

CSE 544

Principles of Database Management Systems

Fall 2016

Lecture 2 – Relational Algebra and SQL

Announcements

- Paper review
 - First paper review is posted now (due Wednesday 6pm)
 - Details on website
- Milestone 1 of the project was due
 - You don't need to choose a project yet; more suggestions will continue to be posted on website
 - M2 Project Proposal due next Wednesday

Outline

Three topics today

- Relational algebra
- Crash course on SQL

Relational Operators

- **Selection**: $\sigma_{\text{condition}}(\mathbf{S})$
 - Condition is Boolean combination (\wedge, \vee) of terms
 - Term is: attr. op constant, attr. op attr.
 - Op is: $<$, \leq , $=$, \neq , \geq , or $>$
- **Projection**: $\pi_{\text{list-of-attributes}}(\mathbf{S})$
- **Union** (\cup), **Intersection** (\cap), **Set difference** ($-$),
- **Cross-product** or **cartesian product** (\times)
- **Join**: $\mathbf{R} \bowtie_{\theta} \mathbf{S} = \sigma_{\theta}(\mathbf{R} \times \mathbf{S})$
- **Division**: \mathbf{R}/\mathbf{S}
- **Rename** $\rho(\mathbf{R}(\mathbf{F}), \mathbf{E})$

Join Galore

- **Theta-join:** $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
 - Join of R and S with a join condition θ
 - Cross-product followed by selection θ
- **Equijoin:** $R \bowtie_{\theta} S = \pi_A (\sigma_{\theta}(R \times S))$
 - Join condition θ consists only of equalities
 - ~~Projection π_A drops all redundant attributes*~~

*Alvin is wrong...

- **Natural join:** $R \bowtie S = \pi_A (\sigma_{\theta}(R \times S))$
 - aka Equijoin
 - Equality on **all** fields with same name in R and in S
 - Natural join does drop redundant attributes

Theta-Join Example

AnonPatient P

age	zip	disease
50	98125	heart
19	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$$P \bowtie_{P.zip = V.zip \text{ and } P.age \leq V.age + 1 \text{ and } P.age \geq V.age - 1} V$$

P.age	P.zip	disease	name	V.age	V.zip
19	98120	flu	p2	20	98120

Equijoin Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie_{P.age=V.age} V$

age	P.zip	disease	name	V.zip
54	98125	heart	p1	98125
20	98120	flu	p2	98120

Natural Join Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie V$

age	zip	disease	name
54	98125	heart	p1
20	98120	flu	p2

Even More Joins

- **Outer join**
 - Include tuples with no matches in the output
 - Use NULL values for missing attributes
- Variants
 - Left outer join
 - Right outer join
 - Full outer join

Outer Join Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu
33	98120	lung

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie V$

age	zip	disease	name
54	98125	heart	p1
20	98120	flu	p2
33	98120	lung	null

Extended Operators of Relational Algebra

- Duplicate elimination (δ)
 - Since commercial DBMSs operate on **multisets/bags** not sets
- Aggregate operators (γ)
 - Useful in practice and requires bag semantics
 - Min, max, sum, average, count
- Grouping operators (γ)
 - Partitions tuples of a relation into “groups”
 - Aggregates can then be applied to groups
- Sort operator (τ)

Relational Calculus

- Alternative to relational algebra
 - Declarative query language
 - Describe what we want NOT how to get it
- Tuple relational calculus query
 - $\{ T \mid p(T) \}$
 - Where T is a tuple variable
 - $p(T)$ denotes a formula that describes T
 - Result: set of all tuples for which $p(T)$ is true
 - Language for $p(T)$ is subset of **first-order logic**

Q1: Names of patients who have heart disease

$\{ T \mid \exists P \in \text{AnonPatient} \exists V \in \text{Voter}$

$(P.\text{zip} = V.\text{zip} \wedge P.\text{age} = V.\text{age} \wedge P.\text{disease} = \text{'heart'} \wedge T.\text{name} = V.\text{name}) \}$

Example

- Show set division on white board...

Outline

Three topics today

- Wrap up relational algebra
- Crash course on SQL
- Brief overview of database design

Structured Query Language: SQL

- Influenced by relational calculus
- Declarative query language
- Multiple aspects of the language
 - Data definition language (DDL)
 - Statements to create, modify tables and views
 - Data manipulation language (DML)
 - Statements to issue queries, insert, delete data
 - More

Outline

- Today: crash course in **SQL DML**
 - Data Manipulation Language
 - **SELECT-FROM-WHERE-GROUPBY**
 - Study independently: **INSERT/DELETE/MODIFY**
- Study independently **SQL DDL**
 - Data Definition Language
 - **CREATE TABLE, DROP TABLE, CREATE INDEX, CLUSTER, ALTER TABLE, ...**
 - E.g. google for the postgres manual, or type this in psql:
`\h create`
`\h create table`
`\h cluster`

SQL Query

Basic form: (plus many many many more bells and whistles)

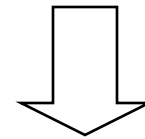
```
SELECT <attributes>  
FROM   <one or more relations>  
WHERE  <conditions>
```

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```

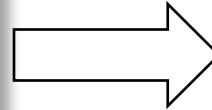


“selection” and
“projection”

PName	Price	Manufacturer
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

Eliminating Duplicates

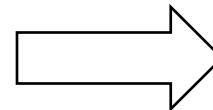
```
SELECT DISTINCT category  
FROM Product
```



Category
Gadgets
Photography
Household

Compare to:

```
SELECT category  
FROM Product
```



Category
Gadgets
Gadgets
Photography
Household

Ordering the Results

```
SELECT pname, price, manufacturer
FROM Product
WHERE category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the 2nd attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the **DESC** keyword.

Can also request only top-k with **LIMIT** clause

Joins

Product (pname, price, category, manufacturer)

Company (cname, stockPrice, country)

Find all products under \$200 manufactured in Japan;
return their names and prices.

```
SELECT P.pname, P.price
FROM Product P, Company C
WHERE P.manufacturer=C.cname AND C.country='Japan'
AND P.price <= 200
```

```
SELECT P.pname, P.price
FROM Product P JOIN Company C ON P.manufacturer=C.cname
WHERE C.country='Japan' AND P.price <= 200
```

Semantics of SQL Queries

```
SELECT a1, a2, ..., ak  
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn  
WHERE Conditions
```

```
Answer = {}  
for x1 in R1 do  
    for x2 in R2 do  
        .....  
            for xn in Rn do  
                if Conditions  
                    then Answer = Answer ∪ {(a1, ..., ak)}  
return Answer
```

Aggregation

```
SELECT avg(price)
FROM Product
WHERE maker='Toyota'
```

```
SELECT count(*)
FROM Product
WHERE year > 1995
```

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

Grouping and Aggregation

Purchase(product, price, quantity)

Find total quantities for all sales over \$1, by product.

```
SELECT    product, Sum(quantity) AS TotalSales
FROM      Purchase
WHERE     price > 1
GROUP BY  product
```

Let's see what this means...

Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.
2. Group by the attributes in the **GROUPBY**
3. Compute the **SELECT** clause:
grouped attributes and aggregates.

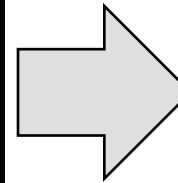
1&2. FROM-WHERE-GROUPBY

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

WHERE price > 1

3. SELECT

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10



Product	TotalSales
Bagel	40
Banana	20

What can go in SELECT clause?
Will return ONE TUPLE per group

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```

HAVING Clause

Same query as earlier, except that we consider only products that had at least 30 sales.

```
SELECT    product, sum(price*quantity)
FROM      Purchase
WHERE     price > 1
GROUP BY  product
HAVING    Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

WHERE vs HAVING

- WHERE condition is applied to individual rows
 - The rows may or may not contribute to the aggregate
 - No aggregates allowed here
- HAVING condition is applied to the entire group
 - Entire group is returned, or not at all
 - May use aggregate functions in the group

General form of Grouping and Aggregation

SELECT	S
FROM	R_1, \dots, R_n
WHERE	C1
GROUP BY	a_1, \dots, a_k
HAVING	C2

S = may contain attributes a_1, \dots, a_k and/or any aggregates but **NO OTHER ATTRIBUTES**

C1 = is any condition on the attributes in R_1, \dots, R_n

C2 = is any condition on aggregate expressions and on attributes a_1, \dots, a_k

Semantics of SQL With Group-By

SELECT	S
FROM	R_1, \dots, R_n
WHERE	C1
GROUP BY	a_1, \dots, a_k
HAVING	C2

Evaluation steps:

1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes a_1, \dots, a_k
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
 - A SELECT clause
 - A FROM clause
 - A WHERE clause
- Rule of thumb: avoid writing nested queries when possible; keep in mind that sometimes it's impossible

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Existential quantifiers

Find all companies that make some products with price < 200

Using **EXISTS**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  EXISTS (SELECT *
               FROM Product P
               WHERE C.cid = P.cid and P.price < 200)
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Existential quantifiers

Find all companies that make some products with price < 200

Using **IN**

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                  FROM Product P
                  WHERE P.price < 200)
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Existential quantifiers

Find all companies that make some products with price < 200

Using **ANY**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 > ANY (SELECT price
                  FROM Product P
                  WHERE P.cid = C.cid)
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Existential quantifiers

Find all companies that make some products with price < 200

Now let's unnest it:

```
SELECT DISTINCT C.cname
FROM   Company C, Product P
WHERE  C.cid= P.cid and P.price < 200
```

Existential quantifiers are easy ! 😊

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Universal quantifiers

Find all companies that make only products with price < 200

same as:

Find all companies whose products all have price < 200

Universal quantifiers are hard ! ☹️

Subqueries in WHERE

1. Find *the other* companies: i.e. s.t. some product ≥ 200

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                 FROM Product P
                 WHERE P.price >= 200)
```

2. Find all companies s.t. all their products have price < 200

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  C.cid NOT IN (SELECT P.cid
                    FROM Product P
                    WHERE P.price >= 200)
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Universal quantifiers

Find all companies that make only products with price < 200

Using **EXISTS**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  NOT EXISTS (SELECT *
                  FROM Product P
                  WHERE P.cid = C.cid and P.price >= 200)
```

Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Universal quantifiers

Find all companies that make only products with price < 200

Using **ALL**:

```
SELECT DISTINCT C.cname
FROM   Company C
WHERE  200 > ALL (SELECT price
                  FROM Product P
                  WHERE P.cid = C.cid)
```


Can we unnest the *universal quantifier* query ?

- A query Q is **monotone** if:
 - Whenever we add tuples to one or more of the tables...
 - ... the answer to the query cannot contain fewer tuples
- Fact: all unnested queries are monotone
 - Proof: using the “nested for loops” semantics
- Fact: Query with universal quantifier is not monotone
- Consequence: we cannot unnest a query with a universal quantifier