SQL: Part I

CPS 216

Advanced Database Systems

Announcements

- Reading assignment for this week: "A History and Evaluation of System R," by Chamberlin et al.
- Homework #1 assigned today
  - Due February 10 (in 2-½ weeks)
- Course project assigned today
  - Milestone 1 (proposal): March 5 (after midterm and before spring break)
  - Milestone 2 (status report): April 14
  - Demo period (final report): April 28 – May 3
- No recitation session this Friday (January 24)

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*$_i$ *column_type*$_i$`, …);`
- `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
  $\pi_{A_1, A_2, \ldots, A_n}(\sigma_{condition}(R_1 \times R_2 \times \ldots \times R_m))$

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 2003 – 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:

    $\pi_{e1.SID}$
    $((\rho_{e1}\ Enroll) \bowtie_{e1.SID\ =\ e2.SID\ \wedge\ e1.CID\ \neq\ e2.CID} (\rho_{e2}\ Enroll))$
  - 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

```
SELECT c.title
FROM Student sb, Student sl, Enroll eb, Enroll el, Course c
WHERE sb.name = 'Bart' AND sl.name = 'Lisa'
AND eb.SID = sb.SID AND el.SID = el.SID
AND eb.CID = el.CID
AND eb.CID = c.CID;
```

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: $\pi_L (\sigma_p (R_1 \times \ldots \times R_m))$
    - Example: $\pi_{R.A,\ S.B} (R \bowtie_{p1} S) \bowtie_{p2} (\pi_{T.C}\ \sigma_{p3}\ T) = \pi_{R.A,\ S.B,\ T.C}\ \sigma_{p1 \wedge p2 \wedge p3} (R \times S \times T)$
  - `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

Enroll

SID	*CID*
142	CPS216
142	CPS214
123	CPS216
857	CPS216
857	CPS230
456	CPS214
...	...

π_{SID} *Enroll*

SID
142
123
857
456
...

```
SELECT SID
FROM Enroll;
```

SID
142
142
123
857
857
456
...

A case for bag semantics

- ❖ Efficiency
  - Saves time of eliminating duplicates
- ❖ Which one is more useful?
  - π_{GPA} *Student*
  - `SELECT GPA FROM Student;`
  - The first query just returns all possible GPA's
  - The second query returns the actual GPA distribution
- ❖ Besides, SQL provides the option of set semantics with `DISTINCT` keyword

Operational semantics of SFW

- ❖ `SELECT [DISTINCT]` E_1, E_2, …, E_n
  `FROM` R_1, R_2, …, R_m
  `WHERE` *condition*;
- ❖ For each t_1 in R_1:

```
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 $t_1, t_2, …, t_m$:
                Compute and output $E_1, E_2, …, E_n$
If DISTINCT is present
    Eliminate duplicate rows in output
```

- ❖ t_1, t_2, …, 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;
    ```
    - What if Bart takes CPS216 and CPS214?
    - Changing `<>` to `>` may help in this case
    - But what if Bart takes CPS216, CPS214, and CPS230?

  -
    ```
    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

`Bag1`

fruit
apple
apple
orange

`Bag2`

fruit
apple
orange
orange

`Bag1 UNION ALL Bag2`

fruit
apple
apple
orange
apple
orange
orange

`Bag1 EXCEPT ALL Bag2`

fruit
apple

`Bag1 INTERSECT ALL Bag2`

fruit
apple
orange

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
(SELECT SID FROM Enroll);
```

  - SID's of students who are in clubs but not taking any classes

```
(SELECT SID FROM ClubMember)
EXCEPT ALL
(SELECT SID FROM Enroll);
```

  - SID's of students who are in more clubs than classes

Summary of SQL features covered so far

❖ SELECT-FROM-WHERE statements (select-project-join queries)

❖ Set and bag operations

☞ Next: how to nest SQL queries

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

```
SELECT DISTINCT name
FROM Student,
     ((SELECT SID FROM ClubMember)
      EXCEPT ALL
      (SELECT SID FROM Enroll)) AS S
WHERE Student.SID = S.SID;
```

Scalar subqueries

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

❖ Example: students at the same age as Bart

```
SELECT *
FROM Student
WHERE age = (SELECT age
             FROM Student
             WHERE name = 'Bart');
```

What's Bart's age?

❖ Runtime error if subquery returns more than one row

❖ Under what condition can we be sure that this runtime error would not occur?

- ▪ name is a key of Student

❖ What if subquery returns no rows?

IN subqueries

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

❖ Example: students at the same age as (some) Bart

```
SELECT *
FROM Student
WHERE age IN (SELECT age
              FROM Student
              WHERE name = 'Bart');
```

What's Bart's age?

EXISTS subqueries

❖ EXISTS (*subquery*) checks if the result of *subquery* is non-empty

❖ Example: students at the same age as (some) Bart

```
SELECT *
FROM Student AS s
WHERE EXISTS (SELECT * FROM Student
              WHERE name = 'Bart'
              AND age = s.age);
```

- ▪ It is a correlated subquery—a subquery that references tuple variables in surrounding queries

Operational semantics of subqueries

❖
```
SELECT *
FROM Student AS s
WHERE EXISTS (SELECT * FROM Student
              WHERE name = 'Bart'
              AND age = s.age);
```

❖ For each row `s` in `Student`
- Evaluate the subquery with the appropriate value of `s.age`
- If the result of the subquery is not empty, output `s.*`

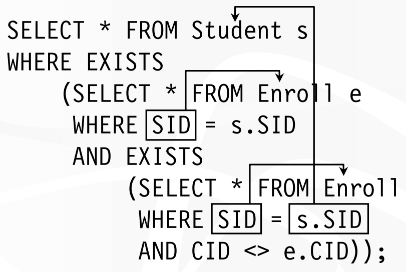
❖ The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)

Scoping rule of subqueries

❖ To find out which table a column belongs to
- Start with the immediately surrounding query
- If not found, look in the one surrounding that; repeat if necessary

❖ Use *table_name*`.`*column_name* notation and `AS` (renaming) to avoid confusion

Another example



```
SELECT * FROM Student s
WHERE EXISTS
    (SELECT * FROM Enroll e
     WHERE SID = s.SID
     AND EXISTS
         (SELECT * FROM Enroll
          WHERE SID = s.SID
          AND CID <> e.CID));
```

Students who are taking at least two courses

Quantified subqueries

❖ A quantified subquery can be used as a value in a `WHERE` condition

❖ Universal quantification (for all):
… `WHERE` *x op* `ALL` (*subquery*) …
- True iff for all *t* in the result of *subquery*, *x op t*

❖ Existential quantification (exists):
… `WHERE` *x op* `ANY` (*subquery*) …
- True iff there exists some *t* in the result of *subquery* such that *x op t*

☞Beware
- In common parlance, "any" and "all" seem to be synonyms
- In SQL, `ANY` really means "some"

Examples of quantified subqueries

❖ Which students have the highest GPA?

-
  ```
  SELECT *
  FROM Student
  WHERE GPA >= ALL
      (SELECT GPA FROM Student);
  ```

-
  ```
  SELECT *
  FROM Student
  WHERE NOT
      (GPA < ANY (SELECT GPA FROM Student);
  ```

☞Use `NOT` to negate a condition

More ways of getting the highest GPA

❖ Which students have the highest GPA?

-
  ```
  SELECT *
  FROM Student AS s
  WHERE NOT EXISTS
      (SELECT * FROM Student
       WHERE GPA > s.GPA);
  ```

-
  ```
  SELECT * FROM Student
  WHERE SID NOT IN
      (SELECT s1.SID
       FROM Student AS s1, Student AS s2
       WHERE s1.GPA < s2.GPA);
  ```

ORDER BY

- SELECT {DISTINCT} ...
  FROM ... WHERE ...
  ORDER BY *output_column* {ASC | DESC}, ...;
- ASC = ascending, DESC = descending
- Operational semantics
  - After SELECT list has been computed and optional duplicate elimination has been carried out,
    sort the output according to ORDER BY specification

ORDER BY example

- List all students, sort them by GPA (descending) and then name (ascending)

  -
    ```
    SELECT SID, name, age, GPA
    FROM Student
    ORDER BY GPA 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 of output columns instead
    ORDER BY 4 DESC, 2;

Summary of SQL features covered so far

- SELECT-FROM-WHERE statements
- Set and bag operations
- Table expressions, subqueries
  - Subqueries allow queries to be written in more declarative ways (recall the highest GPA query)
  - But they do not add any expressive power
    - Try translating other forms of subqueries into [NOT] EXISTS, which in turn can be translated into join (and difference)
- Ordering
  - More expressive power than relational algebra

☞ Next: aggregation and grouping, NULL's, data modification, constraints, ...