
<tr>
<td>222</td>
</tr>
<tr>
<td>333</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Other Rules Implied by Armstrong’s Axioms

1. Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
   
   Proof:
   i. $X \rightarrow YZ$
   ii. $YZ \rightarrow Y$ (by reflexivity); $YZ \rightarrow Z$ (by reflexivity)
   iii. $X \rightarrow Y$ (by transitivity); $X \rightarrow Z$ (by transitivity)

2. Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$ (Prove as exercise)

3. Pseudo-transitivity: If $X \rightarrow Y$ and $YZ \rightarrow T$ then $XZ \rightarrow T$ (Prove as exercise)

Using these rules, you can prove or disprove a (derived) FD given a set of (base) FDs
Closure of FD sets: $\mathcal{F}^+$

• How do we know what additional FDs hold in a schema?

• A set of FDs $\mathcal{F}$ logically implies a FD $X \rightarrow Y$ if $X \rightarrow Y$ holds in all instances of $R$ that satisfy $\mathcal{F}$

• The closure of a FD set $\mathcal{F}$ (denoted $\mathcal{F}^+$):
  • The set of all FDs that are logically implied by $\mathcal{F}$
  • Informally, $\mathcal{F}^+$ includes all of the FDs in $\mathcal{F}$, i.e., $\mathcal{F} \subseteq F^+$, plus any dependencies they imply.
Attribute closure

• The closure of attributes $Z$ in a relation $R$ (denoted $Z^+$) with respect to a set of FDs, $\mathcal{F}$, is the set of all attributes $\{A_1, A_2, \ldots\}$ functionally determined by $Z$ (that is, $Z \rightarrow A_1 A_2 \ldots$)

• Algorithm for computing the closure

Compute$Z^+(Z, \mathcal{F})$:

• Start with closure $= Z$
• If $X \rightarrow Y$ is in $\mathcal{F}$ and $X$ is already in the closure, then also add $Y$ to the closure
• Repeat until no new attributes can be added
Example for computing attribute closure

Given relation $R(ABCDEFG)$

Compute $Z^+ (\{B,F\}, \mathcal{F})$: 

$\mathcal{F}$ includes:
- $A, B \rightarrow F$
- $A \rightarrow C$
- $B \rightarrow E, D$
- $D, F \rightarrow G$

<table>
<thead>
<tr>
<th>FD</th>
<th>$Z^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>$B, F$</td>
</tr>
<tr>
<td>$B \rightarrow E, D$</td>
<td>$B, F, E, D$</td>
</tr>
<tr>
<td>$D, F \rightarrow G$</td>
<td>$B, F, E, D, G$</td>
</tr>
</tbody>
</table>

$B, F \rightarrow E, D, G$
Using attribute closure

Given a relation \( R \) and set of FD’s \( \mathcal{F} \)

• Does another FD \( X \rightarrow Y \) follow from \( \mathcal{F} \)?
  • Compute \( X^+ \) with respect to \( \mathcal{F} \)
  • If \( Y \subseteq X^+ \), then \( X \rightarrow Y \) follows from \( \mathcal{F} \)

• Is \( K \) a key of \( R \)?
  • Compute \( K^+ \) with respect to \( \mathcal{F} \)
  • If \( K^+ \) contains all the attributes of \( R \), \( K \) is a super key
  • Still need to verify that \( K \) is minimal (how?)
    • Hint: check the attribute closure of its proper subset.
    • i.e., Check that for no set \( X \) formed by removing attributes from \( K \) is \( K^+ \) the set of all attributes
“Good” Schema Decomposition

- Lossless-join decompositions
  - We should be able to **construct the instance** of the original table from the instances of the tables in the decomposition

A decomposition \( \{R_1, R_2\} \) of \( R \) is **lossless** iff the common attributes of \( R_1 \) and \( R_2 \) form a superkey for either schema,

\[
R_1 \cap R_2 \rightarrow R_1 \text{ or } R_1 \cap R_2 \rightarrow R_2
\]

*If \( X \) is a superkey of \( R \), then \( X \rightarrow R \) (all the attributes) [last lecture]*
“Good” Schema Decomposition

- Lossless-join decompositions
- Dependency-preserving decompositions

Given a schema $R$ and a set of FDs $\mathcal{F}$, decomposition of $R$ is **dependency preserving** if there is an **equivalent set of FDs** $\mathcal{F}'$, none of which is **interrelational** in the decomposition.

- Next, how to obtain such decompositions?
  - BCNF $\rightarrow$ guaranteed to be a **lossless join** decomposition!
Boyce-Codd Normal Form (BCNF)

• A relation $R$ is in BCNF iff whenever $(X \rightarrow Y) \in \mathcal{F}^+$ and $XY \subseteq R$, then either
  • $(X \rightarrow Y)$ is trivial (i.e., $Y \subseteq X$), or
  • $X$ is a super key of $R$ (i.e., $X \rightarrow R$)
    • That is, all non-trivial FDs follow from “key → other attributes”

• Example: $R = \{\text{Sno, Sname, City, Pno, Pname, Price}\}$

\[\mathcal{F} \text{ includes: }\]

- **FD1**: $\text{Sno} \rightarrow \text{Sname}, \text{City}$
- **FD2**: $\text{Pno} \rightarrow \text{Pname}$
- **FD3**: $\text{Sno}, \text{Pno} \rightarrow \text{Price}$

• The schema is not in BCNF because, for example, Sno determines Sname, City, is non-trivial but is not a superkey of $R$
BCNF decomposition algorithm

• Find a BCNF violation
  • That is, a non-trivial FD $X \rightarrow Y$ in $\mathcal{F}^+$ of $R$ where $X$ is not a super key of $R$
    • Example: $R = \{\text{Sno, Sname, City, Pno, Pname, Price}\}$

$\mathcal{F}$ includes:

- **FD1**: Sno $\rightarrow$ Sname, City
- **FD2**: Pno $\rightarrow$ Pname
- **FD3**: Sno, Pno $\rightarrow$ Price

• Decompose $R$ into $R_1$ and $R_2$, where
  • $R_1$ has attributes $X \cup Y$;
  • $R_2$ has attributes $X \cup Z$, where $Z$ contains all attributes of $R$ that are in neither $X$ nor $Y$

• Repeat (till all are in BCNF)
BCNF decomposition example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)

\( \mathcal{F} \) includes:
- FD1: \( \text{Sno} \rightarrow \text{Sname, City} \)
- FD2: \( \text{Pno} \rightarrow \text{Pname} \)
- FD3: \( \text{Sno, Pno} \rightarrow \text{Price} \)

\( \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)

**BCNF violation:** \( \text{Sno} \rightarrow \text{Sname, City} \)

- \( R_2\{\text{Sno, Pno, Pname, Price}\} \)
- \( R_1\{\text{Sno, Sname, City}\} \)

- \( \text{Pno} \rightarrow \text{Pname} \)
- \( \text{Sno, Pno} \rightarrow \text{Price} \)

**BCNF violation:** \( \text{Pno} \rightarrow \text{Pname} \)

- \( R_{2b}\{\text{Sno, Pno, Price}\} \)
- \( R_{2a}\{\text{Pno, Pname}\} \)

BCNF:
- \( \text{Sno, Pno} \rightarrow \text{Price} \)
- \( \text{Pno} \rightarrow \text{Pname} \)

\( \{\text{Sno}\}^+ = \{\text{Sno, Sname, City}\} \rightarrow \) a superkey of \( R_1 \)
“Good” Schema Decomposition

• Lossless-join decompositions
• Dependency-preserving decompositions
• BCNF $\rightarrow$ guaranteed to be a lossless join decomposition!
  • Depend on the sequence of FDs for decomposition
  • Not necessarily dependency preserving

Example: consider $R=\{A, B, C\}$

$\mathcal{F}$ includes: $\text{FD}_1: AB \rightarrow C$ \hspace{1cm} $\text{FD}_2: C \rightarrow B$

BCNF violation: $C \rightarrow B$

$\{A, C\}$ \hspace{2cm} $\{C, B\}$

$AB \rightarrow C$ is interrelational and cannot be tested directly
“Good” Schema Decomposition

- Lossless-join decompositions
- Dependency-preserving decompositions
- BCNF \(\rightarrow\) guaranteed to be a lossless join decomposition!
  - Depend on the sequence of FDs for decomposition
  - Not necessarily dependency preserving
- 3NF \(\rightarrow\) both lossless join and dependency preserving
Third normal form (3NF)

- A relation $R$ is in 3NF iff whenever $(X \rightarrow Y) \in F^+$ and $XY \subseteq R$, then either
  - $(X \rightarrow Y)$ is trivial (i.e., $Y \subseteq X$), or
  - $X$ is a super key of $R$ (i.e., $X \rightarrow R$) or
  - Each attribute in $Y - X$ is contained in a candidate key of $R$

- Example: consider $R=\{A, B, C\}$
  - Satisfies 3NF, but not BCNF

- 3NF is looser than BCNF $\Rightarrow$ Allows more redundancy

$F$ includes: FD1: $AB \rightarrow C$  
FD2: $C \rightarrow B$  

$\{B\} - \{C\} = \{B\}$ is part of the key $\{AB\}$
Finding minimal cover

• A minimal cover for $\mathcal{F}$ can be computed in 3 steps.
  1. Replace $X \rightarrow YZ$ with the pair $X \rightarrow Y$ and $X \rightarrow Z$
  2. Remove $A$ from the left-hand side of $X \rightarrow B$ in $\mathcal{F}$ if $B \in computeX^+(X - \{A\}, \mathcal{F})$
  3. Remove $X \rightarrow A$ from $\mathcal{F}$ if $A \in computeX^+(X, \mathcal{F} - \{X \rightarrow A\})$
    • Note that each step must be repeated until it no longer succeeds in updating $\mathcal{F}$.

• Example: $R = \{\text{Sno,Sname,City,Pno,Pname,Price, PType} \}$

$\mathcal{F}$: FD1: Sno $\rightarrow$ Sname, City
FD2: Pno $\rightarrow$ Pname
FD3: Sno, Pno $\rightarrow$ Price
FD4: Sno, Pname $\rightarrow$ Price
FD5: Pno, Pname $\rightarrow$ Ptype

Remove FD3

Sno $\rightarrow$ Sname, Sno $\rightarrow$ City

Pno $\rightarrow$ Ptype
Computing 3NF decomposition

Efficient algorithm for computing a 3NF decomposition of $R$ with FDs $\mathcal{F}$:

1. Initialize the decomposition with empty set
2. Find a minimal cover for $\mathcal{F}$, let it be $\mathcal{F}^*$
3. For every $(X \rightarrow Y) \in \mathcal{F}^*$, add a relation $\{XY\}$ to the decomposition
4. If no relation contains a candidate key for $R$, then compute a candidate key $K$ for $R$, and add relation $\{K\}$ to the decomposition.
Example for 3NF decomposition

• $R = \{\text{Sno}, \text{Sname}, \text{City}, \text{Pno}, \text{Pname}, \text{Price}\}$

\[\mathcal{F}: \text{FD1: } \text{Sno} \rightarrow \text{Sname}, \text{City} \]
\[\text{FD2: } \text{Pno} \rightarrow \text{Pname} \]
\[\text{FD3: } \text{Sno}, \text{Pno} \rightarrow \text{Price} \]
\[\text{FD4: } \text{Sno}, \text{Pname} \rightarrow \text{Price} \]

• Minimal cover $\mathcal{F}^*$

\[\mathcal{F}^*: \text{FD1a: } \text{Sno} \rightarrow \text{Sname} \]
\[\text{FD1b: } \text{Sno} \rightarrow \text{City} \]
\[\text{FD2: } \text{Pno} \rightarrow \text{Pname} \]
\[\text{FD4: } \text{Sno}, \text{Pname} \rightarrow \text{Price} \]

• Add relation for candidate key

• Optimization for this example: combine relations R1a and R1b
Next lecture

• DB Architecture Overview & Physical Data Organization