Database Management Systems - I, CS 157A

SQL JOIN, Aggregate, Grouping, HAVING and DML Clauses

JOIN

Database Logical and Physical JOIN Operators Overview

Join (Logical Join) Operators

- **Inner Join:**
  - Cross Join (X): cartesian product
  - Equi-Join (where R1.col1 = R2.col2): cross join with equality predicates only
  - Natural Join ($\bowtie$): cross join with union of the attributes of the two relations
  - Theta Join ($\bowtie_C$): like natural join but we apply a boolean-valued condition
- **Outer Join:**
  - Left Outer Join (left join): for every tuple on left relation, join with every tuple on the right relation and if none matches the condition return a tuple with left side and NULLs for the right side relation
  - Right Outer Join (right join): opposite of the left join
  - Full Outer Join (full join): union of left join and right join
- **Self Join:** joining table to itself

Database Logical and Physical Operators Overview

Physical Operators (Ch 15.1)

Physical Join Operators

- **Nested Join:** every outer element is tested against the inner table
- **Merge Join:** Efficient if both tables are already sorted on the join attribute
- **Hash Join:** Only used for equi-join

Sort

Scan

…

Cartesian Products

- **A Cartesian product is formed when:**
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table

- To avoid a Cartesian product, always include a valid join condition

Outer joins - Details

- R **OUTER JOIN** S is the core of an outerjoin expression. It is modified by:
  1. Optional NATURAL in front of OUTER.
  2. Optional ON <condition> after JOIN.
  3. Optional LEFT, RIGHT, or FULL before OUTER:
     - LEFT = pad dangling tuples of R only.
     - RIGHT = pad dangling tuples of S only.
     - FULL = pad both; this choice is the default.

Only one of these

Left Outer Join

```
SELECT    e.last_name, d.department_name
FROM      employees e LEFT OUTER JOIN
          departments d
ON        (e.department_id = d.department_id);
```

- Display all the rows in the EMPLOYEES table, which is the left table, even if there is no match in the DEPARTMENTS table

Right Outer Join

```
SELECT    e.last_name, d.department_name
FROM      employees e  RIGHT OUTER JOIN
          departments d
ON        (e.department_id =
          d.department_id);
```

- Display all the rows in the DEPARTMENTS table, which is the right table, even if there is no match in the EMPLOYEES table

Full Outer Join

```
SELECT    e.last_name, d.department_name
FROM      employees e  FULL OUTER JOIN
          departments d
ON        (e.department_id =
           d.department_id);
```

- Display all the rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. Also display all the rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

Aggregate

Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(*) counts the number of tuples.

Example: Aggregation

- From Sells(bar, beer, price), find the average price of Bud in all bars:

```
SELECT AVG(price)
FROM   Sells
WHERE  beer = 'Bud';
```

Eliminating Duplicates in an Aggregation

- Use DISTINCT inside an aggregation.
- Example: find the number of *different* prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM   Sells
WHERE  beer = 'Bud';
```

NULL's Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.
  - Exception: COUNT of an empty set is 0.

Example: Effect of NULL's

```
SELECT count(*)
FROM Sells
WHERE beer = 'Bud';
```

The number of bars that sell Bud.

Sells(bar, beer, price)

```
SELECT  count(*)
FROM      Sells
WHERE (beer = 'Bud'    AND
              price = xxx);
```

The number of bars that sell Bud at a known price.

GROUP-BY

Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.

Example: Grouping

- From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)
FROM   Sells
GROUP BY beer;
```

beer	AVG(price)
Bud	2.33
…	…

Example: Grouping

- From Sells(bar, beer, price) and Frequents(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'Bud' AND
      Frequents.bar = Sells.bar
GROUP BY drinker;
```

Compute all <drinker-bar-price> triples for Bud.

Then group them by drinker.

Restriction on SELECT Lists With Aggregation

- If Group-By is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example

- You might think you could find the bar that sells Bud the cheapest by:

```
SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Bud';
```

- But this query is illegal in SQL.

HAVING

HAVING Clauses

- **HAVING** <condition> may follow a **GROUP BY** clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Example: HAVING

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

Sells(bar, beer, price) Beers(name, manf)

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING  COUNT(bar) >= 3  OR
     beer IN (SELECT name
              FROM Beers
              WHERE manf = 'Pete''s');
```

Beer groups sold by at least 3 bars and also beer groups where the manufacturer is Pete's.

Beers manu-factured by Pete's.

Requirements on HAVING Conditions

- Anything goes in a subquery.
- Outside sub-queries (Having clause), they may refer to attributes only if they are either:
  1. A grouping attribute (beer), or
  2. Aggregated (bar)

  (same condition as for SELECT clauses with aggregation).

DML

Database Modifications (DML)

- A *modification* command does not return a result (as a query does), but returns changes the database in some way – number of tuples impacted.
- Three kinds of modifications (IUD):
  1. *Insert* a tuple or tuples.
  2. *Update* the value(s) of an existing tuple or tuples.
  3. *Delete* a tuple or tuples.

Insertion

- To insert a single tuple:

  INSERT INTO <relation>

  VALUES (<list of values>);

- Example: add to Likes(drinker, beer) the fact that Sally likes Bud:

```
INSERT INTO Likes
       VALUES('Sally', 'Bud');
```

Specifying Attributes in INSERT

- We may add to the relation name a list of attributes.
- **Two reasons to do so:**
  1. We forget the standard order of attributes for the relation.
  2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
       VALUES('Bud', 'Sally');
```

Adding Default Values

- In a **CREATE** TABLE statement, we can follow an attribute by **DEFAULT** and a value.

- When an inserted tuple has no value for that attribute, the default will be used.

Example: Default Values

```
CREATE TABLE Drinkers (
    name CHAR(30) PRIMARY KEY,
    addr CHAR(50) DEFAULT '123 Sesame St.',
    phone CHAR(16)
);
```

Example: Default Values

```
INSERT INTO Drinkers(name)
       VALUES('Sally');
```

Resulting tuple:

name	address	phone
Sally	123 Sesame St	NULL

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

  INSERT INTO <relation> (<subquery>);

Example: Insert a Subquery

- Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.

Solution

The other drinker

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same.

```
INSERT INTO PotBuddies
(SELECT d2.drinker
FROM Frequents d1, Frequents d2
WHERE d1.drinker = 'Sally' AND
      d2.drinker <> 'Sally' AND
      d1.bar = d2.bar);
```

Deletion

- To delete tuples satisfying a condition from some relation:

DELETE FROM <relation>
WHERE <condition>;

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
      beer = 'Bud';
```

Example: Delete all Tuples

- Make the relation “Likes” empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Some Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS (
    SELECT name FROM Beers
    WHERE manf = b.manf AND
          name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion --- (1)

- Suppose Anheuser-Busch makes only Bud and Bud Lite.
- Suppose we come to the tuple *b* for Bud first.
- The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- Now, when *b* is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion --- (2)

- **Answer:** we *do* delete Bud Lite as well.
- The reason is that deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied.
  2. Delete the marked tuples at the end.

Updates

- To change certain attributes in certain tuples of a relation:

  UPDATE <relation>

  SET <list of attribute assignments>

  WHERE <condition on tuples>;

Example: Update

- Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers
SET    phone = '555-1212'
WHERE  name = 'Fred';
```

Example: Update Several Tuples

- Make $4 the maximum price for beer:

```
UPDATE Sells
SET    price = 4.00
WHERE  price > 4.00;
```

END