

More SQL

EXTENDED RELATIONAL ALGEBRA

OUTERJOINS,
GROUPING/AGGREGATION

INSERT/DELETE/UPDATE

The Extended Algebra

δ = eliminate duplicates from bags.

τ = sort tuples.

γ = grouping and aggregation.

Outerjoin : avoids “dangling tuples” = tuples that do not join with anything.

Duplicate Elimination

$$R1 := \delta(R2).$$

R1 consists of one copy of each tuple that appears in R2 one or more times.

Example: Duplicate Elimination

$R =$ (

A	B
1	2
3	4
1	2

)

$\delta(R) =$

A	B
1	2
3	4

Sorting

$R1 := \mathbf{T}_L (R2).$

- L is a list of some of the attributes of $R2$.

$R1$ is the list of tuples of $R2$ sorted first on the value of the first attribute on L , then on the second attribute of L , and so on.

- Break ties arbitrarily.

\mathbf{T} is the only operator whose result is neither a set nor a bag.

Example: Sorting

$R =$

A	B
1	3
3	4
5	2

$$\tau_B(R) = [(5,2), (1,3), (3,4)]$$

Aggregation Operators

They apply to entire columns of a table and produce a single result.

The most important examples: SUM, AVG, COUNT, MIN, and MAX.

Example: Aggregation

R = (

A	B
1	3
3	4
3	2

)

SUM(A) = 7

COUNT(A) = 3

MAX(B) = 4

AVG(B) = 3

Grouping Operator

$R1 := \gamma_L (R2)$. L is a list of elements that are either:

1. Individual (*grouping*) attributes.
2. $AGG(A)$, where AGG is one of the aggregation operators and A is an attribute.
 - An arrow and a new attribute name renames the component.

Applying $\gamma_L(R)$

Group R according to all the grouping attributes on list L .

- That is: form one group for each distinct list of values for those attributes in R .

Within each group, compute $AGG(A)$ for each aggregation on list L .

Result has one tuple for each group:

1. The grouping attributes and
2. Their group's aggregations.

Example: Grouping/Aggregation

$R = ($

A	B	C
1	2	3
4	5	6
1	2	5

Then, average C
within groups:

A	B	X
1	2	4
4	5	6

$\Upsilon_{A,B,AVG(C)} \rightarrow X (R) = ??$

First, group R by A and B :

A	B	C
1	2	3
1	2	5
4	5	6

Outerjoin

Suppose we join $R \bowtie_c S$.

A tuple of R that has no tuple of S with which it joins is said to be *dangling*.

- Similarly for a tuple of S .

Outerjoin preserves dangling tuples by padding them NULL.

Example: Outerjoin

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

(1,2) joins with (2,3), but the other two tuples are dangling.

R OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

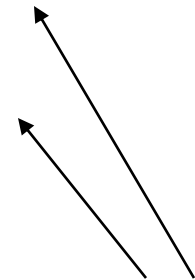
Now --- Back to SQL

EACH OPERATION HAS A SQL
EQUIVALENT

SQL Outerjoins syntax

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 S.
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

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:

```
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 table is 0.

Example: Effect of NULL's


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

The number of bars
that sell Bud.



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

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



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
Miller	2.23
...	...

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 any aggregation is used, then each element of the SELECT list must be either:

1. Aggregated, or
2. An attribute on the GROUP BY list.

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

Beer groups with at least 3 non-NULL bars or beer groups where the manufacturer is Pete's.

```
SELECT beer, AVG(price)
```

```
FROM Sells
```

```
GROUP BY beer
```

```
HAVING COUNT(bar) >= 3 OR
```

```
beer IN (SELECT name
```

```
FROM Beers
```

```
WHERE manf = 'Pete''s');
```

Beers manufactured by Pete's.

Requirements on HAVING Conditions

Anything goes in a subquery.

Outside subqueries, they may refer to attributes only if they are either:

1. A grouping attribute, or
2. Aggregated

(same condition as for SELECT clauses with aggregation).

Database Modifications

A *modification* command does not return a result (as a query does), but changes the database in some way.

Three kinds of modifications:

1. *Insert* a tuple or tuples.
2. *Delete* a tuple or tuples.
3. *Update* the value(s) of an existing 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.

The other
drinker

Solution

```
INSERT INTO PotBuddies
```

```
(SELECT d2.drinker
```

```
FROM Frequents d1, Frequents d2
```

```
WHERE d1.drinker = 'Sally' AND
```

```
d2.drinker <> 'Sally' AND
```

```
d1.bar = d2.bar
```

```
);
```

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

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.

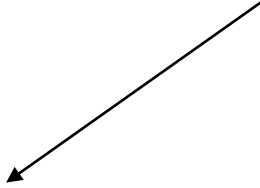
```
DELETE FROM Beers b1
```

```
WHERE EXISTS (
```

```
SELECT name FROM Beers b2
```

```
WHERE b1.manf = b2.manf AND  
      b1.name <> b2.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.

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;
```

Actions

Review slides.

Read chapter about SQL (Chapters 6.1-4 in 2nd Edition of course book).

Play with SQL [w3schools.com](https://www.w3schools.com)