Announcements

• Assignment 1 is released: Due May 30th

• Project description is released
  • Milestone 0: not graded but due on May 25th

• No class next Tuesday, May 23rd (Monday schedule)
Basic SQL features

• Query
  • 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)

• Modification
  • INSERT/DELETE/UPDATE

• Constraints

Lecture 4
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 \(pop\)?
Solution 1

• Dedicate a value from each domain (type)
  • \textit{pop} cannot be $-1$, so use $-1$ as a special value to indicate a missing or invalid \textit{pop}

\begin{verbatim}
SELECT AVG(pop) FROM User;
SELECT AVG(pop) FROM User WHERE pop <> -1;
\end{verbatim}

• Perhaps the value is not as special as you think!
  • the Y2K bug

Incorrect answers
Complicated

http://www.90s411.com/images/y2k-cartoon.jpg
Solution 2

• A valid-bit for every column
  • User (uid, name, name_is_valid, age, age_is_valid, pop, pop_is_valid)

SELECT AVG(pop) FROM User WHERE pop_is_valid;

• Complicates schema and queries
  • Need almost double the number of columns
Solution 3

- Decompose the table; missing row = missing value
  - **UserName** (uid, name) ➔ Has a tuple for Nelson
  - **UserAge** (uid, age) ➔ No entry for Nelson
  - **UserPop** (uid, pop) ➔ No entry for Nelson
  - **UserID** (uid) ➔ Has a tuple for Nelson

- Conceptually the cleanest solution
- Still complicates schema and queries
  - How to get all information about users in a table?
  - Natural join doesn’t work!
SQL’s solution

• A special value **NULL**
  • For every domain (i.e., any datatype)
  • Special rules for dealing with NULL’s

• Example: *User (uid, name, age, pop)*
  • ⟨789, “Nelson”, NULL, NULL⟩

Truth table?
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 \( \text{AVG}(\text{pop}) \) FROM User;

Q1b. SELECT \( \text{SUM}(\text{pop})/\text{COUNT}(\*) \) 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
(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;
Need for a new join query

• Example: construct a master group membership list with all groups and its members info

```
SELECT g.gid, g.name AS gname, u.uid, u.name AS uname
FROM Group g, Member m, User u
WHERE g.gid = m.gid AND m.uid = u.uid;
```

• What if a group is empty?

• It may be reasonable for the master list to include empty groups as well
  • For these groups, uid and uname columns would be NULL
Outerjoin examples

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

<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>

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

**Group**
- gid: abc, gov, dps, nuk
- name: Book Club, Student Government, Dead Putting Society, United Nuclear Workers

**Member**
- uid: 142, 123, 857, 789
**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**

- 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

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;
```
SQL features covered so far

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

Next: data modification statements, constraints
INSERT

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

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

INSERT INTO User (uid, name) VALUES (389, 'Marge');
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”

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

• Example: We are all popular!

```sql
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
Constraints

• Restricts what data is allowed in a database
  • In addition to the simple structure and type restrictions imposed by the table definitions

• Why use constraints?
  • Protect data integrity (catch errors)
  • Tell the DBMS about the data (so it can optimize better)

• Declared as part of the schema and enforced by the DBMS
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 INT NOT NULL,
name VARCHAR(30) NOT NULL,
twitterid VARCHAR(15) NOT NULL,
age INT,
pop DECIMAL(3,2));

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

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

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

CREATE TABLE Member
(uid INT NOT NULL PRIMARY KEY,
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

Incorrect!
Referential integrity example

- If a *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”
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

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>User</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>uid</em></td>
<td><em>name</em></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>

<table>
<thead>
<tr>
<th>Member</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>uid</em></td>
<td><em>gid</em></td>
</tr>
<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>
<tr>
<td>000</td>
<td>gov</td>
</tr>
</tbody>
</table>
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](Image)

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

**Option 1: Reject**

- *uid* 142: Bart
- *uid* 123: Milhouse
- *uid* 857: Lisa
- *uid* 456: Ralph
- *uid* 789: Nelson
- ... (Other rows)

**Option 2: Cascade**

- *uid* 142: dps
- *uid* 123: gov
- *uid* 857: abc
- *uid* 456: abc
- *uid* 456: gov
- ... (Other 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 INT NOT NULL REFERENCES User(uid)
ON DELETE SET NULL,
....);
```

Option 3: Set NULL
(set all references to NULL)
Deferred constraint checking

- Example:

  ```sql
  CREATE TABLE Dept
  (name CHAR(20) NOT NULL PRIMARY KEY,
   chair CHAR(30) NOT NULL
   REFERENCES Prof(name));
  CREATE TABLE Prof
  (name CHAR(30) NOT NULL PRIMARY KEY,
   dept CHAR(20) NOT NULL
   REFERENCES Dept(name));
  ```

- The first INSERT will always violate a constraint!

- Deferred constraint checking is necessary
  - Check only at the end of a set of operations (transactions)
  - Allowed in SQL as an option
  - Use keyword `deferred`
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

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:

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

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

Checked when new tuples are added to Member but not when User is modified
Naming constraints

- It is possible to name constraints (similar to assertions)

```sql
CREATE TABLE User(
  age INT,  constraint minAge check(age IS NULL OR age > 0),
  ...);
```
Schema modification

• How to add constraints once the schema is defined??

• Add or Modify attributes/domains

• Add or Remove constraints
Add or Modify attributes/domains

- *Alter table* `table_name` *Add column* `column_name`
- *Alter table* `table_name` *Rename column* `old_name` *to* `new_name`
- *Alter table* `table_name` *Drop column* `column_name`

Domain change:

- *Alter table* `table_name` *Alter column* `column_name` *datatype*

*Error if column already has conflicting data!*
Add or Remove constraints

• *Alter table* `table_name` *Add constraint* `constraint_name` `constraint_condition`

```
ALTER TABLE Member
ADD CONSTRAINT fk_user FOREIGN KEY(uid)
REFERENCES User(uid)
```

• *Alter table* `table_name` *Drop constraint* `constraint_name`

```
ALTER TABLE Member
DROP CONSTRAINT fk_user
```
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 lecture: triggers, views, indexes
Two ways to practice queries

• School servers have db2 installed
  • Instructions in db2tutorial.pdf posted along with the project description
  • The JDBC example also provides instructions for the same

• The textbook’s website has an SQLite db that runs in the browser: https://www.db-book.com/university-lab-dir/sqljs.html