

More Algebra and SQL

Basic operations in algebra and SQL

Algebra of Bags

Extended Relational Algebra

Outerjoins, Grouping/Aggregation

Core Relational Algebra

- Union, intersection, and difference.
 - Usual set operations, but *both operands must have the same relation schema*.
- Selection: picking certain rows.
- Projection: picking certain columns.
- Products and joins: compositions of relations.
- Renaming of relations and attributes.

Union, intersection, difference

□ $R \cup S$

`SELECT ... UNION SELECT ...;`

(Duplicate elimination -> UNION ALL: multiset, UNION: set)

□ $R \cap S$

`SELECT ... INTERSECT SELECT ...;`

□ $R - S$

`SELECT ... MINUS SELECT ...;`

(Some DBMSs use EXCEPT instead of MINUS)

Selection

□ $R1 := \sigma_C(R2)$

□ C is a condition (as in “if” statements) that refers to attributes of $R2$.

□ $R1$ is all those tuples of $R2$ that satisfy C .

SELECT * FROM R2 WHERE C;

Projection

□ $R1 := \pi_L(R2)$

□ L is a list of attributes from the schema of $R2$.

□ $R1$ is constructed by looking at each tuple of $R2$, extracting the attributes on list L , in the order specified, and creating from those components a tuple for $R1$.

□ **Eliminate duplicate** tuples, if any.

□ **SELECT DISTINCT** L **FROM** $R2$;

Product

□ $R3 := R1 \times R2$

- Pair each tuple $t1$ of $R1$ with each tuple $t2$ of $R2$.
- Concatenation $t1t2$ is a tuple of $R3$.
- Schema of $R3$ is the attributes of $R1$ and then $R2$, in order.
- But beware attribute A of the same name in $R1$ and $R2$: use $R1.A$ and $R2.A$.

SELECT * FROM $R1, R2$; or

SELECT * FROM $R1$ CROSS JOIN $R2$;

Example: $R3 := R1 \times R2$

R1(

A,	B)
1	2
3	4

R2(

B,	C)
5	6
7	8
9	10

R3(

A,	R1.B,	R2.B,	C)
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

SELECT A, R1.B, R2.B, C FROM R1, R2;

Theta-Join

□ $R3 := R1 \bowtie_C R2$

□ Take the product $R1 \times R2$.

□ Then apply σ_C to the result.

□ As for σ , C can be any boolean-valued condition.

□ Historic versions of this operator allowed only $A \theta B$, where θ is $=$, $<$, etc.; hence the name “theta-join.”

```
SELECT * FROM R1 JOIN R2 ON (C);
```


Example: Theta Join

Sells(

bar,	beer,	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

)

Bars(

name,	addr
Joe's	Maple St.
Sue's	River Rd.

)

BarInfo := Sells $\bowtie_{\text{Sells.bar} = \text{Bars.name}}$ Bars

BarInfo(

bar,	beer,	price,	name,	addr
Joe's	Bud	2.50	Joe's	Maple St.
Joe's	Miller	2.75	Joe's	Maple St.
Sue's	Bud	2.50	Sue's	River Rd.
Sue's	Coors	3.00	Sue's	River Rd.

)

Natural Join

- A useful join variant (*natural* join) connects two relations by:
 - Equating attributes of the same name, and
 - Projecting out one copy of each pair of equated attributes.
- Denoted $R3 := R1 \bowtie R2$.

```
SELECT * FROM R1 NATURAL JOIN R2;
```

Example: Natural Join

Sells(

bar,	beer,	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

)

Bars(

bar,	addr
Joe's	Maple St.
Sue's	River Rd.

)

BarInfo := Sells \bowtie Bars

Note: Bars.name has **become Bars.bar** to make the natural join “work.”

BarInfo(

bar,	beer,	price,	addr
Joe's	Bud	2.50	Maple St.
Joe's	Milller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

)

Renaming

- The ρ operator gives a new schema to a relation. [suppose: $R2(X1, X2, \dots Xn)$]
- $R1 := \rho_{R1(A1, \dots, An)}(R2)$ makes $R1$ be a relation with attributes $A1, \dots, An$ and the same tuples as $R2$.
- Simplified notation: $R1(A1, \dots, An) := R2$.

SELECT $X1$ as $A1$, $X2$ as $A2$, ... Xn as An FROM $R2$;

CREATE TABLE $R1$ AS SELECT $X1$ $A1$, $X2$ $A2$, ... Xn An FROM $R2$;

Example: Renaming

Bars(

name,	addr
Joe's	Maple St.
Sue's	River Rd.

)

$R(\text{bar}, \text{addr}) := \text{Bars}$

R(

bar,	addr
Joe's	Maple St.
Sue's	River Rd.

)

Relational Algebra on Bags

- A *bag* (or *multiset*) is like a set, but an element may appear more than once.
- Example: $\{1, 2, 1, 3\}$ is a bag.
- Example: $\{1, 2, 3\}$ is also a bag that happens to be a set.

Why Bags?

- **SQL**, the most important query language for relational databases, is actually **a bag language**.
- Some operations, like projection, are **more efficient** on bags than sets.

Operations on Bags

- **Selection** applies to each tuple, so its effect on bags is like its effect on sets.
- **Projection** also applies to each tuple, but as a bag operator, we **do not eliminate duplicates**.
- **Products** and **joins** are done on each pair of tuples, so duplicates in bags have no effect on how we operate.

Example: Bag Selection

R(

A,	B
1	2
5	6
1	2

)

$\sigma_{A+B < 5} (R) =$

A	B
1	2
1	2

SELECT * FROM R WHERE A+B < 5;

Example: Bag Projection

R(

A,	B
1	2
5	6
1	2

)

$\pi_A(R) =$

A
1
5
1

SELECT A FROM R;

Example: Bag Product

R(

A,	B
1	2
5	6
1	2

)

S(

B,	C
3	4
7	8

)

R x S =

A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	3	4
5	6	7	8
1	2	3	4
1	2	7	8

Example: Bag Theta-Join

R(

A,	B
1	2
5	6
1	2

)

S(

B,	C
3	4
7	8

)

R $\bowtie_{R.B < S.B}$ S =

A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	7	8
1	2	3	4
1	2	7	8

Bag Union

- An element appears in the union of two bags the **sum of the number of times** it appears in each bag.
- **Example:** $\{1, 2, 1\} \cup \{1, 1, 2, 3, 1\} = \{1, 1, 1, 1, 1, 2, 2, 3\}$

`SELECT ... UNION ALL SELECT ...;`

Bag Intersection

- An element appears in the intersection of two bags the minimum of the number of times it appears in either.
- **Example:** $\{1,2,1,1\} \cap \{1,2,1,3\} = \{1,1,2\}$.

`SELECT ... INTERSECT ALL SELECT ...;`

(Oracle doesn't support 'ALL' !)

Bag Difference

- An element appears in the difference $A - B$ of bags as many times as it appears in A , minus the number of times it appears in B .
 - But never less than 0 times.
- **Example:** $\{1,2,1,1\} - \{1,2,3\} = \{1,1\}$.
`SELECT ... MINUS ALL SELECT ...;`
ORACLE doesn't support 'ALL' !)

Beware: Bag Laws \neq Set Laws

- Some, but *not all* algebraic laws that hold for sets also hold for bags.
- **Example:** the commutative law for union ($R \cup S = S \cup R$) *does* hold for bags.
 - Since addition is commutative, adding the number of times x appears in R and S doesn't depend on the order of R and S .

Example: A Law That Fails

- Set union is *idempotent*, meaning that $S \cup S = S$.
- However, for bags, if x appears n times in S , then it appears $2n$ times in $S \cup S$.
- Thus $S \cup S \neq S$ in general.
 - e.g., $\{1\} \cup \{1\} = \{1,1\} \neq \{1\}$.

The Extended Algebra

π_L extended projection

δ = eliminate duplicates from bags.

τ = sort tuples.

γ = grouping and aggregation.

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

Extended Projection

- Using the same π_L operator, we allow the list L to contain arbitrary expressions involving attributes:

1. Arithmetic on attributes, e.g., $A+B \rightarrow C$

" \rightarrow " stands for renaming the attribute in the result to "C"

2. Duplicate occurrences of the same attribute.

`SELECT A+B AS C FROM R;`

("AS" is optional)

Example: Extended Projection

$R =$ (

A	B
1	2
3	4

)

$\pi_{A+B \rightarrow C, A, A}(R) =$

C	A1	A2
3	1	1
7	3	3

SELECT A+B as C, A as A1, A as A2 FROM R;

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

- $R1 := \tau_L(R2)$.
 - 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.
- τ is the only operator whose result is neither a set nor a bag.

Example: Sorting

$R =$ (

A	B
1	2
3	4
5	2

)

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

$$\tau_{B,A}(R) = [(1,2), (5,2), (3,4)]$$

Aggregation Operators

- Aggregation operators are not operators of relational algebra.
- Rather, 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

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

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.

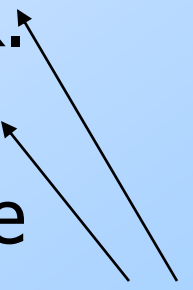
R OUTERJOIN S =

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

Now --- Back to SQL

Each Operation Has a SQL
Equivalent

Outerjoins

- 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
- 

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 set is 0.

Example: Effect of NULL's

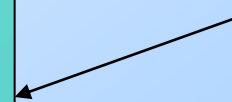
```
SELECT count(*)  
FROM Sells  
WHERE beer = 'Bud';
```

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
...	...

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.

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 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.

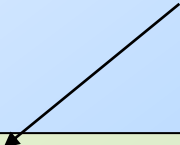
Example: HAVING

- 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

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


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



```
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).