Review Lectures 2-4

Introduction to Database Management

CS348 Fall 2022
Announcements (Thur. Sep 22)

• Project milestone 0 due by Sep 27 (Tue), 11:59pm
  • Form a team on Learn
  • Report.pdf and link to GitHub repo
  • Not graded, but very important!

• Assignment 1 due by Sep 29 (Thur), 11:59pm
  • Part 1: general questions and r.a.
    • Submit via Crowdmark
  • Part 2: writing SQL on DB2 on school servers (try soon)
    • Submit via Marmoset
Outline

• Lecture 2: Intro to the relational model
  • Relational data model
  • Relational algebra

• Lectures 3 & 4: SQL (1) & (2)
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 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!
Example for Relational data model

**User**

<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>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
</tbody>
</table>

**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>
</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>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
</tbody>
</table>

**tuples (or rows)**

Duplicates are not allowed

Ordering of rows doesn’t matter (even though output is always in some order)
Schema vs. instance

• **Schema (metadata)**
  • Specifies the **logical structure** of data
  • Is defined at setup time, rarely changes

  \[
  \text{**User** (} uid \text{ int, } name \text{ string, } age \text{ int, } pop \text{ float)}
  \]
  \[
  \text{**Group** (} gid \text{ string, } name \text{ string)}
  \]
  \[
  \text{**Member** (} uid \text{ int, } gid \text{ string)}
  \]

• **Instance**
  • Represents the data content
  • Changes rapidly, but always **conforms** to the schema

  \[
  \text{**User**: } \{ (142, \text{ Bart}, 10, 0.9), (857, \text{ Milhouse}, 10, 0.2), \ldots \}
  \]
  \[
  \text{**Group**: } \{ (\text{abc}, \text{ Book Club}), (\text{gov}, \text{ Student Government}), \ldots \}
  \]
  \[
  \text{**Member**: } \{ (142, \text{ dps}), (123, \text{ gov}), \ldots \}
  \]
Types of integrity constraints

• Tuple-level
  • Domain restrictions, attribute comparisons, etc.
    • E.g. age cannot be negative

• 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

Only Check Condition 1
More examples of keys

- **Member** *(uid, gid)*
  - {uid, gid}
  - A key can contain multiple attributes

- **Address** *(street_address, city, state, zip)*
  - Key 1: `{street_address, city, state}`
  - Key 2: `{street_address, zip}`
  - A relation can have multiple keys!

- **Primary key**: a designated candidate key in the schema declaration
  - **Underline** all its attributes, e.g., Address *(street_address, city, state, zip)*
“Pointers” to other rows

- **Foreign key**: primary key of one relation appearing as attribute of another relation

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

<table>
<thead>
<tr>
<th><strong>gid</strong></th>
<th><strong>name</strong></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>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>uid</strong></th>
<th><strong>gid</strong></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>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
</tbody>
</table>
“Pointers” to other rows

- **Referential integrity**: A tuple with a non-null value for a foreign key that does not match the primary key value of a tuple in the referenced relation is not allowed.

<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>
</tbody>
</table>

<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>Xi</td>
</tr>
<tr>
<td>857</td>
<td>gov</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>
Outline

• Lecture 2: Intro to the relational model
  • Relational data model
  • Relational algebra

• Lectures 3 & 4: SQL (1) & (2)
Relational algebra

A language for querying relational data based on “operators”

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

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

• Compose operators to make complex queries
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: $\rho_S(A_1 \rightarrow A'_1, A_2 \rightarrow A'_2, \ldots) R$
   - Does not really add “processing” power

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 & quiz
More on selection

• Selection condition can include any column of $R$, constants, comparison ($\leq$, etc.) and Boolean connectives ($\wedge$: and, $\lor$: or, $\neg$: not)
  • Example: users with popularity at least 0.9 and age under 10 or above 12

\[
\sigma_{\text{pop} \geq 0.9 \land (\text{age} < 10 \lor \text{age} > 12)} \text{User}
\]

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

\[
\sigma_{\text{pop} \geq \text{every pop in User}} \text{User} \text{ WRONG!}
\]
More on projection

• Duplicate output rows are removed (by definition)
  • Example: user ages

\[ \pi_{age} \text{User} \]

<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>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \pi_{age} \]

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

...
Core operator 3: Cross product

• Input: two tables $R$ and $S$
• Natation: $R \times S$
• Purpose: pairs rows from two tables
• Output: for each row $r$ in $R$ and each $s$ in $S$, output a row $rs$ (concatenation of $r$ and $s$)
Derived operator 1: Join

- Info about users, plus IDs of their groups

\[ \text{User} \bowtie_{\text{User.uid} = \text{Member.uid}} \text{Member} \]

Prefix a column reference with table name and "." to disambiguate identically named columns from different tables.
Derived operator 2: Natural join

\[ User \bowtie Member = \pi_{uid,name,age,pop,gid} \left( User \bowtie_{User.uid=Member.uid} Member \right) \]
Core operator 4: Union

• Input: two tables $R$ and $S$
• Notation: $R \cup S$
  • $R$ and $S$ must have identical schema
• Output:
  • Has the same schema as $R$ and $S$
  • Contains all rows in $R$ and all rows in $S$ (with duplicate rows removed)

<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>

$\cup$

<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>
</tbody>
</table>

=  

<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>
<tr>
<td>901</td>
<td>edf</td>
</tr>
</tbody>
</table>
Core operator 5: Difference

• Input: two tables $R$ and $S$
• Notation: $R - S$
  • $R$ and $S$ must have identical schema
• Output:
  • Has the same schema as $R$ and $S$
  • Contains all rows in $R$ that are not in $S$

<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>
</tbody>
</table>

= 

<table>
<thead>
<tr>
<th>$uid$</th>
<th>$gid$</th>
</tr>
</thead>
<tbody>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>
Derived operator 3: Intersection

• Input: two tables $R$ and $S$
• Notation: $R \cap S$
  • $R$ and $S$ must have identical schema
• Output:
  • Has the same schema as $R$ and $S$
  • Contains all rows that are in both $R$ and $S$
• Shorthand for $R - (R - S)$
• Also equivalent to $S - (S - R)$
• And to $R \bowtie S$
Core operator 6: Renaming

• Input: a table \( R \)
• Notation: \( \rho_s \ R \), \( \rho(A_1 \rightarrow A'_1, \ldots) \ R \), or \( \rho_s(A_1 \rightarrow A'_1, \ldots) \ R \)
• Purpose: “rename” a table and/or its columns
• Output: a table with the same rows as \( R \), but called differently

\[
\rho_{M1}(uid \rightarrow uid_1, gid \rightarrow gid_1) \text{Member}
\]

<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>uid1</th>
<th>gid1</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>gov</td>
</tr>
<tr>
<td>857</td>
<td>abc</td>
</tr>
</tbody>
</table>
9. Basic operator: Renaming

- IDs of users who belong to **at least two groups**

```
<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>gov</td>
</tr>
<tr>
<td>100</td>
<td>abc</td>
</tr>
<tr>
<td>200</td>
<td>gov</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>uid</th>
<th>gid</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>gov</td>
</tr>
<tr>
<td>100</td>
<td>abc</td>
</tr>
<tr>
<td>200</td>
<td>gov</td>
</tr>
</tbody>
</table>
```

**Member ▶, Member**

**Condition 1: same uid**

**Condition 2: different gids**
Renaming example

• IDs of users who belong to at least two groups

\[ \text{Member} \bowtie ? \text{Member} \]

\[ \pi_{\text{uid}} \left( \text{Member} \bowtie \text{Member.uid=Member.uid } \land \text{Member.gid}\neq \text{Member.gid} \right) \]

\[ \pi_{\text{uid}_1} \left( \rho(\text{uid}\rightarrow\text{uid}_1,\text{gid}\rightarrow\text{gid}_1) \text{Member} \right) \]

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

WRONG!
Expression tree notation

\[
\begin{aligned}
\pi_{uid_1} & \quad \bowtie_{uid_1 = uid_2 \land gid_1 \neq gid_2} \\
\rho(uid \rightarrow uid_1, gid \rightarrow gid_1) & \quad \rho(uid \rightarrow uid_2, gid \rightarrow gid_2)
\end{aligned}
\]
Take-home Exercises

- Exercise 1: IDs of groups who have at least 2 users?
- Exercise 2: IDs of users who belong to at least three groups?

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?

\[
\begin{align*}
\pi_{\text{uid}} & \quad \pi_{\text{User}_1.\text{uid}} \\
\text{User} & \quad \bowtie_{\text{User}_1.\text{pop} < \text{User}_2.\text{pop}} \\
\rho_{\text{User}_1} & \quad \rho_{\text{User}_2} \\
\text{User} & \quad \text{User}
\end{align*}
\]

A deeper question:
When (and why) is “—” needed?

User (\text{uid} int, \text{name} string, \text{age} int, \text{pop} float)
Group (\text{gid} string, \text{name} string)
Member (\text{uid} int, \text{gid} string)
Non-monotone operators

• If some old output rows may become invalid, and need to be removed → 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>
```
```
<table>
<thead>
<tr>
<th>uid</th>
<th>gid</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 $S$
Classification of relational operators

- Selection: $\sigma_p R$  
  Monotone
- Projection: $\pi_L R$  
  Monotone
- Cross product: $R \times S$  
  Monotone
- Join: $R \bowt \rho S$  
  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
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
  - ?
- Selection
  - ?
Extensions to relational algebra

• Duplicate handling (“bag algebra”)
• Grouping and aggregation
• “Extension” (or “extended projection”) to allow new column values to be computed

🚀 All these will come up when we talk about SQL
🚀 But for now we will stick to standard relational algebra without these extensions
Outline

• Lecture 2: Intro to the relational model
  • Relational data model
  • Relational algebra

• Lectures 3 & 4: SQL (1) & (2)
  • Data-definition language (DDL): define/modify schemas, delete relations
  • Data-manipulation language (DML): query information, and insert/delete/modify tuples
  • Integrity constraints: specify constraints that the data stored in the database must satisfy
• **CREATE TABLE** `table_name` 
  (...,, `column_name column_type`, ...);

  ```sql
  CREATE TABLE User(uid DECIMAL(3,0), name VARCHAR(30), age DECIMAL (2,0), pop DECIMAL(3,2));
  CREATE TABLE Group (gid CHAR(10), name VARCHAR(100));
  CREATE TABLE Member (uid DECIMAL (3,0), gid CHAR(10));
  ```

• **DROP TABLE** `table_name`;

  ```sql
  DROP TABLE User;
  DROP TABLE Group;
  DROP TABLE Member;
  ```

  -- everything from -- to the end of line is ignored.
  -- SQL is insensitive to white space.
  -- SQL is insensitive to case (e.g., ...CREATE... is
  -- equivalent to ...create...).
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:

  \[ \pi_{A_1, A_2, \ldots, A_n}(\sigma_{\text{condition}}(R_1 \times R_2 \times \cdots \times R_m)) \]
Forcing set semantics

• ID’s of all pairs of users that belong to one group

```
SELECT m1.uid AS uid1, m2.uid AS uid2
    FROM Member AS m1, Member AS m2
    WHERE m1.gid = m2.gid
        AND m1.uid > m2.uid;
```

→ Say Lisa and Ralph are in both the book club and the student government, they id pairs will appear twice

• Remove duplicate (uid1, uid2) pairs from the output

```
SELECT DISTINCT m1.uid AS uid1, m2.uid AS uid2
    FROM Member AS m1, Member AS m2
    WHERE m1.gid = m2.gid;
        AND m1.uid > m2.uid;
```
Semantics of SFW

- SELECT [DISTINCT] $E_1, E_2, \ldots, E_n$
  FROM $R_1, R_2, \ldots, R_m$
  WHERE condition;

- For each $t_1$ in $R_1$:
  For each $t_2$ in $R_2$: ... ...
  For each $t_m$ in $R_m$:
    If condition is true over $t_1, t_2, \ldots, t_m$:
      Compute and output $E_1, E_2, \ldots, E_n$ as a row

If DISTINCT is present
  Eliminate duplicate rows in output

- $t_1, t_2, \ldots, t_m$ are often called tuple variables
SQL set and bag operations

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

```
(SELECT * FROM Bag1)
UNION ALL
(SELECT * FROM Bag2);
```

Sum up the counts from two tables

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
<th>fruit</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td>apple</td>
<td>apple</td>
</tr>
<tr>
<td>apple</td>
<td>orange</td>
<td>apple</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
<td>orange</td>
</tr>
<tr>
<td>apple: 2</td>
<td>apple: 1</td>
<td>apple: 3</td>
</tr>
<tr>
<td>orange:1</td>
<td>orange:2</td>
<td>orange:3</td>
</tr>
</tbody>
</table>
SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations

Next: how to nest SQL queries
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

```sql
SELECT DISTINCT name
FROM User,
  (SELECT uid1 FROM Poke)
EXCEPT ALL
  (SELECT uid2 FROM Poke) AS T
WHERE User.uid = T.uid;
```
Scalar subqueries

• A query that returns a single row can be used as a value in WHERE, SELECT, etc.

• Example: users at the same age as Bart

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

• When can this query go wrong?
  • Return more than 1 row
  • Return no rows
IN subqueries

• \(x \text{ IN } (\text{subquery})\) checks if \(x\) is in the result of subquery

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

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

- **EXISTS** *(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
Quantified subqueries

• Universal quantification (for all):
  • ... WHERE $x \text{ op } \text{ALL}(\text{subquery})$ ...
  • True iff for all $t$ in the result of subquery, $x \text{ op } t$

```
SELECT * 
FROM User 
WHERE pop >= \text{ALL}(\text{SELECT pop FROM User});
```

• Existential quantification (exists):
  • ... WHERE $x \text{ op } \text{ANY}(\text{subquery})$ ...
  • True iff there exists some $t$ in subquery result s.t. $x \text{ op } t$

```
SELECT * 
FROM User 
WHERE \text{NOT} (pop < \text{ANY}(\text{SELECT pop FROM User});
```
More ways to get the most popular

• Which users are the most popular?

Q1. SELECT *
    FROM User
    WHERE pop >= ALL(SELECT pop FROM User);

Q2. SELECT *
    FROM User
    WHERE NOT
    (pop < ANY(SELECT pop FROM User);

Q3. SELECT *
    FROM User AS u
    WHERE NOT [EXITS or IN?] (SELECT * FROM User
    WHERE pop > u.pop);

Q4. SELECT * FROM User
    WHERE uid NOT [EXISTS or IN?] (SELECT u1.uid
    FROM User AS u1, User AS u2
    WHERE u1.pop < u2.pop);
SQL features covered so far

• SELECT-FROM-WHERE statements
• Set and bag operations
• Subqueries
  • Subqueries allow queries to be written in more declarative ways (recall the “most popular” query)
  • But in many cases, they don’t add expressive power

Next: aggregation and grouping
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

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

• Example: How many users are in some group?

```
SELECT COUNT(*)
FROM (SELECT DISTINCT uid FROM Member);
```

Is equivalent to

```
SELECT COUNT(DISTINCT uid)
FROM Member;
```
Example of computing GROUP BY

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

Compute SELECT for each group

<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>

<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>
Aggregates with no GROUP BY

• An aggregate query with no GROUP BY clause = all rows go into one group

SELECT AVG(pop) FROM User;

<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>

<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>

Group all rows into one group

Aggregate over the whole group

avg_pop

0.525
Restriction on SELECT

• If a query uses aggregation/group by, then every column referenced in SELECT must be either
  • Aggregated, or
  • A GROUP BY column

Why?

This restriction ensures that any SELECT expression produces only one value for each group

```sql
SELECT uid, age FROM User GROUP BY age;
WRONG!
```

```sql
SELECT uid, MAX(pop) FROM User;
WRONG!
```
HAVING examples

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

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

• Can be written using WHERE and table subqueries

```sql
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)
- Can use sequence numbers instead of names to refer to output columns: **ORDER BY 4 DESC, 2;**
Outline

• Lecture 2: Intro to the relational model

• Lectures 3 & 4: SQL (1) & (2)
  • Data-definition language (DDL)
  • Data-manipulation language (DML)
    • SELECT-FROM-WHERE statements
    • Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
    • Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
    • Aggregation and grouping (GROUP BY, HAVING)
    • Ordering (ORDER)
    • Outerjoins (and Nulls)
    • INSERT/DELETE/UPDATE

• Integrity constraints: specify constraints that the data stored in the database must satisfy
Incomplete information

• Example: User \((uid, name, age, pop)\)

• Value unknown
  • We do not know Nelson’s age

• Value not applicable
  • Suppose \(pop\) is based on interactions with others on our social networking site
  • Nelson is new to our site; what is his \(pop\)?
SQL’s solution

• A special value **NULL**
  • For every domain
  • Special rules for dealing with NULL’s

• Example: *User (uid, name, age, pop)*
  • \(789, \text{“Nelson”}, \text{NULL, NULL}\)
Three-valued logic

TRUE = 1, FALSE = 0, UNKNOWN = 0.5

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

• 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
Another problem

• Example: Who has NULL pop values?

SELECT * FROM User WHERE pop = NULL;

(SELECT * FROM User) EXCEPT ALL
(SELECT * FROM USER WHERE pop=pop);

• SQL introduced special, built-in predicates **IS NULL** and **IS NOT NULL**

SELECT * FROM User WHERE pop IS NULL;
**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 ⋈ 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 \(R \bowtie S\) 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 \(R \bowtie S\) 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;

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

SELECT * FROM Group NATURAL JOIN Member;
SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
- Aggregation and grouping
- Ordering
- NULL’s and outerjoins

Next: data modification statements, constraints
INSERT

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

• Insert the result of a query
  • Everybody joins Dead Putting Society!
    ```sql
    INSERT INTO Member
    (SELECT uid, 'dps' FROM User
    WHERE uid NOT IN (SELECT uid
      FROM Member
    WHERE gid = 'dps'));
    ```
DELETE

• 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';
    ```
  • Example: Users under age 18 must be removed from United Nuclear Workers
    ```sql
    DELETE FROM Member
    WHERE uid IN (SELECT uid FROM User WHERE age < 18) AND gid = 'nuk';
    ```
**UPDATE**

- Example: User 142 changes name to “Barney”

```
UPDATE User
SET name = 'Barney'
WHERE uid = 142;
```

- Example: We are all popular!

```
UPDATE User
SET pop = (SELECT AVG(pop) FROM User);
```

- But won’t update of every row causes average pop to change?

☞ Subquery is always computed over the old table
Outline

• Lecture 2: Intro to the relational model

• Lectures 3 & 4: SQL (1) & (2)
  • Data-definition language (DDL)
  • Data-manipulation language (DML)
    • SELECT-FROM-WHERE statements
    • Set/bag (DISTINCT, UNION/EXCEPT/INTERSECT (ALL))
    • Subqueries (table, scalar, IN, EXISTS, ALL, ANY)
    • Aggregation and grouping (GROUP BY, HAVING)
    • Ordering (ORDER)
    • Outerjoins (and Nulls)
    • INSERT/DELETE/UPDATE

• **Integrity constraints**: specify constraints that the data stored in the database must satisfy
Types of SQL constraints

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

CREATE TABLE User
(uid DECIMAL(3,0) NOT NULL,
name VARCHAR(30) NOT NULL,
twitterid VARCHAR(15) NOT NULL,
age DECIMAL(2,0),
pop DECIMAL(3,2));

CREATE TABLE Group
(gid CHAR(10) NOT NULL,
name VARCHAR(100) NOT NULL);

CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL,
gid CHAR(10) NOT NULL);
Key declaration examples

CREATE TABLE User
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY, name VARCHAR(30) NOT NULL, twitterid VARCHAR(15) NOT NULL UNIQUE, age DECIMAL (2,0), pop DECIMAL(3,2));

CREATE TABLE Group
(gid CHAR(10) NOT NULL PRIMARY KEY, name VARCHAR(100) NOT NULL);

CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL PRIMARY KEY, gid CHAR(10) NOT NULL, PRIMARY KEY(uid,gid));

This form is required for multi-attribute keys

At most one primary key per table

Any number of UNIQUE keys per table

Incorrect!
Referential integrity example

- If an *uid* appears in *Member*, it must appear in *User*
  - *Member.uid* references *User.uid*
- If a *gid* appears in *Member*, it must appear in *Group*
  - *Member.gid* references *Group.gid*

☞ That is, no “dangling pointers”

<table>
<thead>
<tr>
<th>User</th>
<th>Member</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>uid</strong></td>
<td><strong>name</strong></td>
<td><strong>...</strong></td>
</tr>
<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>
Referential integrity in SQL

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

```sql
CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL REFERENCES User(uid),
gid CHAR(10) NOT NULL,
PRIMARY KEY(uid,gid),
FOREIGN KEY (gid) REFERENCES Group(gid));
```

This form is required for multi-attribute foreign keys

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

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

- Insert or update a *Member* row so it refers to a non-existent *uid*
  - Reject

<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>

<table>
<thead>
<tr>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>142</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>857</td>
</tr>
<tr>
<td>857</td>
</tr>
<tr>
<td>456</td>
</tr>
<tr>
<td>456</td>
</tr>
</tbody>
</table>

| 000 | gov |

Reject
Enforcing referential integrity

Example: Member.uid references User.uid
• Delete or update a User row whose uid is referenced by some Member row
  • Multiple Options (in SQL)

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

CREATE TABLE Member
(uid DECIMAL(3,0) NOT NULL REFERENCES User(uid)
ON DELETE CASCADE,
......);

Option 1: Reject

Option 2: Cascade (ripple changes to all referring rows)
Enforcing referential integrity

Example: *Member.uid references User.uid*

• Delete or update a *User* row whose *uid* is referenced by some *Member* row
  • Multiple Options (in SQL)

```
CREATE TABLE Member
    (uid DECIMAL(3,0) NOT NULL REFERENCES User(uid)
     ON DELETE SET NULL,
     .....);
```

**Option 3: Set NULL**
(set all references to NULL)
General assertion

• **CREATE ASSERTION** `assertion_name` 
  **CHECK** `assertion_condition`;

• `assertion_condition` is checked for each modification that could potentially violate it

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

```sql
CREATE ASSERTION MemberUserRefIntegrity 
CHECK (NOT EXISTS 
  (SELECT * FROM Member 
  WHERE uid NOT IN 
  (SELECT uid FROM User)));
```
Tuple- and attribute-based CHECK’s

• Associated with a single table
• Only checked when a tuple/attribute is inserted/updated
  • Reject if condition evaluates to FALSE
  • TRUE and UNKNOWN are fine
• Examples:

```sql
CREATE TABLE User(...
age INTEGER CHECK(age IS NULL OR age > 0),
...);
```

```sql
CREATE TABLE Member(
  uid INTEGER NOT NULL,
  CHECK(uid IN (SELECT uid FROM User)),
...);
```

Exercise Question: How does it differ from a referential integrity constraint (slides 26-27)?
SQL features covered so far

• Query
  • SELECT-FROM-WHERE statements
  • Set and bag operations
  • Table expressions, subqueries
  • Aggregation and grouping
  • Ordering
  • Outerjoins (and NULL)

• Modification
  • INSERT/DELETE/UPDATE

• Constraints

Next: triggers, views, indexes (lecture 5)