Announcements (January 20)

- Reading assignment for this week (Ailamaki et al., VLDB 2001) has been posted
  - Due Wednesday night
  - Hunt for related/follow-up work too!
- Course project will be assigned this Thursday
- Student presentation sign-up sheet will be circulated this Thursday
  - Allows you to drop your lowest homework grade
- Homework #1 due in two weeks

SQL

- SQL: Structured Query Language
  - Pronounced “S-Q-L” or “sequel”
  - The standard query language support by most commercial DBMS
- A brief history
  - IBM System R
  - ANSI SQL89
  - ANSI SQL92 (SQL2)
  - SQL3 (still under construction after years!)
Creating and dropping tables

- **CREATE TABLE** table_name
  (... column_name column_type, ...);
- **DROP TABLE** table_name;

**Examples**
- `create table Student (SID integer, name varchar(30), email varchar(30), age integer, GPA float);`
- `create table Course (CID char(10), title varchar(100));`
- `create table Enroll (SID integer, CID char(10));`
- `drop table Student;`
- `drop table Course;`
- `drop table Enroll;`

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

Basic queries: SFW statement

- **SELECT** A_1, A_2, ..., A_n
  **FROM** R_1, R_2, ..., R_m
  **WHERE** condition;

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

Equivalent (not really!) to relational algebra query

\[ \pi_{A_1, A_2, ..., A_n} (\sigma_{\text{condition}}(R_1 \times R_2 \times ... \times R_m)) \]

Example: reading a table

- **SELECT** * FROM Student;
  - Single-table query, so no cross product here
  - **WHERE** clause is optional
  - * is a short hand for "all columns"
Example: selection and projection

- Name of students under 18
  - SELECT name FROM Student WHERE age < 18;
- When was Lisa born?
  - SELECT 2004 - age
    FROM Student
    WHERE name = 'Lisa';
  - SELECT list can contain expressions
    - Can also use built-in functions such as SUBSTR, ABS, etc.
  - String literals (case sensitive) are enclosed in single quotes

Example: join

- SID's and name's of students taking courses with the word "Database" in their titles
  - SELECT Student.SID, Student.name
    FROM Student, Enroll, Course
    WHERE Student.SID = Enroll.SID
    AND Enroll.CID = Course.CID
    AND title LIKE '%Database%';
  - LIKE matches a string against a pattern
    - % matches any sequence of 0 or more characters
  - Okay to omit table_name in table_name.column_name if column_name is unique

Example: rename

- SID's of students who take at least two courses
  - Relational algebra query:
    \[ \pi_{e1.SID} ( (\rho_{e1.Enroll} >\gamma_{\langle e1.SID = e2.SID \land e1.CID \neq e2.CID \rangle} \rho_{e2.Enroll} ) ) \]
  - SQL:
    SELECT e1.SID AS SID
    FROM Enroll AS e1, Enroll AS e2
    WHERE e1.SID = e2.SID
    AND e1.CID <> e2.CID;
  - AS keyword is completely optional
A more complicated example

Titles of all courses that Bart and Lisa are taking together

Tip: Write the FROM clause first, then WHERE, and then SELECT

Why SFW statements?

Out of many possible ways of structuring SQL statements, why did the designers choose SELECT-FROM-WHERE?

- A large number of queries can be written using only selection, projection, and cross product (or join)
- Any query that uses only these operators can be written in a canonical form: \( \pi_L (\sigma_p (R_1 \times \ldots \times R_n)) \)
  - Example: \( \pi_{R,A,B} (\sigma_{p_1} (R_1 \times S)) \sigma_{p_2} (\pi_{T,C} (\sigma_{p_3} (T))) = \pi_{R,A,B,T,C} (\sigma_{p_1 \land p_2 \land p_3} (R \times S \times T)) \)
  - SELECT-FROM-WHERE captures this canonical form

Set versus bag semantics

- Set
  - No duplicates
  - Relational model and algebra use set semantics
- Bag
  - Duplicates allowed
  - Number of duplicates is significant
  - SQL uses bag semantics by default
Set versus bag example

\[ \pi_{SID} \text{Enroll} \]

\[ \text{Enroll} \]

| SID | Name
|-----|-----|
| 142 | CPS216
| 142 | CPS214
| 123 | CPS216
| 857 | CPS216
| 857 | CPS230
| 456 | CPS214

\[ \text{SELECT SID} \from \text{Enroll}; \]

A case for bag semantics

Besides, SQL provides the option of set semantics with \textsc{DISTINCT} keyword

Operational semantics of SFW

\[ \text{SELECT } (\textsc{DISTINCT} ) E_1, E_2, \ldots, E_n \from R_1, R_2, \ldots, R_m \where \text{condition}; \]

\quad For each \( t_1 \) in \( R_1 \):

\quad \quad For each \( t_2 \) in \( R_2 \); \ldots

\quad \quad \quad For each \( t_m \) in \( R_m \);

\quad \quad \quad \quad If \text{condition} \ is \ true \ over \( t_1, t_2, \ldots, t_m \):

\quad \quad \quad \quad \quad Compute and output \( E_1, E_2, \ldots, E_n \)

\quad \quad If \textsc{DISTINCT} \ is \ present

\quad \quad \quad Eliminate \ duplicate \ rows \ in \ output

\quad \quad \quad \quad \quad \( t_1, t_2, \ldots, t_m \) \ are \ often \ called \ tuple \ variables
Example: forcing set semantics

- SID's of students who take at least two courses
  - SELECT e1.SID AS SID
    FROM Enroll AS e1, Enroll AS e2
    WHERE e1.SID = e2.SID
    AND e1.CID <> e2.CID;

  - SELECT DISTINCT e1.SID AS SID
...
- Duplicate SID values are removed from the output

SQL set and bag operations

- UNION, EXCEPT, INTERSECT
  - Set semantics
  - Exactly like set ∪, −, and ∩ in relational algebra
- UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Bag semantics
  - Think of each row as having an implicit count (the number of times it appears in the table)
  - Bag union: sum up the counts from two tables
  - Bag difference: proper-subtract the two counts
  - Bag intersection: take the minimum of the two counts

Examples of bag operations

<table>
<thead>
<tr>
<th>Bag1</th>
<th>Bag2</th>
<th>Bag1 UNION ALL Bag2</th>
<th>Bag1 INTERSECT ALL Bag2</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>fruit</td>
<td>fruit fruit</td>
<td>fruit fruit</td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
<td>apple apple</td>
<td>apple apple</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
<td>orange orange</td>
<td>orange orange</td>
</tr>
<tr>
<td>Bag1 EXCEPT ALL Bag2</td>
<td>Bag1 INTERSECT ALL Bag2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fruit</td>
<td>fruit</td>
<td>fruit fruit</td>
<td>fruit fruit</td>
</tr>
<tr>
<td>apple</td>
<td>apple</td>
<td>apple apple</td>
<td>apple apple</td>
</tr>
<tr>
<td>orange</td>
<td>orange</td>
<td>orange orange</td>
<td>orange orange</td>
</tr>
</tbody>
</table>
Examples of set versus bag operations

- `Enroll(SID, CID), ClubMember(club, SID)`
  - (SELECT SID FROM ClubMember) EXCEPT (SELECT SID FROM Enroll);
  - (SELECT SID FROM ClubMember) EXCEPT ALL (SELECT SID FROM Enroll);

Table expression

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.
  - A way to "nest" queries
- Example: names of students who are in more clubs than classes
  ```sql
  SELECT DISTINCT name FROM Student,
  (SELECT SID FROM ClubMember) EXCEPT ALL (SELECT SID FROM Enroll)) AS S
  WHERE Student.SID = S.SID;
  ```

Summary of SQL features covered so far

- Basic CREATE/DROP TABLE
- SELECT-FROM-WHERE statements (select-project-join queries)
- Set and bag operations
- Nesting queries using table expressions

- So far, not much more than relational algebra
- Next: aggregation
Aggregates

- Standard SQL aggregate functions: \texttt{COUNT}, \texttt{SUM}, \texttt{AVG}, \texttt{MIN}, \texttt{MAX}
- Example: number of students under 18, and their average GPA
  - \texttt{SELECT COUNT(*), AVG(GPA)}
  - \texttt{FROM Student}
  - \texttt{WHERE age < 18;}
  - \texttt{COUNT(*)} counts the number of rows

GROUP BY

- \texttt{SELECT ... FROM ... WHERE ...}
- \texttt{GROUP BY list_of_columns;}
- Example: find the average GPA for each age group
  - \texttt{SELECT age, AVG(GPA)}
  - \texttt{FROM Student}
  - \texttt{GROUP BY age;}

Operational semantics of GROUP BY

- \texttt{SELECT ... FROM ... WHERE ... GROUP BY ...;}
- Compute \texttt{FROM} (\times)
- Compute \texttt{WHERE} (\sigma)
- Compute \texttt{GROUP BY}: group rows according to the values of \texttt{GROUP BY} columns
- Compute \texttt{SELECT} for each group (\pi)
  - One output row per group in the final output
Example of computing GROUP BY

```
SELECT age, AVG(GPA) FROM Student GROUP BY age;
```

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>2.3</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>4.3</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>3.1</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

`SID` name age GPA
142 Bart 10 2.3
857 Lisa 8 4.3
123 Milhouse 10 3.1
456 Ralph 8 2.3
... ... ...

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_GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.7</td>
</tr>
<tr>
<td>8</td>
<td>3.3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Aggregates with no GROUP BY

An aggregate query with no GROUP BY clause represent a special case where all rows go into one group

```
SELECT AVG(GPA) FROM Student;
```

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>2.3</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>4.3</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>3.1</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Group all rows into one group
Compute aggregate over the group

<table>
<thead>
<tr>
<th>AVG_GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
</tr>
</tbody>
</table>

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

This restriction ensure that
Examples of invalid queries

- SELECT SID, age FROM Student GROUP BY age;
  - Recall there is one output row per group
  - There can be multiple SID values per group
- SELECT MAX(GPA) FROM Student;
  - Recall there is only one group for an aggregate query with no GROUP BY clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work

HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)
- SELECT ... FROM ... WHERE ... GROUP BY ... HAVING condition;
  - Compute FROM (×)
  - Compute WHERE (σ)
  - Compute GROUP BY; group rows according to the values of GROUP BY columns
  - Compute HAVING (another σ over the groups)
  - Compute SELECT (π) for each group that passes HAVING

HAVING examples

- Find the average GPA for each age group over 10
  - SELECT age, AVG(GPA) FROM Student GROUP BY age HAVING age > 10;
  - Can be written using
- List the average GPA for each age group with more than a hundred students
  - SELECT age, AVG(GPA) FROM Student GROUP BY age HAVING COUNT(*) > 100;
  - Can be written using
Summary of SQL features covered so far

- Basic CREATE/DROP TABLE
- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions
- Aggregation and grouping
  - More expressive power than relational algebra

Next: NULL's

Incomplete information

- Example: Student (SID, name, age, GPA)
- Value unknown
  - We do not know Nelson's age
- Value not applicable
  - Nelson has not taken any classes yet; what is his GPA?

Solution 1

- A dedicated special value for each domain (type)
  - GPA cannot be \(-1\), so use \(-1\) as a special value to indicate a missing or invalid GPA
  - Leads to incorrect answers if not careful
    - SELECT AVG(GPA) FROM Student;
  - Complicates applications
    - SELECT AVG(GPA) FROM Student WHERE GPA <> -1;
  - Remember the pre-Y2K bug?
    - 09/09/99 was used as a missing or invalid date value
Solution 2

- A valid-bit for every column
  - `Student (SID, name, name_is_valid,
    age, age_is_valid,
    GPA, GPA_is_valid)`

- Still complicates applications
  - `SELECT AVG(GPA) FROM Student
    WHERE GPA_is_valid;`

SQL’s solution

- A special value NULL
  - Same for every domain
  - Special rules for dealing with NULL’s

- Example: `Student (SID, name, age, GPA)`
  - `{ 789, "Nelson", NULL, NULL }`

Rules for NULL’s

- When we operate on a NULL and another value
  (including another NULL) using +, -, etc., the result is NULL

- Aggregate functions ignore NULL, except `COUNT(*)`
  (since it counts rows)
Three-valued logic

- When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN
- TRUE = 1, FALSE = 0, UNKNOWN = 0.5
- \( x \text{ AND } y = \min(x, y) \)
- \( x \text{ OR } y = \max(x, y) \)
- \( \text{NOT } x = 1 - x \)
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - UNKNOWN is insufficient

Unfortunate consequences

- SELECT AVG(GPA) FROM Student;
  SELECT SUM(GPA)/COUNT(*) FROM Student;

- SELECT * FROM Student;
  SELECT * FROM Student WHERE GPA = GPA;

= Be careful: NULL breaks many equivalences

Another problem

- Example: Who has NULL GPA values?
  - SELECT * FROM Student WHERE GPA = NULL;
  
  * Works, but ugly
  * Introduced built-in predicates IS NULL and IS NOT NULL
  - SELECT * FROM Student WHERE GPA IS NULL;
Summary of SQL features covered so far

- Basic CREATE/DROP TABLE
- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions
- Aggregation and grouping
- NULL’s

Next: subqueries, modifications, constraints, and views