SQL Part II

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

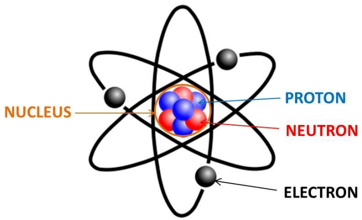
```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

Key concepts

- JOINs
- Aggregation & GROUP BY

Reminder on schemas



Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Students(<u>sid</u>: *string,* name: *string*, gpa: *float*)

Enrolled(<u>student_id</u>: *string, <u>cid</u>*: *string*, grade: *string*)

We'll use different Tables/tuples, for examples
to build ideas

Data about local areas (for real-world examples)

SolarPanel(<u>region_name</u>*: string,* kw_total: *float*, carbon_offset_ton_metrics: *float, …*)

Census(<u>zipcode</u>: *string, population*: *int*, ...)

Pollution(<u>zipcode</u>: string, Particle_count: int…)

BikeShare(<u>zipcode</u>: string, trip_origin: float, trip_end: float, …)

...

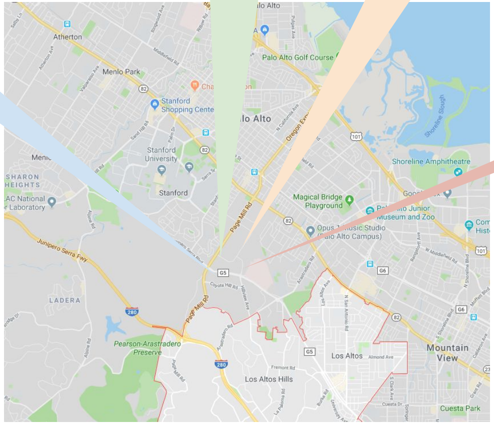
Option 1: ‘Good’ tables, with 10s-100s of columns

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57



Option 2: ‘Frankenstein Table’ (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305						35.2	122.13
94305		????				35.2	122.1
94305						35.1	122.12
94305						...	...

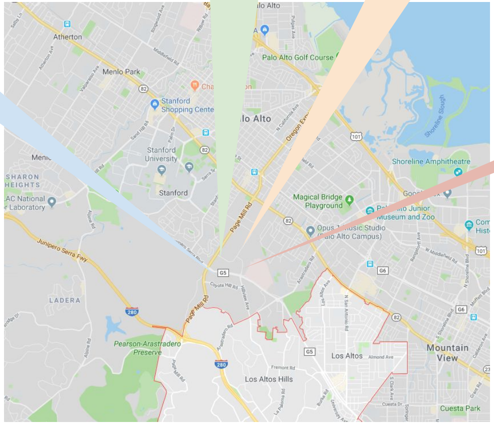
Option 1: A few “good” tables (with 10s of columns)

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57



Trade offs?

- Reads? Writes?
- 100s - thousands of applications reading/writing data

⇒ 1 column? all columns? [hybrid?]

Option 2: ‘FrankenTable’ (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305	14.4	29		14301	40	35.2	122.13
94305	14.4	29		14301	40	35.2	122.1
94305	14.4	29		14301	40	35.1	122.12

Assume

(for now)

- Assume we have a set of "good" tables
  - How do we "connect" (join) those tables?
  - Next 2 weeks

- Related important question
  - How to break up a "Franken" Table into "good" Tables? (i.e, design "good" schema)
  - Study in Week 8, 9

Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Note: we will often omit column types in schema definitions for brevity, but assume columns are always atomic types

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

Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

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

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

A <u>**join**</u> between tables returns all unique combinations of their tuples **which meet specified join condition**

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

```
SELECT PName, Price
FROM   Product
JOIN   Company
ON     Manufacturer = Cname
WHERE  Price <= 200
       AND Country='Japan'
```

A few more later on

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	$19	Gadgets	GizmoWorks
Powergizmo	$29	Gadgets	GizmoWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

CName	Stock Price	Countr y
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
        AND Country='Japan'
        AND Price <= 200
```

PName	Price
SingleTouch	$149

Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)

Company(name, address)

```
1. SELECT DISTINCT name, address
2. FROM  Person, Company
3. WHERE     worksfor = name
```

Which “address” does this refer to**?**

Which name”s??

Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)

Company(name, address)

Both equivalent ways to resolve variable ambiguity

```
SELECT DISTINCT Person.name, Person.address
FROM            Person, Company
WHERE           Person.worksfor = Company.name
```

```
SELECT DISTINCT p.name, p.address
FROM            Person p, Company c
WHERE           p.worksfor = c.name
```

Semantics of JOINs

SELECT $x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions(x_1,…, x_n)

Answer = {}
for x_1 **in** R_1 **do**
  for x_2 **in** R_2 **do**
  …..
    for x_n **in** R_n **do**
    if Conditions(x_1,…, x_n)
      then Answer = Answer $\cup$ {($x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$)}
return Answer

Note:
This is a *multiset* union

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

Recall: Cross product (R X S) is the set of all unique tuples in R,S
Ex: {a,b,c} X {1,2}
  = {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

- Take **cross product**

  V = R x S

- Apply **selections/conditions**

  Y = {(r, s) in V | r.A == s.B}

- Apply **projections** to get final output

  Z = (y.A) for y in Y

= Filtering!

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

An example of SQL semantics

```sql
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

R

A
1
3

S

B	C
2	3
3	4
3	5

Cross Product

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

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Output

A
3
3

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the covers

A Subtlety about Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Find all countries that manufacture some product in the ‘Gadgets’ category.

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=CName AND
Category=‘Gadgets’
```

A Subtlety about Joins

Product

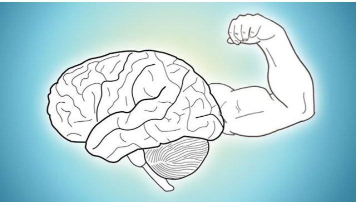
PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
   AND Category='Gadgets'
```

Country
USA ?
USA ?



What is the Problem? What is the Solution?

Ans? DISTINCT

Key concepts

- JOINs
- Aggregation & GROUP BY

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

Aggregation: COUNT

COUNT counts all tuples (including duplicates), unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

versus

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Example 1

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

→ 50
(= 1*20 + 1.50*20)

Example 2

```
SELECT SUM(price * quantity)
FROM   Purchase
```

→ 65
(= 1*20 + 1.50*20 + 0.5*10 + 1*10)

Grouping and Aggregation

What GROUPings are possible?

- Type, Size, Color
- Number of holes
- Combination?

What GROUPings are possible?

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Possible Groups

- Product? (e.g. SUM(quantity) by product) # product units sold
- Date? (e.g., SUM(price*quantity) by date) # daily sales
- Price?
- Product, Date?
- <various column combinations>

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let's see what this means…

Grouping and Aggregation

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Semantics of the query:

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

(Why is Select on top? Similar to Functions -- f ⇒ output on top)

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM →

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the **GROUP BY**

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT →

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Why?

- S = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1,\ldots,R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

SELECT	S
FROM	$R_1,\ldots,R_n$
WHERE	C_1
GROUP BY	$a_1,\ldots,a_k$
HAVING	C_2

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may need to compute aggregates)
4. Compute aggregates in S and return the result

NULL and NOT NULL

- To say “don’t know the value” we use NULL
  NULL has (sometimes painful) semantics, more detail later

Students(sid:string, name:string, gpa: float)

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL,
e.g., “name” in this table

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL

- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
  - Etc.

- The schema specifies for each attribute if can be null (*nullable* attribute) or not

- How does SQL cope with tables that have NULLs?

Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE	**=**	**0**
UNKNOWN	**=**	**0.5**
TRUE	**=**	**1**

  - If x= NULL then x == “Joe” is UNKNOWN (Is x equal to ‘Joe’?)

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190)
```

Won’t return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
   OR age IS NULL
```

Now it includes all Persons!

RECAP: Joins

By default, joins in SQL are **"inner joins"**:

```
Product(name, category)
Purchase(prodName, store)
```

```sql
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

```sql
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

[Like Below]

```sql
SELECT Product.name, Purchase.store
FROM   Product
  INNER JOIN Purchase
            ON Product.name = Purchase.prodName
```

Inner Joins + NULLS = Lost data?

By default, joins in SQL are **"inner joins"**:

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

Product

Name	Category
Iphone	Media
Roomba	Cleaner
Ford Pinto	Car
Tesla	Car

Purchase

ProdName	Store
Iphone	Apple Store
Tesla	Tesla Store

However: Products that never sold (with no Purchase tuple) will be lost!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
  - A LEFT OUTER JOIN will return a tuple (a, NULL)!
- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase ON
        Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

LEFT OUTER JOIN

Product

name	category
iphone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	car
iPhone	Apple store

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
            ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	car
Ford Pinto	NULL

Other Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Key concepts

- JOINs
- Aggregation & GROUP BY

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

SQL - Part 3

Today's Lecture

- SQL Sets operators
- When are two queries equivalent?
- How does SQL work?
  - Intro to Relational Algebra
  - A basic RDBMS query optimizer

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

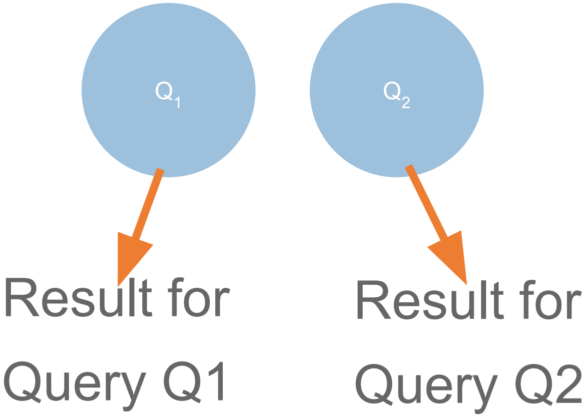
```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

What you will learn about in this section

1. Multiset operators in SQL

2. Nested queries

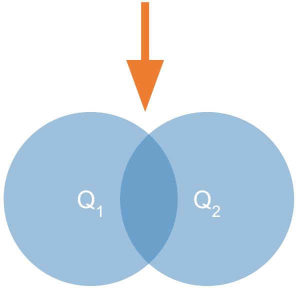
Notation



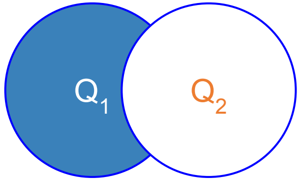
Explicit Set Operators:

INTERSECT, UNIONS on results of Queries Q1, Q2

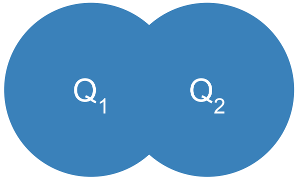
```
SELECT R.A
FROM   R, S           Q1
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T           Q2
WHERE  R.A=T.A
```




```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```



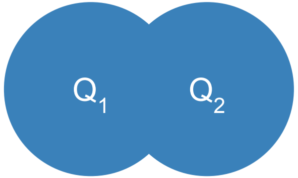

```
SELECT  R.A
FROM   R, S
WHERE  R.A=S.A
UNION
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```



By DEFAULT:
SQL retains Set semantics for Set Operators

Use ALL for Multiset

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION ALL
SELECT  R.A
FROM    R, T
WHERE   R.A=T.A
```



ALL indicates Multiset operations

Recall Multisets

$\lambda(X)$= *"Count of tuple in X"*
(Items not listed have implicit count 0)

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)

Equivalent Representations of a **Multiset**

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

Note: In a set all counts are {0,1}.

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cap$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

$$\lambda(Z) = min(\lambda(X), \lambda(Y))$$

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cup$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

Key concept

SQL is **compositional**

Can construct powerful query chains (e.g., f(g(...(x)))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

Q1 -> Q2 -> … Qn -> Result Table

⇒ Including on same table (e.g., self correlation)

Nested queries: Sub-queries Return Relations

Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)

```
SELECT pr.maker
FROM   Purchase p, Product pr
WHERE  p.product = pr.name
       AND p.buyer = 'Mickey')
```

"

- Companies making products bought by Mickey"
- Location of companies?

"

Subqueries Return Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex: Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(
    SELECT price
    FROM   Product
    WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by "Gizmo-Works"

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
  AND EXISTS(
        SELECT p2.name
        FROM   Product p2
        WHERE  p2.maker <> 'Gizmo-Works'
          AND p1.name = p2.name)
```

<> means !=

Find 'copycat' products, i.e. products made by competitors with the same names as products made by "Gizmo-Works"

Note the scoping of the variables!

Example: Complex Correlated Query

Product(name, price, category, maker, year)

```
SELECT DISTINCT  x.name, x.maker
FROM             Product AS x
WHERE            x.price > ALL(
        SELECT y.price
        FROM   Product AS y
        WHERE  x.maker = y.maker
               AND y.year < 2010)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 2010

Can be very powerful (also much harder to optimize)

Key concept

SQL is compositional

Can construct powerful query chains (e.g., f(g(x)))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

⇒ Including on same table (e.g., self correlation)

Equivalent SQL queries

Key concept

Can write different SQL queries to solve same problem

Key:

-- Be careful with sets and multisets

-- Go back to semantics (1st principles)

Example1: Two equivalent queries?

Product(name, price, company)
Company(name, city)

Find all companies with products having price < 100

VS

Find all companies that make only products with price < 100

'Similar' but non-equivalent'

```
SELECT DISTINCT Company.cname
FROM    Company, Product
WHERE  Company.name = Product.company
       AND Product.price < 100
```

```
SELECT DISTINCT Company.cname
FROM     Company
WHERE   Company.name NOT IN(
          SELECT Product.company
          FROM Product.price >= 100)
```

A universal quantifier is of the form "for all"

Example 2: Headquarters of companies which make gizmos in US AND China

Company(name, hq_city)
Product(pname, maker, factory_loc)

Company

Name	hq_city
X Co.	Seattle
Y Inc.	Seattle

Product

pname	maker	factory_loc
X	X Co.	U.S.
Y	Y Inc.	China

Option 1: With Nested queries

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
    AND name IN (
              SELECT maker
              FROM   Product
              WHERE  factory_loc = ‘US’)
    AND name IN (
              SELECT maker
              FROM   Product
              WHERE  factory_loc = ‘China’)
```

Option 2: With Intersections

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
       AND factory_loc=‘US’
INTERSECT
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
       AND factory_loc=‘China’
```

X Co has a factory in the US (but not China)
Y Inc. has a factory in China (but not US)
But Seattle is returned by the query!
⇒ Option 1 and Option 2 are **NOT** equivalent

Example3: Are these equivalent? [harder version]

```
SELECT c.city
FROM   Company c, Product pr, Purchase p
WHERE  c.name = pr.maker
   AND  pr.name = p.product
   AND  p.buyer = 'Mickey'
```

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
       SELECT pr.maker
       FROM   Purchase p, Product pr
       WHERE  pr.name = p.product
              AND p.buyer = 'Mickey')
```

Step 1: Construct some examples

Company			Product			Purchase	
Name	City		Name	Maker		Product	Buyer
Tesla	Palo Alto		Model X	Tesla		Kindle	Mickey
Amazon	Seattle		Kindle	Amazon		Model X	Mickey
			Kindle Fire	Amazon		Kindle Fire	Mickey
			Books	Amazon		Book	Mickey

Step 2: Test examples

Seattle
Palo Alto
Seattle
Seattle

Palo Alto
Seattle

Beware of duplicates!

Example3: Are these equivalent?

```
SELECT c.city
FROM   Company c, Product pr, Purchase p
WHERE  c.name = pr.maker
   AND  pr.name = p.product
   AND  p.buyer = 'Mickey'
```

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
       SELECT pr.maker
       FROM   Purchase p, Product pr
       WHERE  p.name = pr.product
              AND p.buyer = 'Mickey')
```

Fix duplicates!

```
SELECT DISTINCT c.city
FROM   Company c, Product pr, Purchase p
WHERE  c.name = pr.maker
    AND  pr.name = p.product
    AND  p.buyer = 'Mickey'
```

```
SELECT DISTINCT c.city
FROM   Company c
WHERE  c.name  IN (
    SELECT pr.maker
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
         AND p.buyer = 'Mickey')
```

Now they are equivalent (both use set semantics)

Example4: Are these equivalent?

Students(sid, name, gpa)
Enrolled(student_id, cid, grade)

- Find students enrolled in > 5 classes

Attempt 1: with nested queries

```
SELECT DISTINCT Students.sid
FROM    Students
WHERE (
    SELECT COUNT (cid)
    FROM    Enrolled
    WHERE  Students.sid = Enrolled.student_id
    ) > 5
```

SQL by
a novice

Attempt 2: with GROUP BYs

```
SELECT    Students.sid
FROM      Students, Enrolled
WHERE     Students.sid = Enrolled.student_id
GROUP BY Students.sid
HAVING    COUNT(Enrolled.cid) > 5
```

1. SQL by an expert
2. No need for DISTINCT: automatic from GROUP BY

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested:* How many times do we do a SFW query over all of the Enrolled relations?

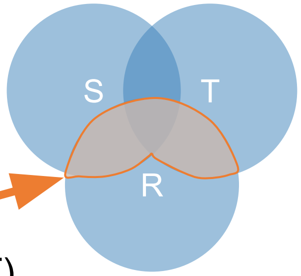
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **<u>much</u>** more efficient!

Example 5: An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?



Computes R ∩ (S ∪ T)

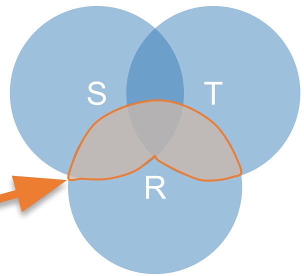
But what if S = φ?

Go back to the semantics!

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

$R \cap (S \cup T)$



Semantics:

1. Take cross-product

Joins / cross-products are just nested for loops (in simplest implementation)!

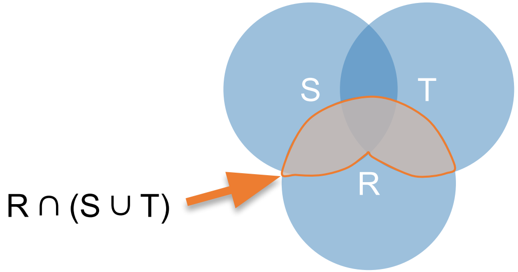
2. Apply selections / conditions

If-then statements!

3. Apply projection

What does this look like in Python?

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```




```
output = {}

for r in R:
   for s in S:
      for t in T:
         if r['A'] == s['A'] or r['A'] == t['A']:
            output.add(r['A'])
return list(output)
```

Can you see now what happens if S = []?

Key concept

Equivalent SQL queries

Can write different SQL queries to solve same problem

Key:

-- Be careful with sets and multisets

-- Go back to semantics (1st principles)

Basic SQL Summary

SQL is a high-level declarative language for manipulating data (DML)

- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

How does it work?

RDBMS Architecture

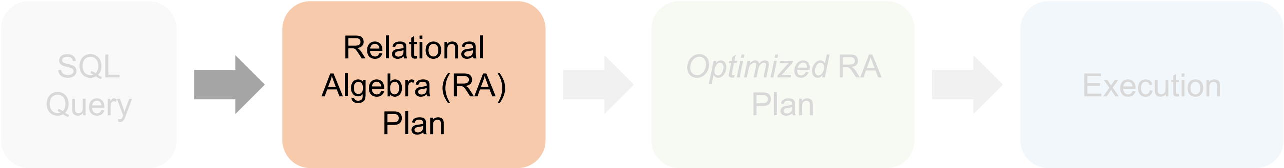
How does a SQL engine work ?

SQL Query → Relational Algebra (RA) Plan → *Optimized* RA Plan → Execution

SQL Query	Relational Algebra (RA) Plan	*Optimized* RA Plan	Execution
Declarative query (from user)	Translate to relational algebra expression	*Find logically equivalent- but more* ***cost-efficient****- RA expression*	Execute each operator of the optimized plan!

RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!

Relational Algebra (RA)

Five **basic** operators:

1. Selection: σ
2. Projection: Π
3. Cartesian Product: $\times$
4. Union: $\cup$
5. Difference: -

Derived or auxiliary operators:

- Intersection
- Joins: $\bowtie$ (natural, equi-join, semi-join)
- Renaming: ρ

What's an Algebra? Why?

- For ex, in Math

  a) (x + y) + z vs x + y + z

  b) (x + y) + 2*x vs (x + y + 2)*x

- Operators and rules

  - Basic notation for operators ('+', '-', '*', '/', '^' etc.)
  - Association, commutative, …

⇒ Why?

  - What can you reorder, simplify?
  - Express complex equations and expressions, and reason about them

Converting SFW Query to RA

Students(sid,sname,gpa)
People(ssn,sname,address)

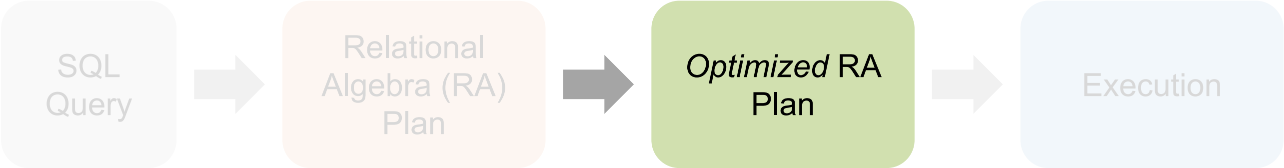
```
SELECT DISTINCT gpa, address
FROM Students S, People P
WHERE gpa > 3.5 AND
          sname = pname;
```

$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

How do we represent this query in RA?

RDBMS Architecture

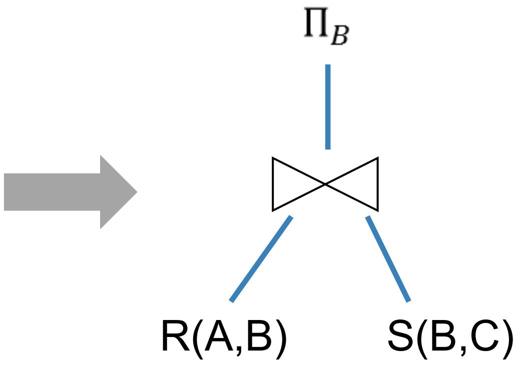
How does a SQL engine work ?



We'll look at how to then optimize these plans now

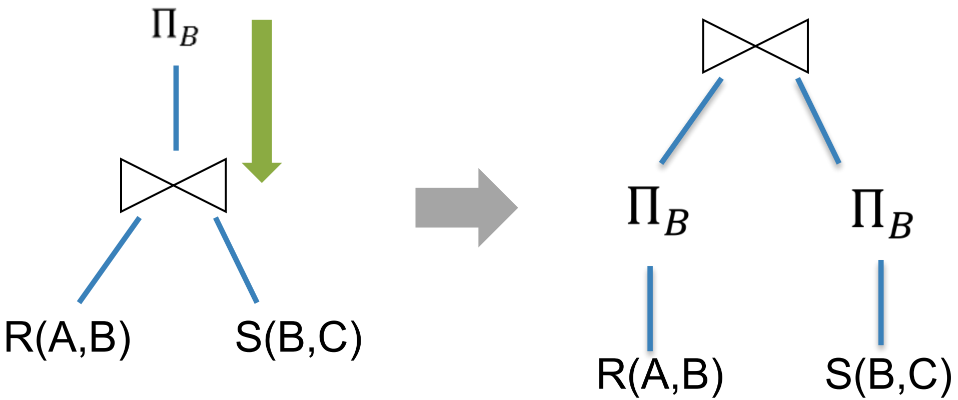
Visualize the plan as a tree

$$\Pi_B(R(A,B) \bowtie S(B,C))$$



Bottom-up tree traversal = order of operation execution!

One simple plan -- “Push down” projection



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

Why might we prefer this plan?

Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections** and **projections”**

- **Intuition:** We will usually have fewer tuples in a plan.

  Exceptions

  - Could fail if the selection condition is very expensive (e.g, run image processing algorithm)
  - Projection could be a waste of effort, but more rarely

⇒ Cost-based Query Optimizers (e.g., Postgres/ BigQuery/ MySQL optimizers, SparkSQL’s Catalyst)

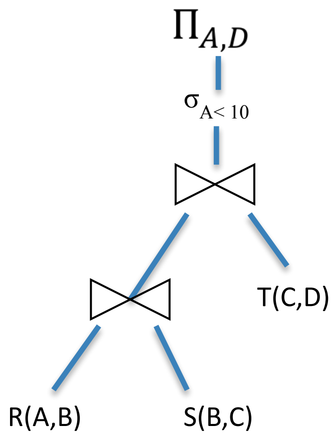
Optimizing the SFW RA Plan

Translating to RA

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,T.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



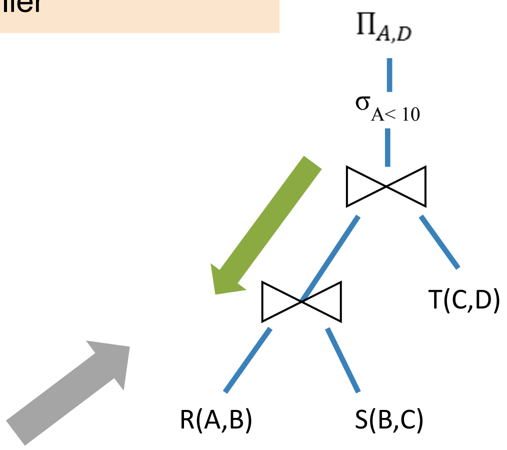
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$

Push down selection on A so it occurs earlier



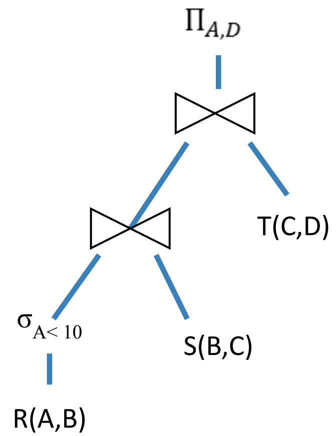
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$

Push down selection on A so it occurs earlier



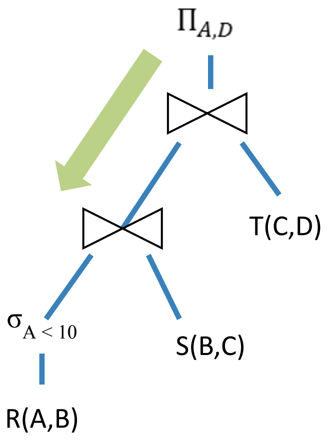
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$

Push down projection so it occurs earlier



Optimizing RA Plan

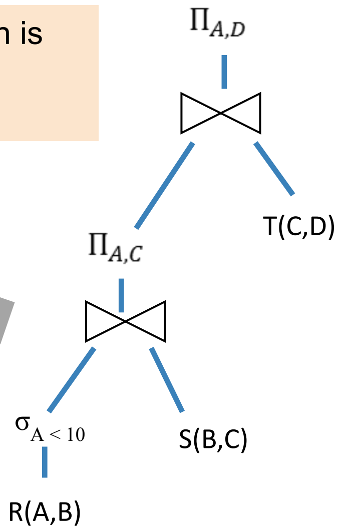
We eliminate B earlier!

In general, when is an attribute not needed…?

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}\left(T \bowtie \Pi_{A,c}(\sigma_{A<10}(R) \bowtie S)\right)$$



Basic RA commutators

- Push **projection** through **(1) selection**, **(2) join**
- Push **selection** through **(3) selection, (4) projection, (5) join**
- *Also:* Joins can be re-ordered!

⇒ Note that this is not an exhaustive set of operations

This covers *local re-writes; global re-writes possible but much harder*

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

Takeaways

- This process is called logical optimization
- Many equivalent plans used to search for “good plans”
- Relational algebra is a simple and elegant abstraction