



SQL Part III

SQL - Part 3

Today's Lecture

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

Preview SQL queries

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 t2

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 %, _

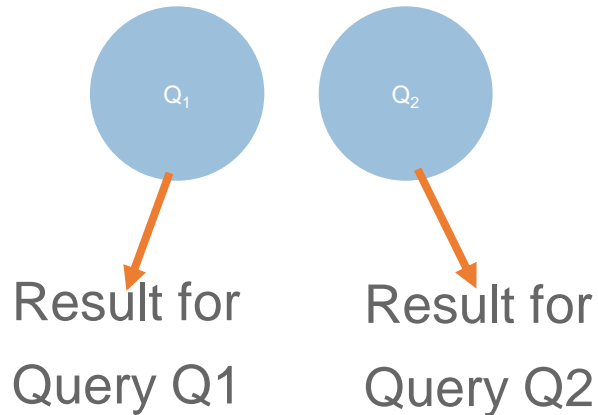
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
Query rows in a list

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



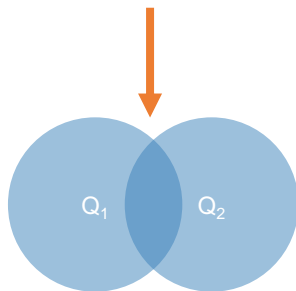
Explicit Set Operators:

INTERSECT, UNIONS on results of Queries Q1, Q2

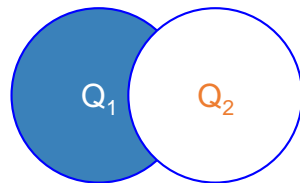
```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
INTERSECT  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

Q1

Q2

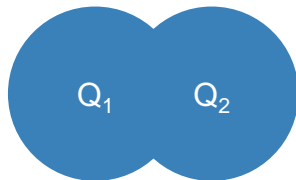


```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
EXCEPT  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```



$\{r.A \mid r.A = s.A\} \setminus \{r.A \mid r.A = t.A\}$

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



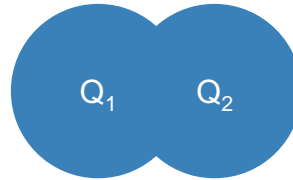
By default:

SQL retains Set semantics for
Set Operators

What if we want duplicates?

ALL indicates 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

Multiset X

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



Equivalent
Representations
of a **Multiset**

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

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

U

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., $f(g(\dots(x)))$)


Inputs / outputs are multisets

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

⇒ 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:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\text{EXISTS } R$

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 < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

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)

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)

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')
```

Note: If we hadn't used DISTINCT here, how many copies of each hq_city would have been returned?

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

Example 3: 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')
```

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!

Example 3: 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)

Example 4: 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?

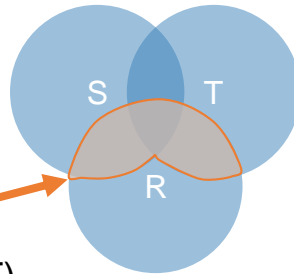
- Attempt #1- *With nested*: How many times do we do a SFW query over all of the Enrolled relations?
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **much** 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 $R \cap (S \cup T)$

But what if $S = \phi$?

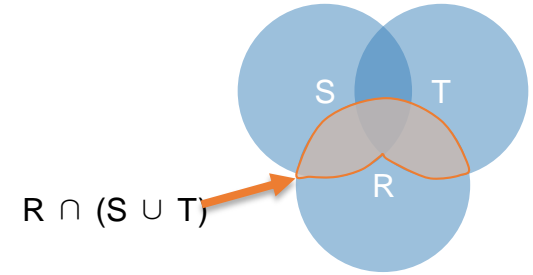
Go back to the semantics!

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
```

Semantics:

1. Take cross-product
1. Apply selections / conditions
1. Apply projection

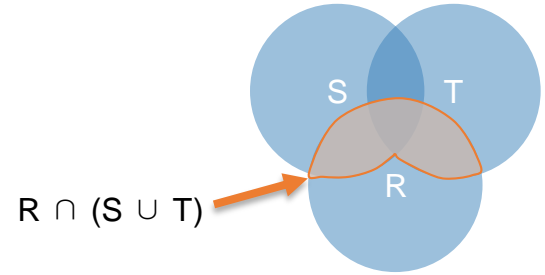


Joins / cross-products are just nested for loops
(in simplest implementation)!

If-then statements!

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 SQL block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed

Preview SQL queries

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 t2

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

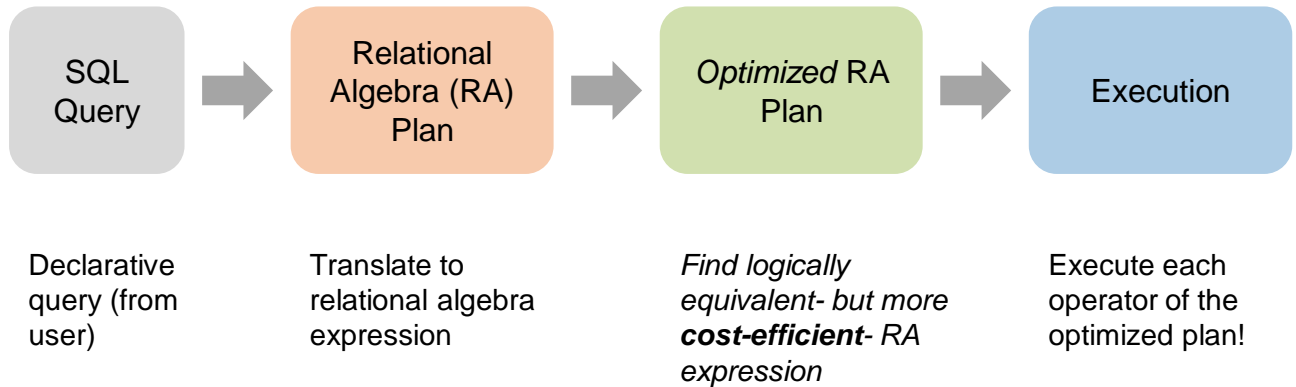
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
Check if values in a table is NULL or not



How does it work?

RDBMS Architecture

How does a SQL engine work ?



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;
```


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

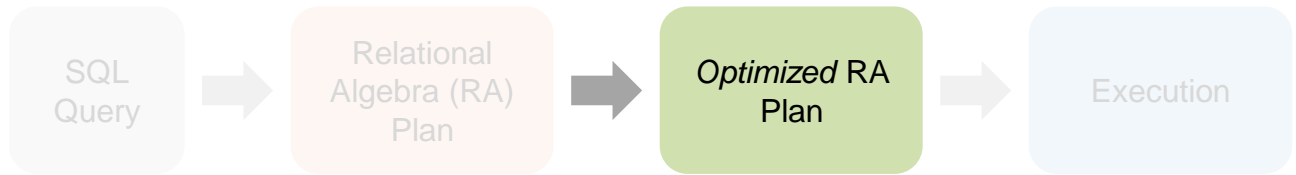
How do we represent this query in RA?

Logical Equivalence of RA Plans

- Given relations $R(A,B)$ and $S(B,C)$:
 - Here, projection & selection commute:
 - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
- What about here?
 - $\sigma_{A=5}(\Pi_B(R)) \stackrel{?}{=} \Pi_B(\sigma_{A=5}(R))$

RDBMS Architecture

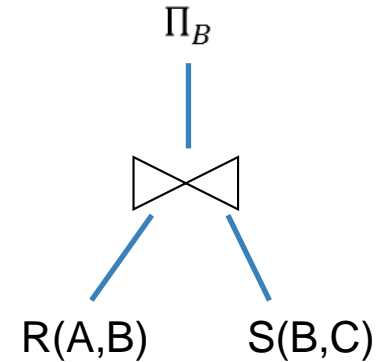
How does a SQL engine work ?



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

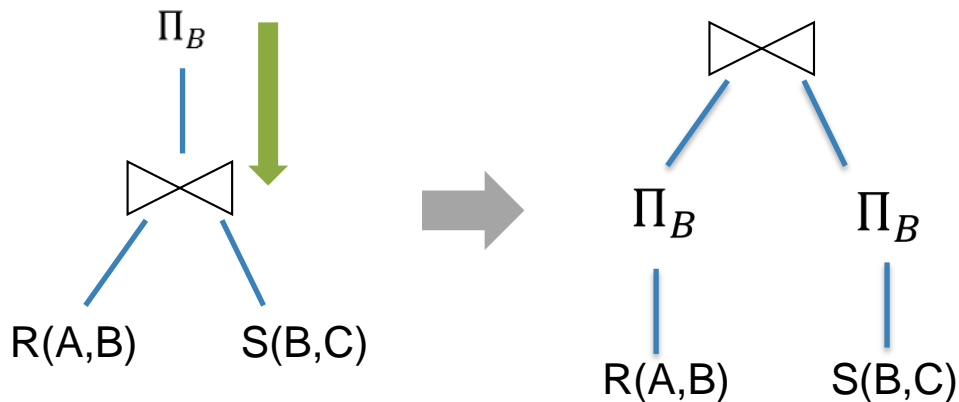
Visualize the plan as a tree

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



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

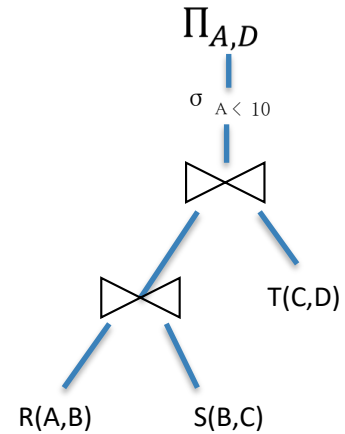
Translating to RA

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

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



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



Optimizing RA Plan

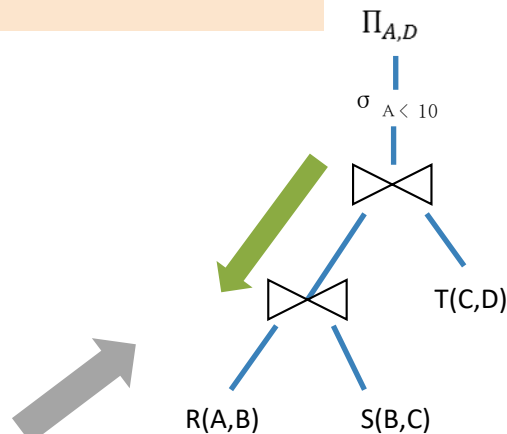
R(A,B) S(B,C) T(C,D)

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

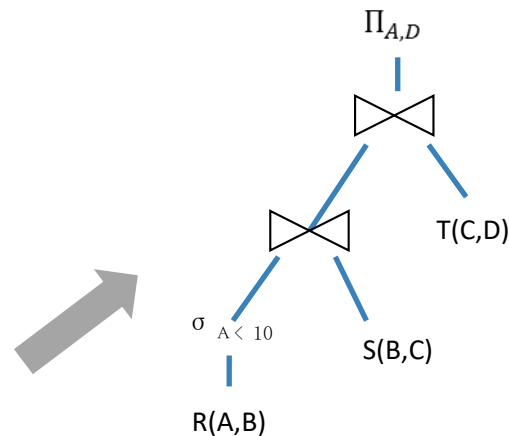
R(A,B) S(B,C) T(C,D)

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



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

Push down selection
on A so it occurs
earlier



Optimizing RA Plan

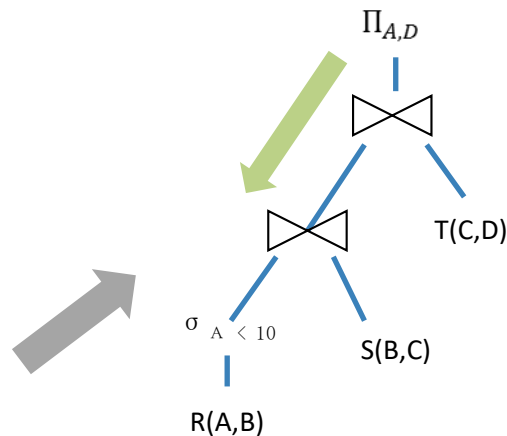
R(A,B) S(B,C) T(C,D)

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



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

Push down
projection so it
occurs earlier



Optimizing RA Plan

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

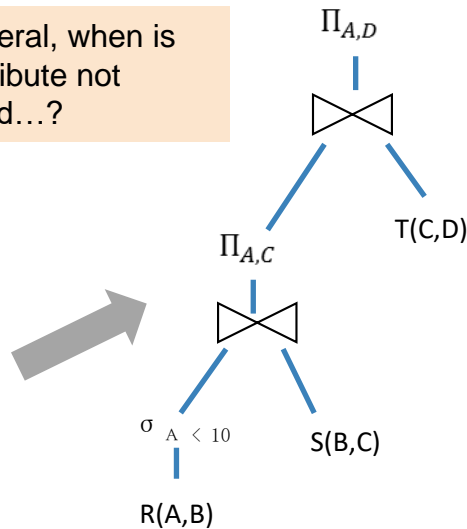
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
AND S.C = T.C
AND R.A < 10;



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

We eliminate B
earlier!

In general, when is
an attribute not
needed...?



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



THANK
YOU!