Databases and Big Data

Query Optimization

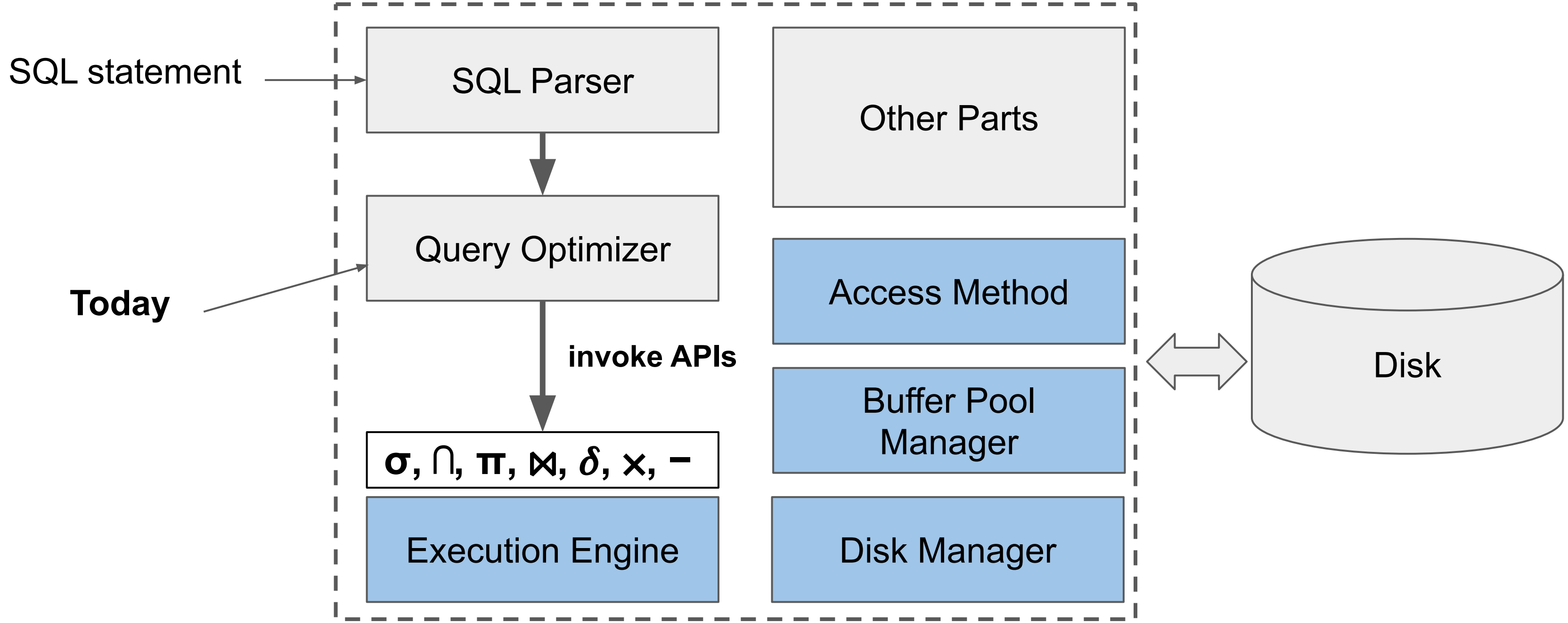
Schedule

Date	Topics	Notes
Week 1	~~Intro~~ ~~Data Model~~	No lab
Week 2	~~SQL~~	
Week 3	~~No-SQL~~ ~~MongoDB~~	**Group project launch**
Week 4	~~Functional Dependencies~~ ~~Normal Form~~	
Week 5	~~Storage~~ ~~Index~~	**Project checkpoint 1**
Week 6	~~Sort, Join~~ Query optimization	**Homework**
Week 7	Recess	

Schedule

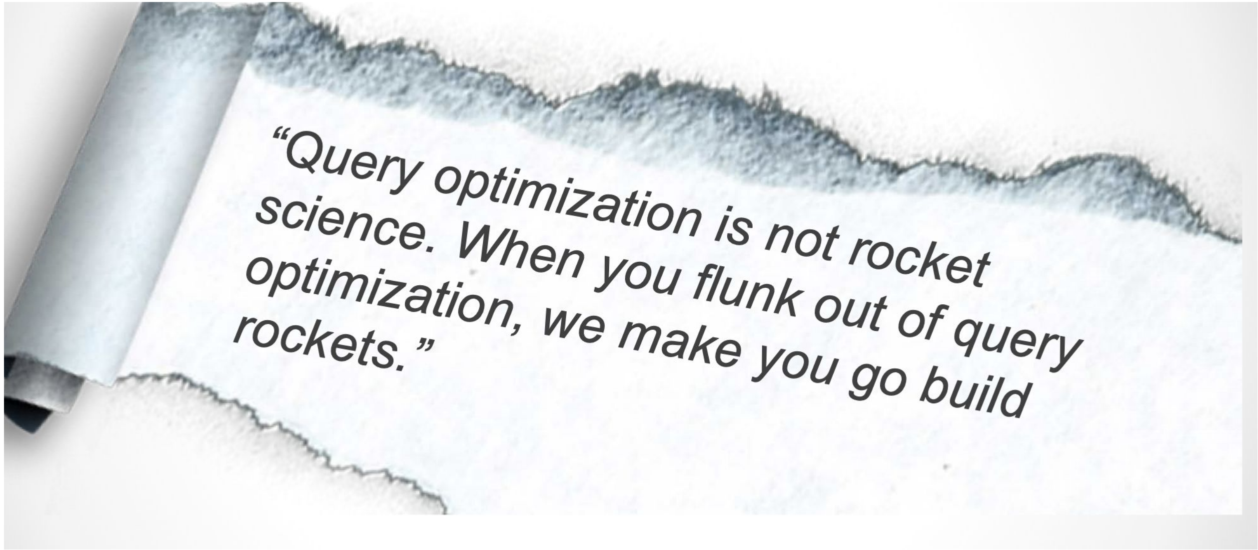
Date	Topics	Notes
Week 7	Recess	
Week 8	Transactions	**In-class quiz**
Week 9	Introduction to Big Data Cloud Computing	**Project checkpoint 2**
Week 10	Hadoop	
Week 11	Spark	
Week 12	Spark Ecosystem	
Week 13	Guest speakers Revision	**In-class quiz** **Project due**
Week 14	Revision Exam	

So Far


SQL statement
SQL Parser
Other Parts
Query Optimizer
Today
Access Method
invoke APIs
Buffer Pool Manager
Disk
σ, ∩, π, ⋈, δ, ×, −
Execution Engine
Disk Manager

Today

- How hard it is to get **it** right



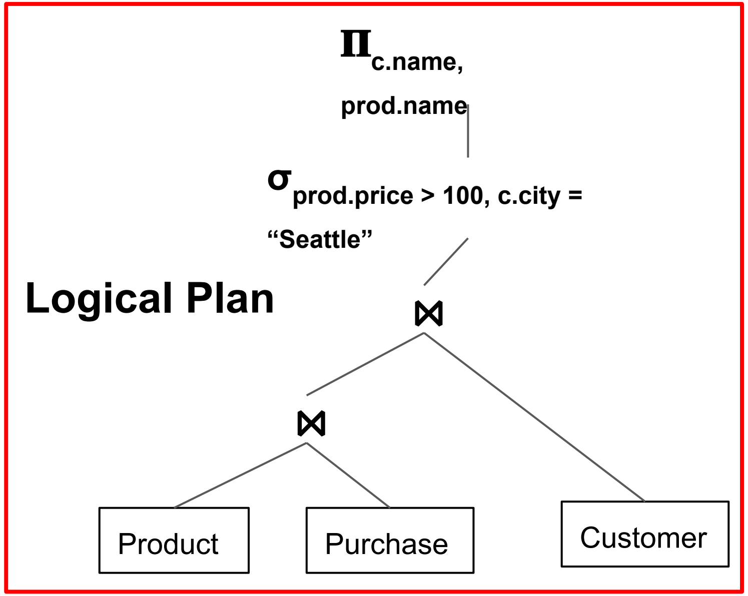
Query Optimization

- Recall

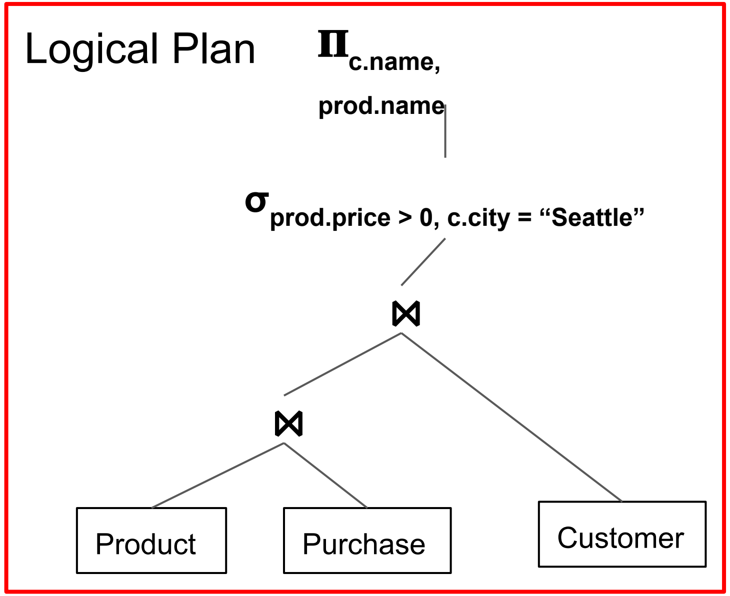
Product (<u>**pid**</u>, name, price)
Purchase (<u>**pid**</u>, cid, store)
Customer (<u>**cid**</u>, name, city)

```
select c.name, prod.name
from customer c, product prod, purchase pur
where prod.pid = pur.pid and c.cid = pur.cid
      and prod.price > 100 and c.city = "Seattle";
```

Find name of the customer in Seatle
who buys anything over $100, and
name of the product he buys



Query Optimization



Week 2

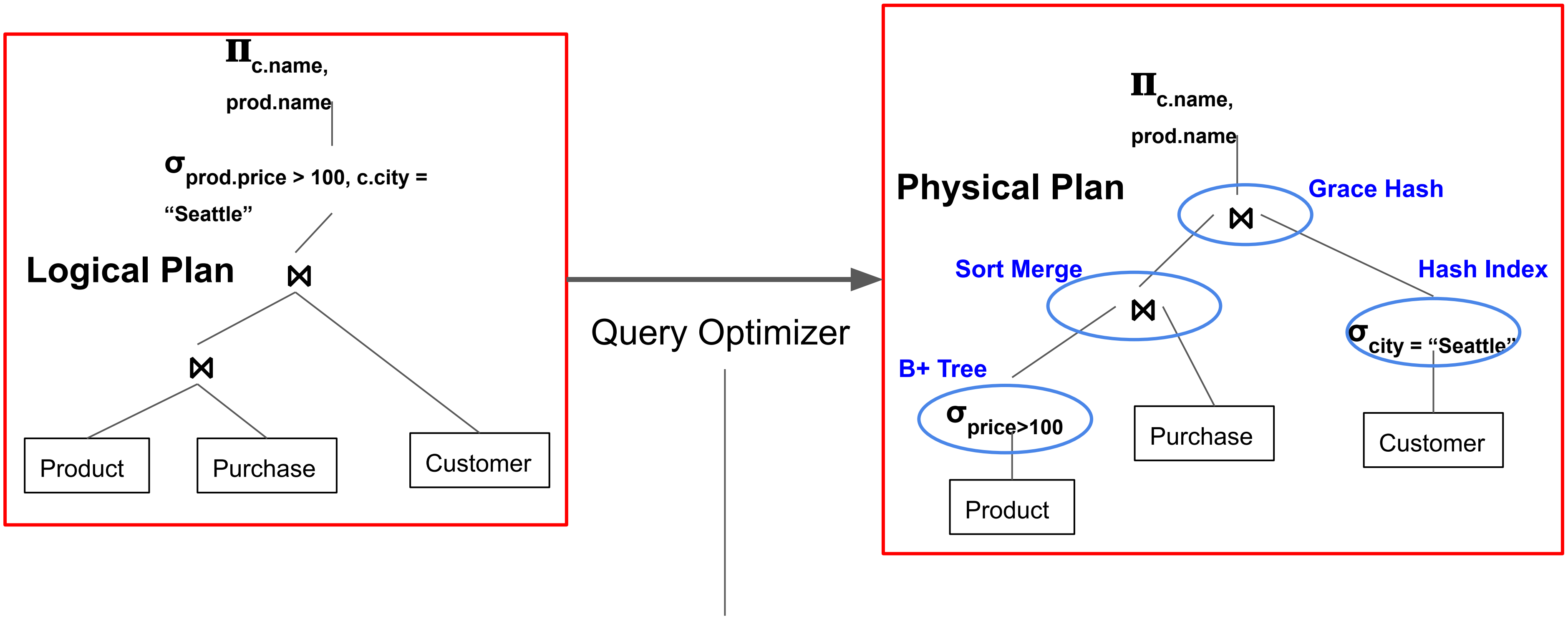
The final piece

- Grace Hash Join
- Nested Loop Join
- Sort Merge Join
- Index Scan
- Heap File Scan
- External Sort

...

Week 5,6

Query Optimization



That's why DBMS is declarative

Query Optimizer

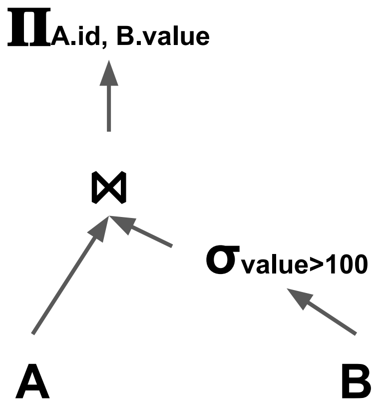
- Turn a logical plan into physical plan
- Questions:
  - How to execute a physical plan?
  - How to enumerate equivalent plans?
  - How to estimate cost of each plan?
  - How to search for the best plan?

Execution Model

Execution Model

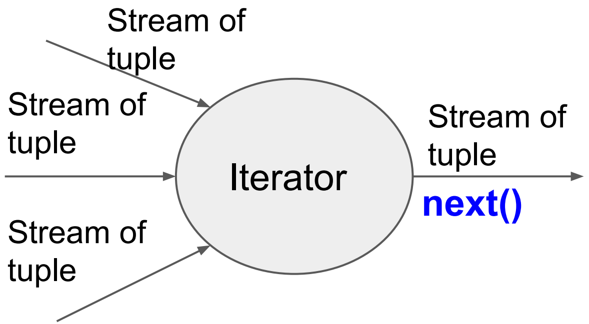
- Given a physical query plan, how does DBMS execute it?
  - Iterator Model
  - Materialization Model
  - Vector Model

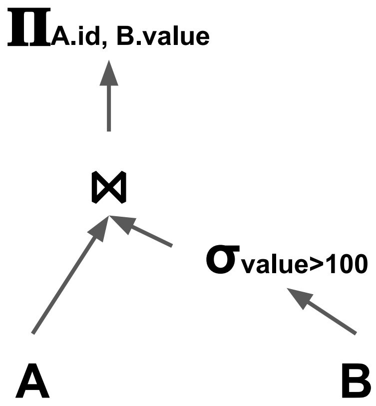
```
select A.id, B.value
from A, B
where A.id = B.id and B.value > 100
```



Execution Model

- Iterator Model
  - Everything in the plan is an <u>Iterator</u>
  - Each operator implement a **next()** method
  - Upstream operator calls next(.) of its children





Execution Model

- Iterator Model

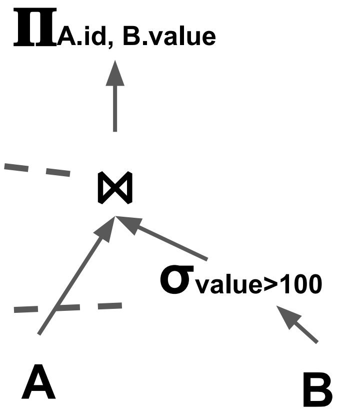
```
for t in child.next():
    output(projection(t))
```

```
for t1 in left.next():
    buildHashTable(t1)
for t2 in right.next():
    if (t1 = probe(t2))
      output(t1 joins t2)
```

```
for t in child.next():
    if condition(t): output(t)
```

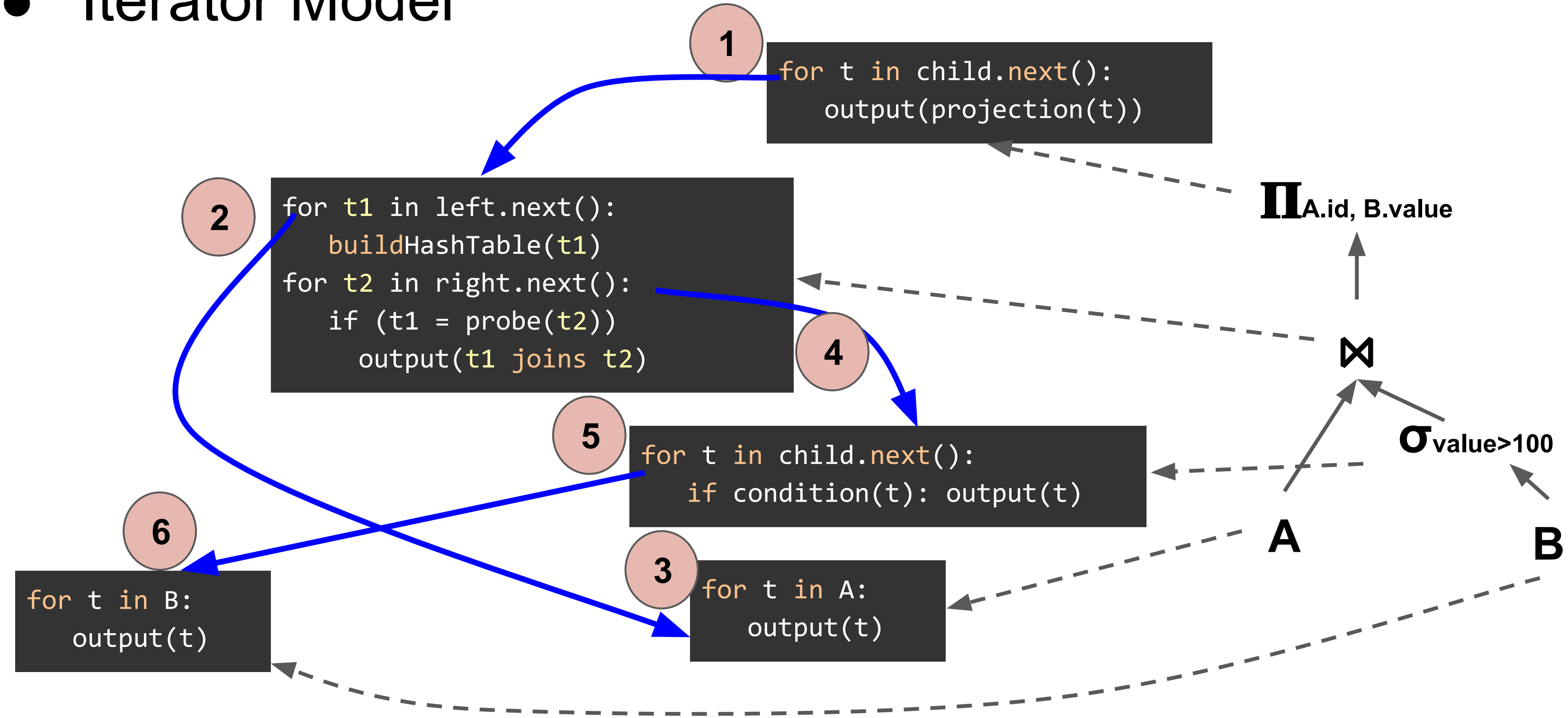
```
for t in B:
    output(t)
```

```
for t in A:
    output(t)
```



Execution Model

- Iterator Model



Execution Model

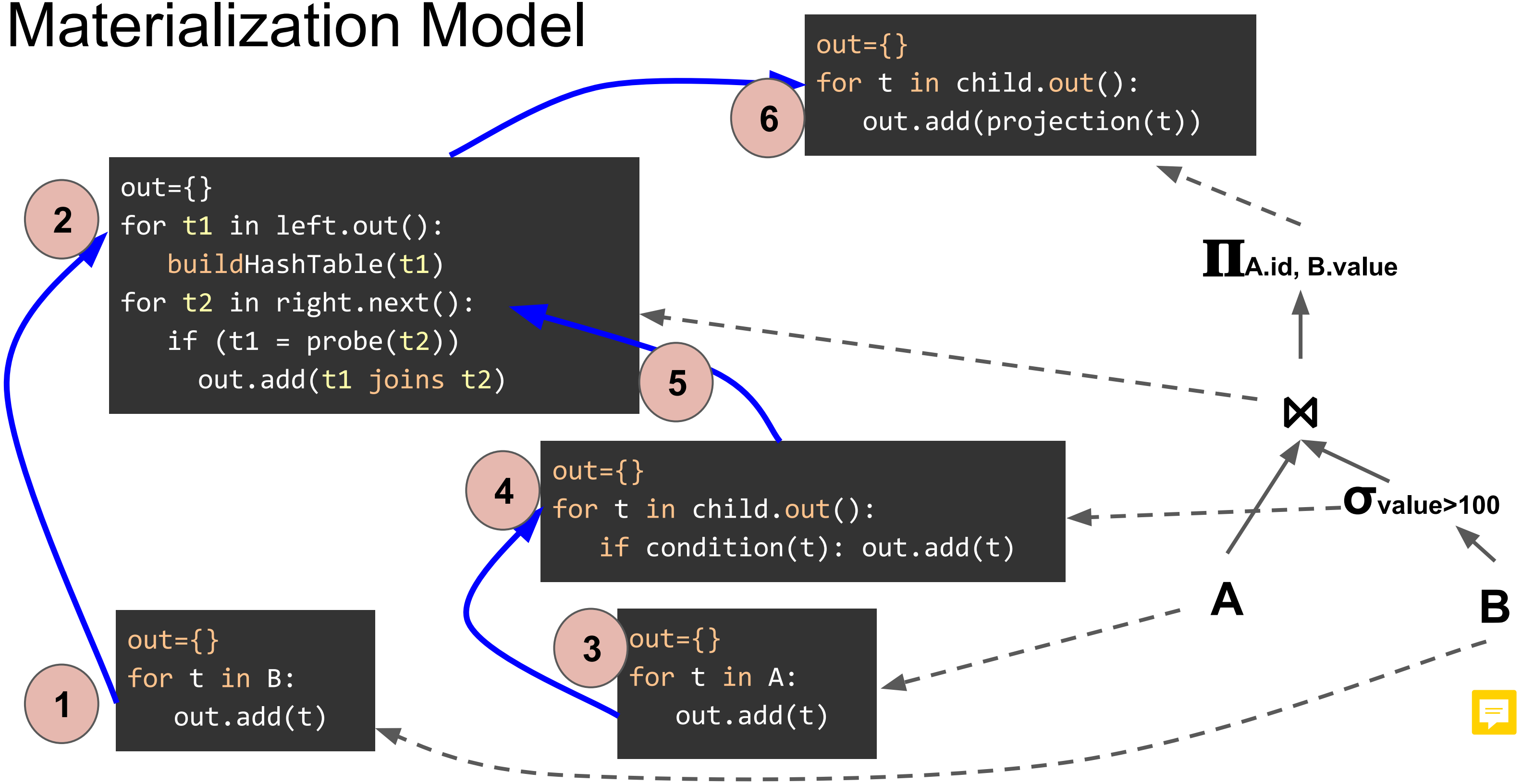
- Iterator Model
  - Data flows bottom up
  - Control (call to next(.)) from top down
- Almost all DBMS use this
  - Simple to implement
  - Allows pipelining
  - Great if only subset of results consumed: LIMIT operator

Execution Model

- Iterator Model
  - But high overhead from method calls
- Materialization Model
  - Process all input, and emit all output **at once**
  - Better than Iterator when immediate results **not too much larger** than final result

Execution Model

- Materialization Model



Execution Model

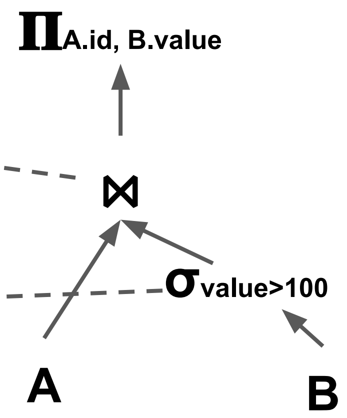
- Vectorized Model
  - Combining Iterator + Materialization
  - Every operator implement next(.)
  - next(.) processes in **batch**
    - Reduce number of invocations in Iterator model

Execution Model

- Vectorized Model

```
out={}
for t in child.out():
   out.add(projection(t))
   if |out| > m: output(out)
```

```
out={}
for t1 in left.out():
   buildHashTable(t1)
for t2 in right.next():
   if (t1 = probe(t2))
     out.add(t1 joins t2)
     if |out| > m: output(out)
```




```
out={}
for t in child.out():
   if condition(t): out.add(t)
   if |out| > m: output(out)
```

```
out={}
for t in B:
   out.add(t)
   If |out| > m: output(out)
```

```
out={}
for t in A:
   out.add(t)
   if |out| > m: output(out)
```

Execution Model

Iterator Model:

- Direction: top down
- One tuple at a time
- General purpose

Vectorized:

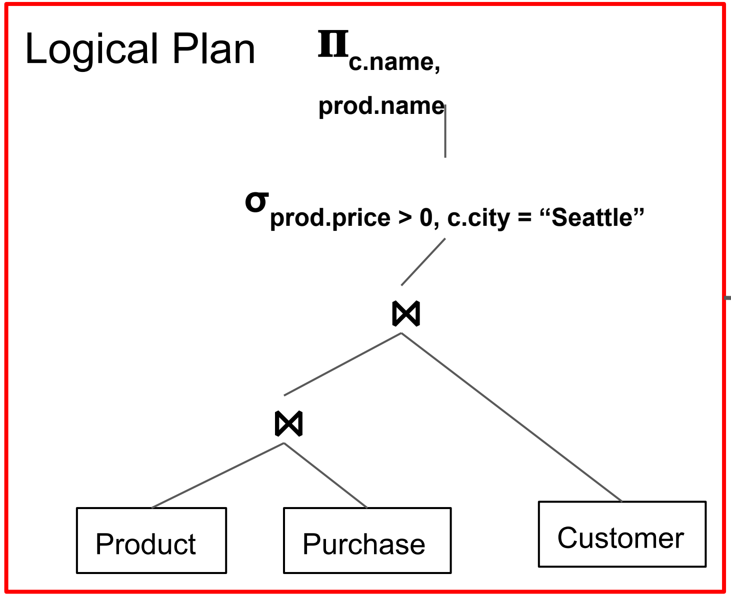
- Direction: top down
- Batch of tuple at a time
- Read-heavy workloads

Materialization:

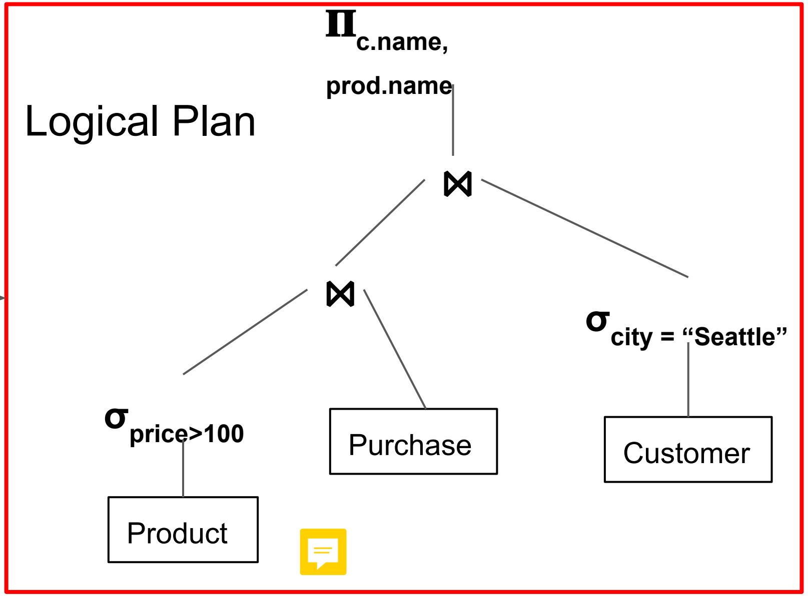
- Direction: bottom up
- Entire output set at a time
- Write-heavy workloads

Query Rewriting

Query Rewriting



Equivalent Plan



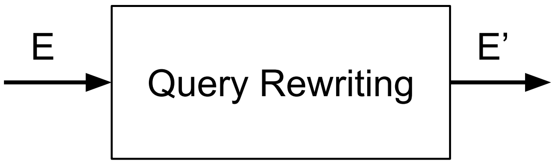
$\Pi(\sigma(\text{Product} \bowtie \text{Purchase}) \bowtie \text{Customer})$

$\Pi(\sigma(\text{Product}) \bowtie \text{Purchase}) \bowtie \sigma(\text{Customer}))$

Query Rewriting

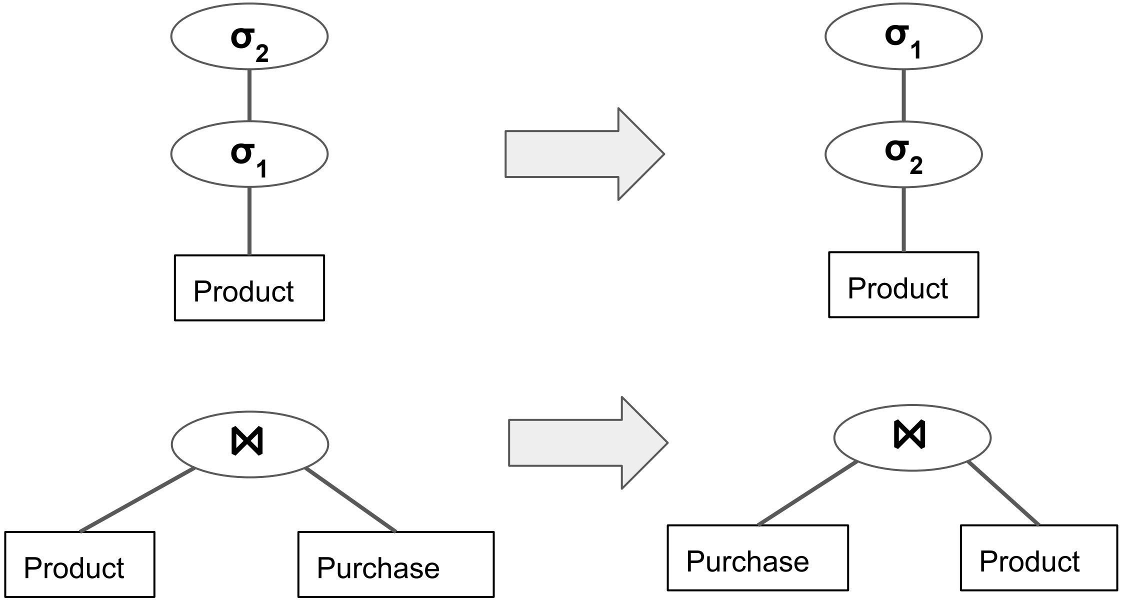
- E, E' are relational expression
  - e.g. $\Pi(\sigma(\text{Product} \bowtie \text{Purchase}) \bowtie \text{Customer})$
- E, E' are equivalent:
  - Let $\mathcal{I}$ be all possible database instances

$$\forall I \in \mathcal{I}. E(I) = E'(I)$$



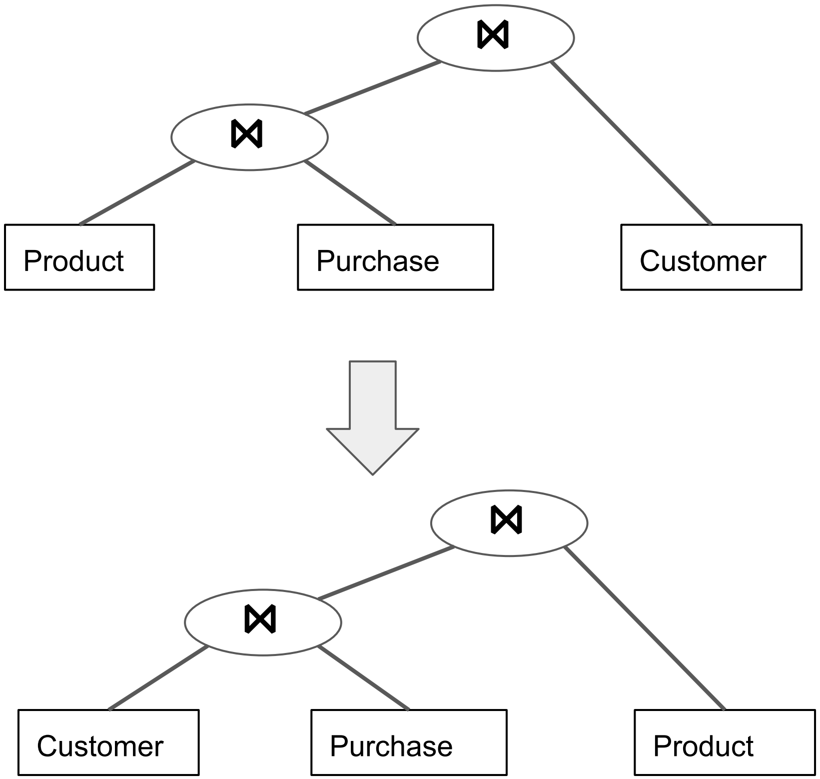
Equivalence Rules

Select and Join operators commute with each other



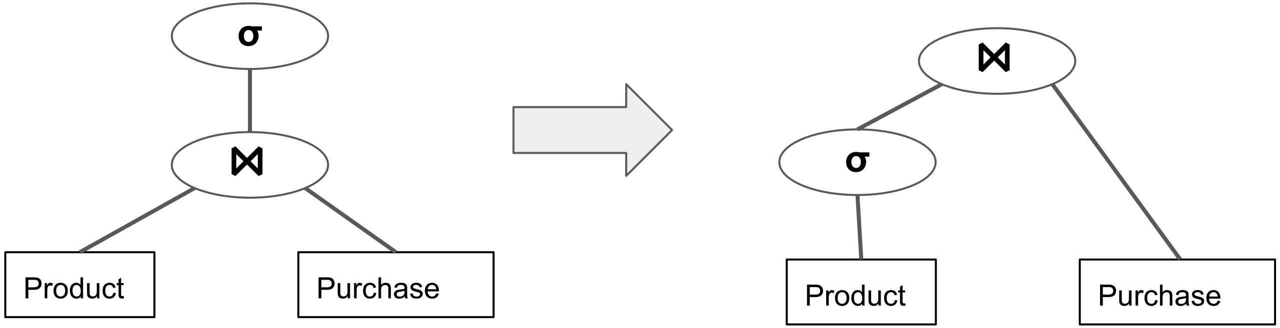
Equivalence Rules

Join operators are associative

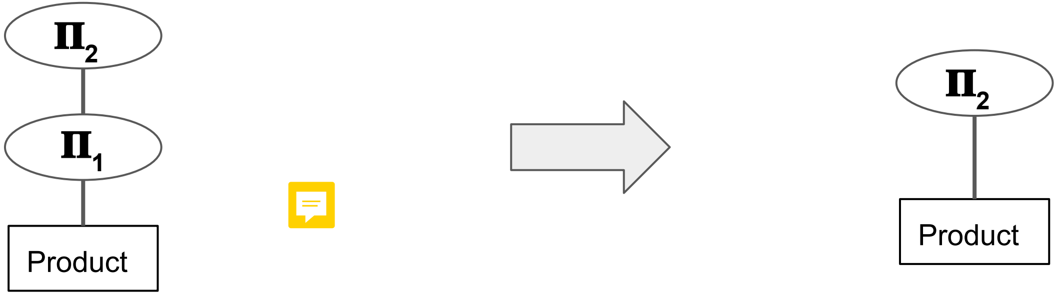


Equivalence Rules

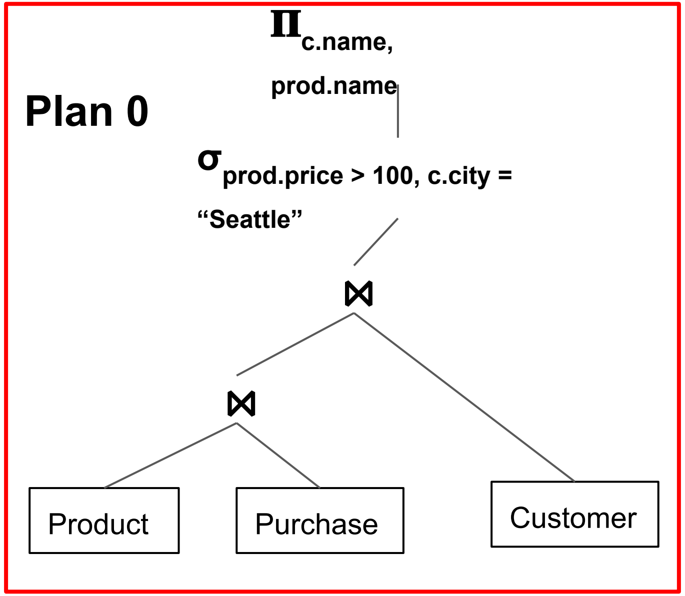
Select operator distributes over Join

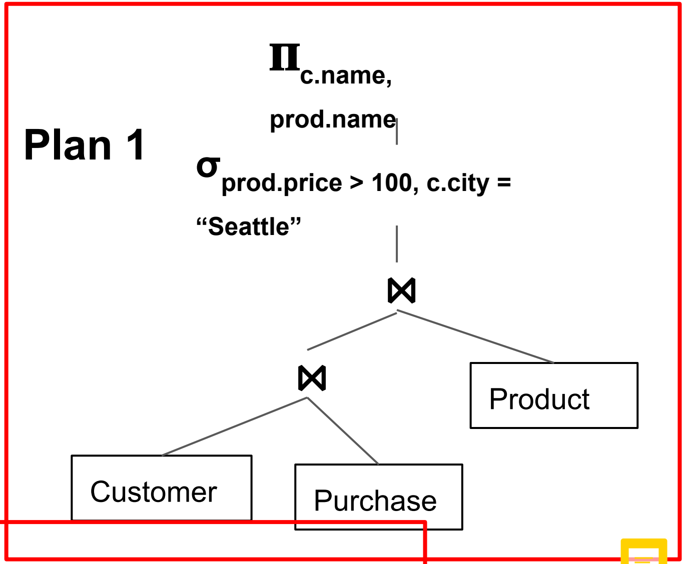


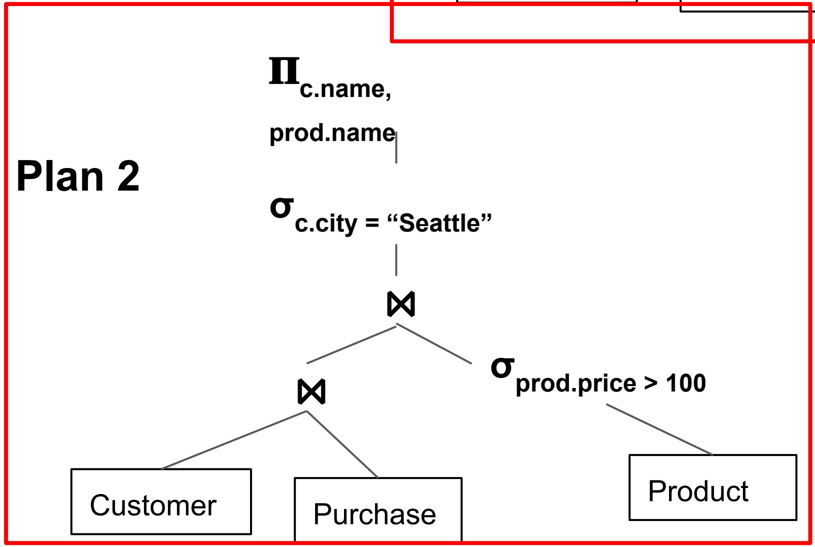
Project operator cascades



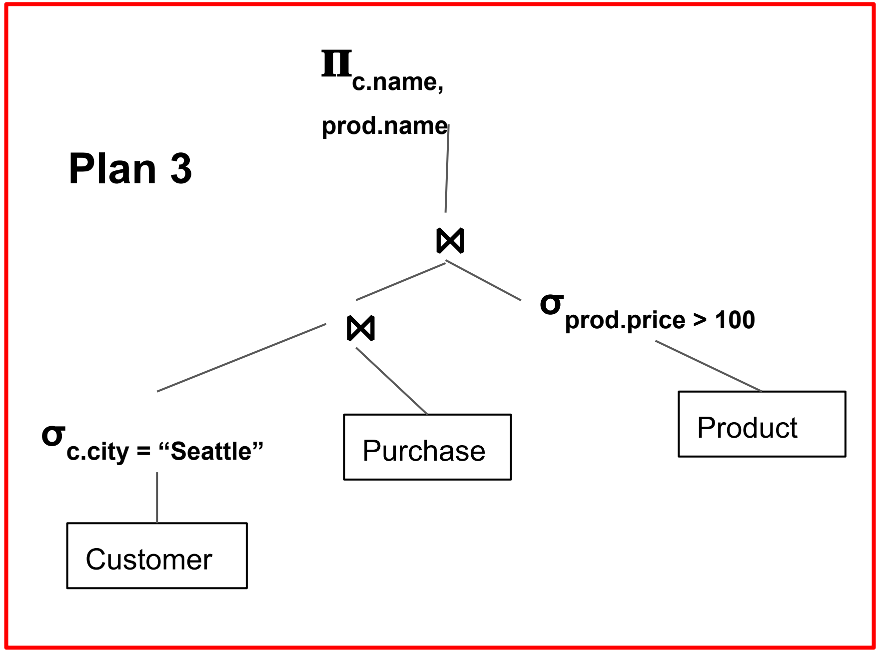
Example

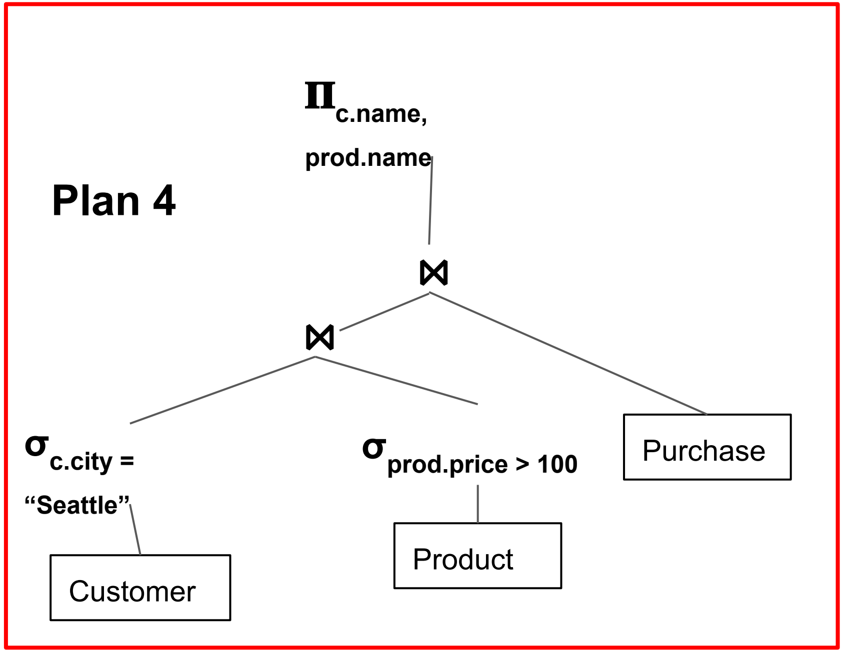

Plan 0
Π c.name, prod.name
σ prod.price > 100, c.city = “Seattle”
⋈
⋈
Product
Purchase
Customer


Plan 1
Π c.name, prod.name
σ prod.price > 100, c.city = “Seattle”
⋈
⋈
Product
Customer
Purchase


Plan 2
Π c.name, prod.name
σ c.city = “Seattle”
⋈
⋈
σ prod.price > 100
Customer
Purchase
Product

Example





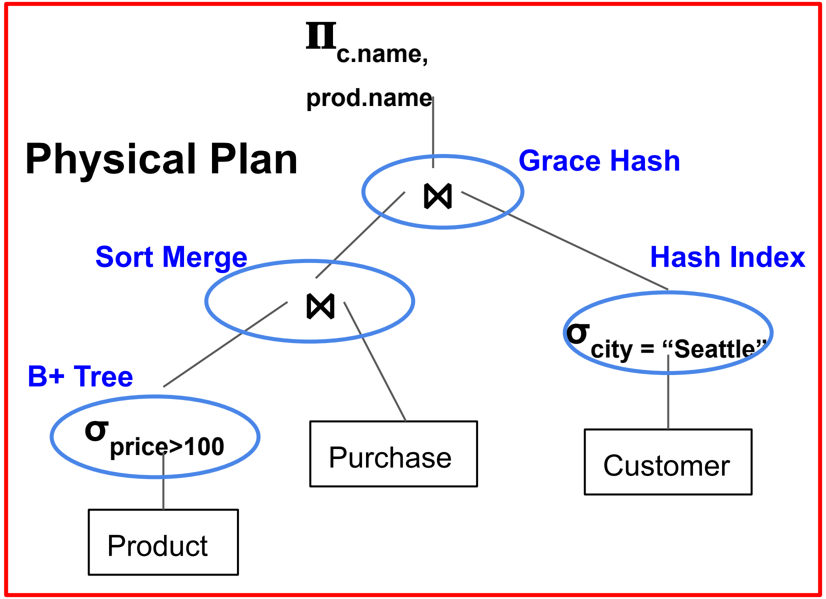
Any a few more logical plans

For each logical plan, there are many physical plans

Cost Estimation

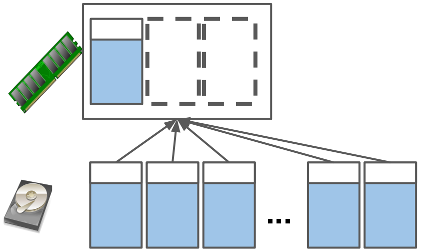
Cost Estimation

- Question: given a physical plan
  - What's the estimated cost of running it
  - Without running it

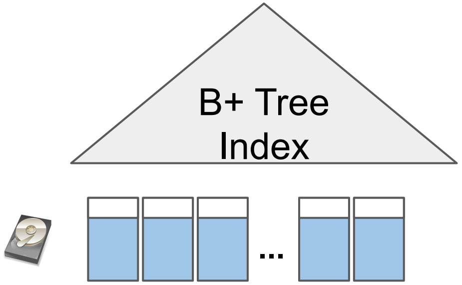


Cost Estimation

- Recall
  - Cost of select **depends on selectivity**

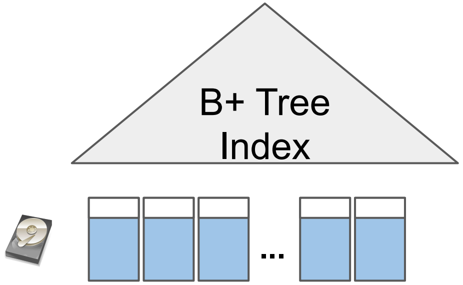


$B(R)$



$$\log(B(R)) + \alpha(C,R).|R|$$

(unclustered)



$$\log(B(R)) + \alpha(C,R).B(R)$$

(clustered)

Cost Estimation

- Cost of Join:
  - Depends on the size of the results

$(R \bowtie S) \bowtie T$

$$3.(B(R) + B(S)) + 3.(\boxed{B(R \bowtie S)} + B(T))$$

$R \bowtie (S \bowtie T)$

$$3.(B(S) + B(T)) + 3.(\boxed{B(S \bowtie T)} + B(R))$$

Cost Estimation

- Unsolved problems
- Still active area of research
- Two main approaches:
  - Build histogram over attributes
  - Sampling

Query Plan Search

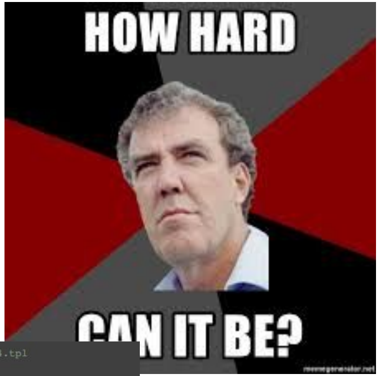
Query Plan Search

- So far, we have:
  - Generated lots of physical plans
  - Estimated their costs
- Now: how to pick the best one
  - With lowest cost

Query Search

- If your queries are like these:

```
select * from R,S
where R.id = S.id
```



- But in practice...

```
-- start query 14 in stream 0 using template query14.tpl
WITH cross_items
     AS (SELECT i_item_sk ss_item_sk
         FROM   item,
                (SELECT iss.i_brand_id    brand_id,
                        iss.i_class_id    class_id,
                        iss.i_category_id category_id
                 FROM   store_sales,
                        item iss,
                        date_dim d1
                 WHERE  ss_item_sk = iss.i_item_sk
                        AND ss_sold_date_sk = d1.d_date_sk
                        AND d1.d_year BETWEEN 1999 AND 1999 + 2
                 INTERSECT
                 SELECT ics.i_brand_id,
                        ics.i_class_id,
                        ics.i_category_id
                 FROM   catalog_sales,
                        item ics,
                        date_dim d2
                 WHERE  cs_item_sk = ics.i_item_sk
                        AND cs_sold_date_sk = d2.d_date_sk
                        AND d2.d_year BETWEEN 1999 AND 1999 + 2
                 INTERSECT
                 SELECT iws.i_brand_id,
                        iws.i_class_id,
                        iws.i_category_id
                 FROM   web_sales,
                        item iws,
                        date_dim d3
                 WHERE  ws_item_sk = iws.i_item_sk
                        AND ws_sold_date_sk = d3.d_date_sk
                        AND d3.d_year BETWEEN 1999 AND 1999 + 2)
         WHERE  i_brand_id = brand_id
                AND i_class_id = class_id
                AND i_category_id = category_id),
     avg_sales
     AS (SELECT Avg(quantity * list_price) average_sales
         FROM   (SELECT ss_quantity   quantity,
                        ss_list_price list_price
                 FROM   store_sales,
                        date_dim
```

Query Search

- But in practice…
- In fact:
  - Consider: $R_1 \bowtie R_2 \ldots \bowtie R_N$
  - # plans ~ Catalan number

The first Catalan numbers for n = 0, 1, 2, 3, ... are

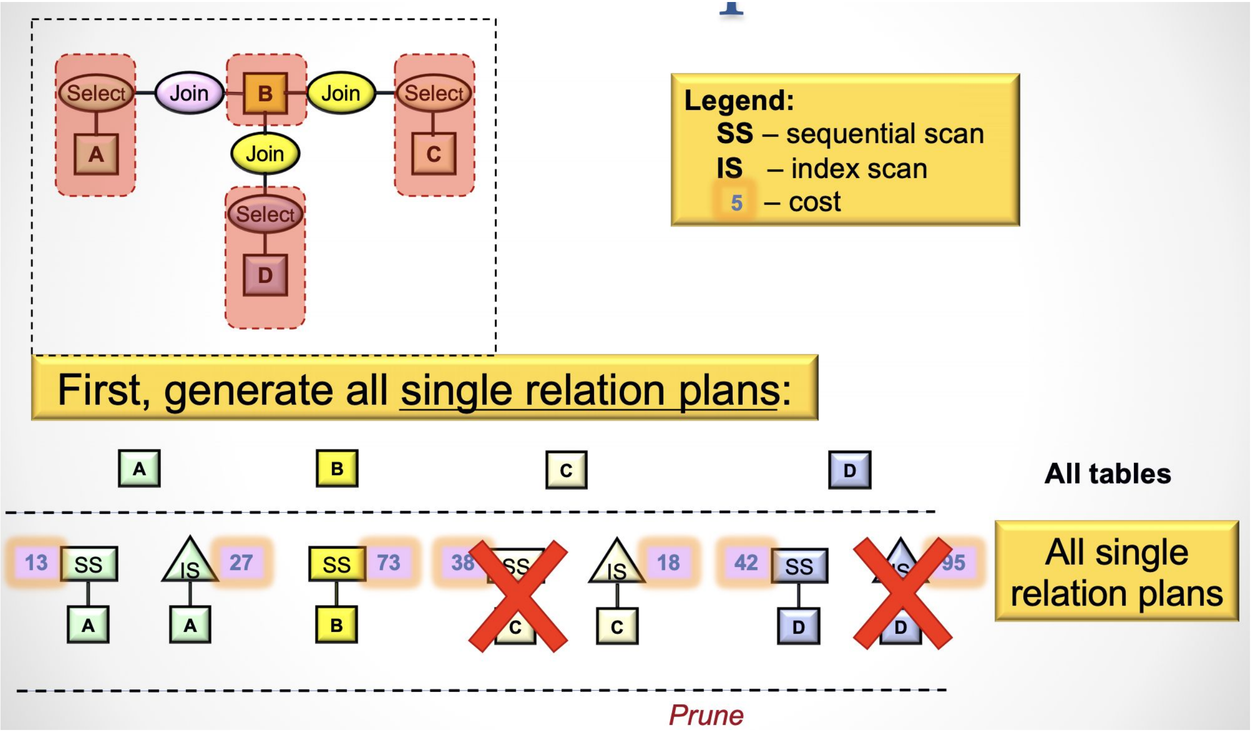
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, ...

```sql
-- start query 14 in stream 0 using template query14.tpl
WITH cross_items
     AS (SELECT i_item_sk ss_item_sk
         FROM   item,
                (SELECT iss.i_brand_id    brand_id,
                        iss.i_class_id    class_id,
                        iss.i_category_id category_id
                 FROM   store_sales,
                        item iss,
                        date_dim d1
                 WHERE  ss_item_sk = iss.i_item_sk
                        AND ss_sold_date_sk = d1.d_date_sk
                        AND d1.d_year BETWEEN 1999 AND 1999 + 2
                 INTERSECT
                 SELECT ics.i_brand_id,
                        ics.i_class_id,
                        ics.i_category_id
                 FROM   catalog_sales,
                        item ics,
                        date_dim d2
                 WHERE  cs_item_sk = ics.i_item_sk
                        AND cs_sold_date_sk = d2.d_date_sk
                        AND d2.d_year BETWEEN 1999 AND 1999 + 2
                 INTERSECT
                 SELECT iws.i_brand_id,
                        iws.i_class_id,
                        iws.i_category_id
                 FROM   web_sales,
                        item iws,
                        date_dim d3
                 WHERE  ws_item_sk = iws.i_item_sk
                        AND ws_sold_date_sk = d3.d_date_sk
                        AND d3.d_year BETWEEN 1999 AND 1999 + 2)
         WHERE  i_brand_id = brand_id
                AND i_class_id = class_id
                AND i_category_id = category_id),
     avg_sales
     AS (SELECT Avg(quantity * list_price) average_sales
         FROM   (SELECT ss_quantity   quantity,
                        ss_list_price list_price
                 FROM   store_sales,
                        date_dim
                 WHERE  ss_sold_date_sk = d_date_sk
                        AND d_year BETWEEN 1999 AND 1999 + 2
                 UNION ALL
                 SELECT cs_quantity   quantity,
                        cs_list_price list_price
                 FROM   catalog_sales,
                        date_dim
                 WHERE  cs_sold_date_sk = d_date_sk
                        AND d_year BETWEEN 1999 AND 1999 + 2
                 UNION ALL
                 SELECT ws_quantity   quantity,
                        ws_list_price list_price
                 FROM   web_sales,
                        date_dim
                 WHERE  ws_sold_date_sk = d_date_sk
```

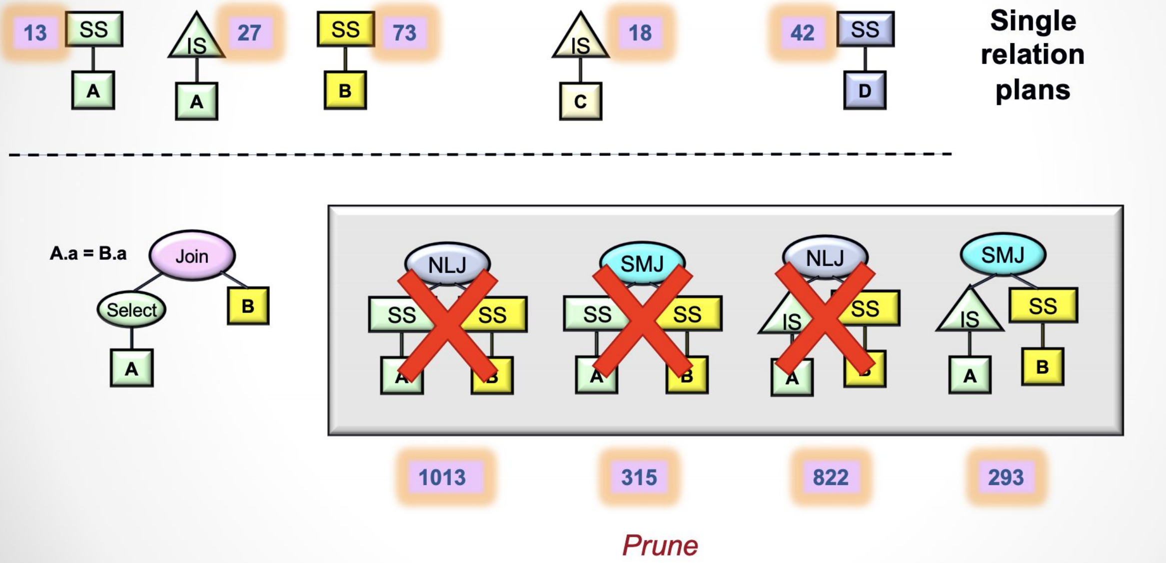
Query Search

- Another unsolved problem
- Most common approach: Dynamic Programming
  - Pass 1: find best 1-relation plans
  - Pass 2: find best 2-relation plans by joining results from Pass 1
  - …
  - Pass N: find best N-relation plans by joining results from Pass N-1
- Still exponential in search space!

Query Search



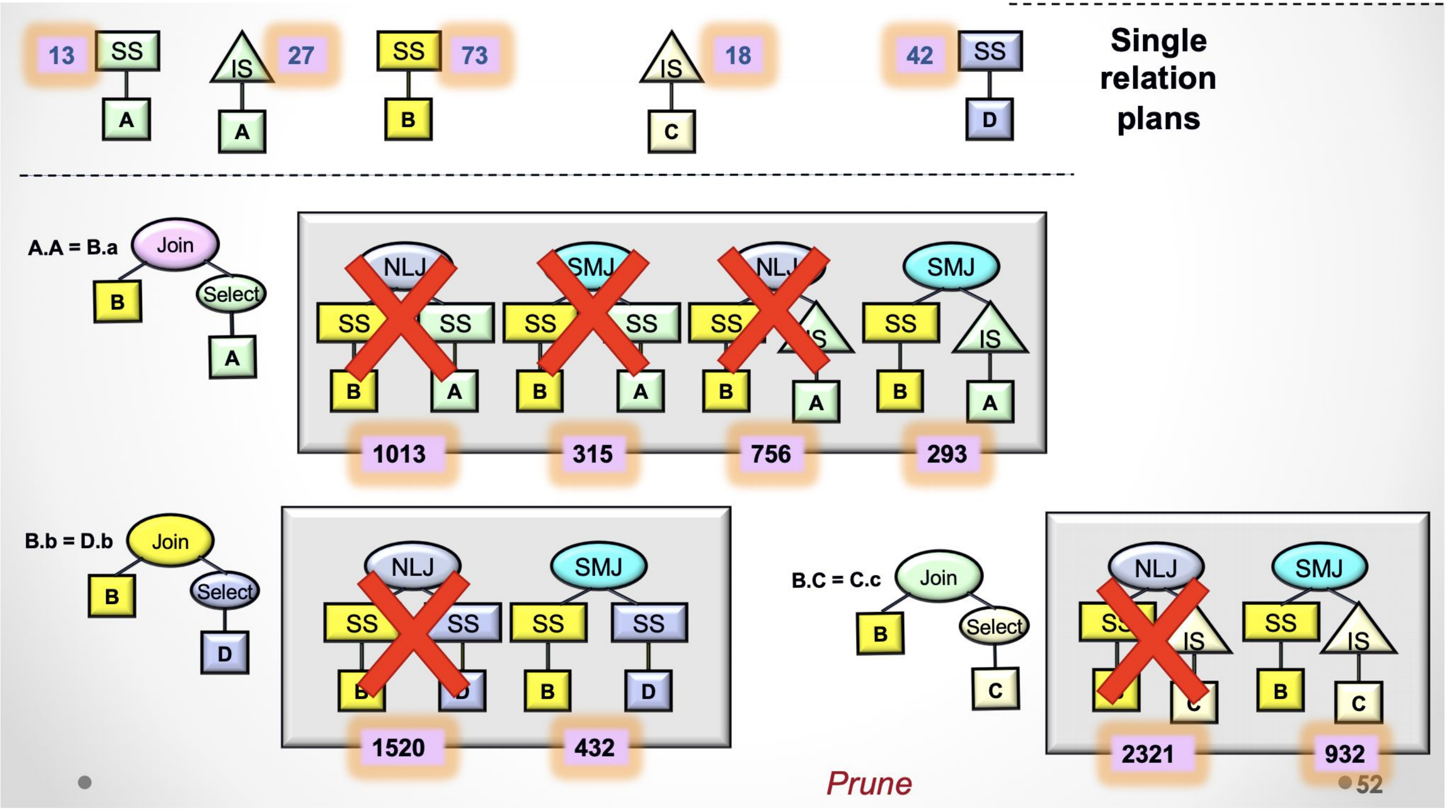
Query Search



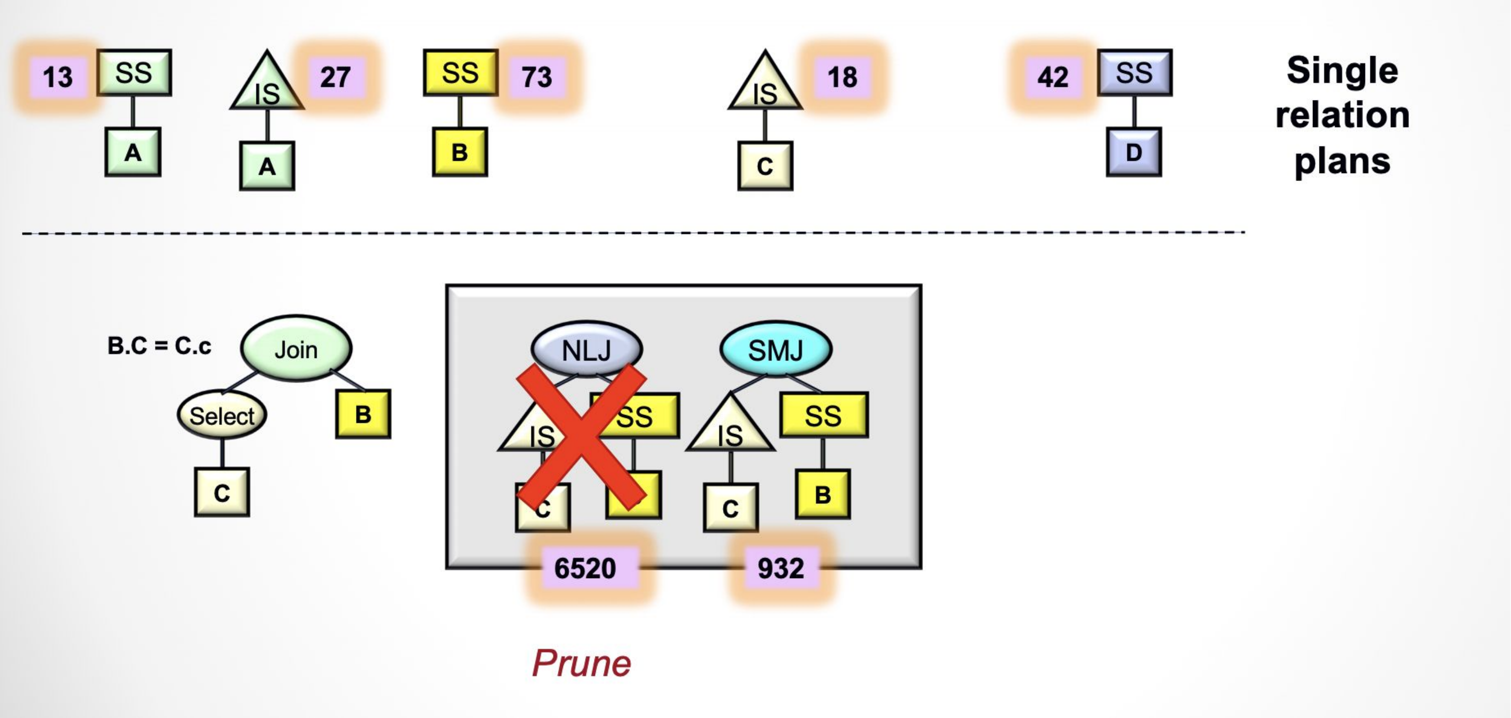
Let's assume there are 2 alternative join methods for the QO to select from:

1. **NLJ** = Nested Loops Join
2. **SMJ** = Sort Merge Join

Query Search



Query Search



Query Search

- DBMS often picks bad plans
  - Search is heuristic, space is too big
  - Too sensitive to errors in cost estimation
  - And cost estimation is very hard!
  - Cost models go out of date quickly
    - Hardware, software upgrade

Summary

- Query Optimization far from solved problem
    - ML for help!

Learning to Optimize Join Queries With Deep Reinforcement Learning

Sanjay Krishnan[1,2], Zongheng Yang[1], Ken Goldberg[1], Joseph M. Hellerstein[1], Ion Stoica[1]
[1]RISELab, UC Berkeley [2]Computer Science, University of Chicago
skr@cs.uchicago.edu {zongheng, goldberg, hellerstein, istoica}@berkeley.edu

DB] 10 Jan 2019

ABSTRACT

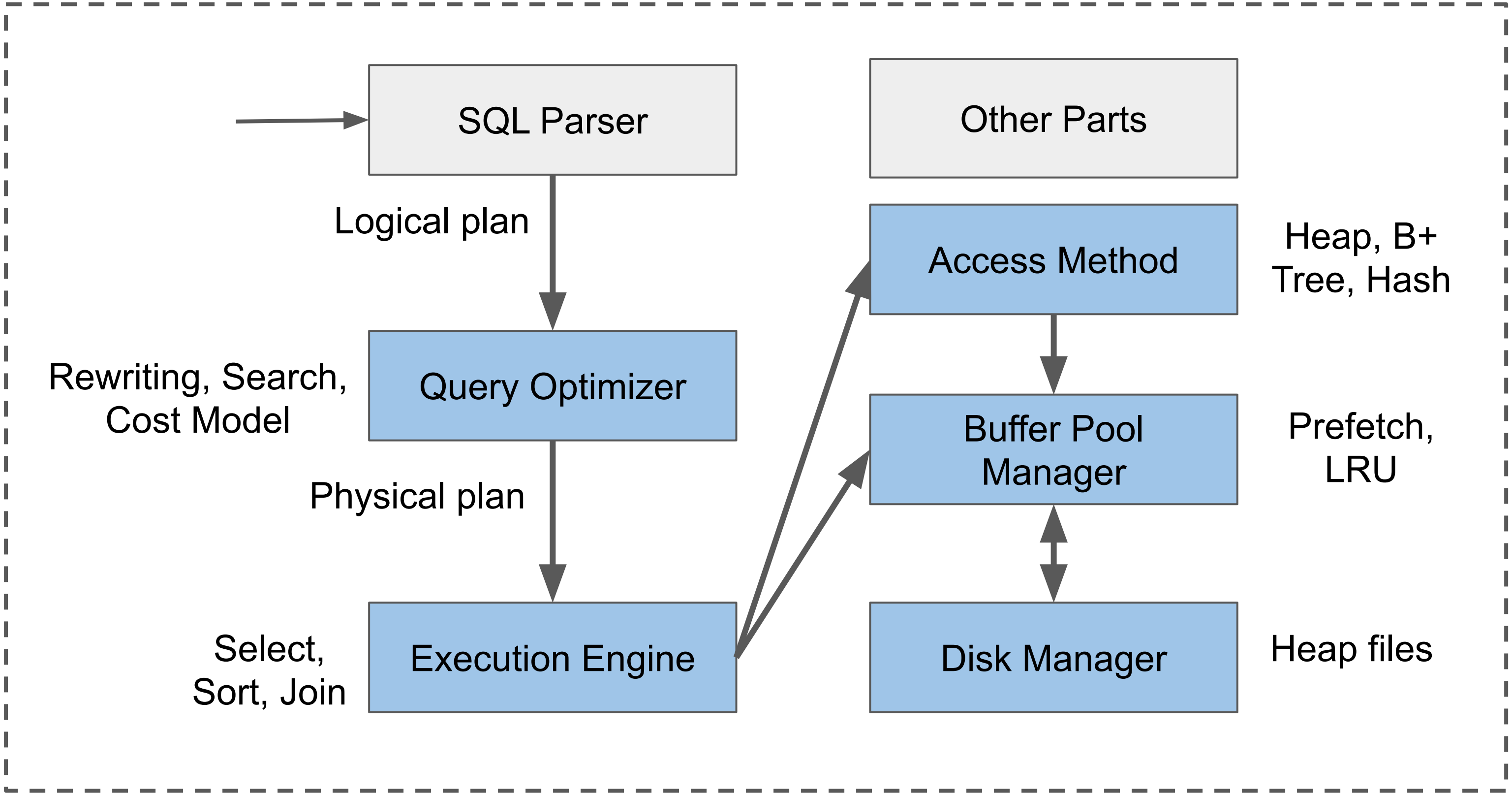
Exhaustive enumeration of all possible join orders is often avoided, and most optimizers leverage heuristics to prune the search space. The design and implementation of heuristics are well-understood when the cost model is roughly linear, and we find that these heuristics can be significantly suboptimal when there are non-linearities in cost. Ideally, instead of a fixed heuristic, we would want a strategy to guide the search space in a more data-driven way—tailoring the search to a specific dataset and query workload. Recognizing the link between classical Dynamic Programming enumeration methods and recent results in Reinforcement Learning (RL), we propose a new method for learning optimized join search strategies. We present our RL-based DQ optimizer, which

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Join optimization has been studied for more than four decades [44] and continues to be an active area of research [33, 40, 49]. The problem's combinatorial complexity leads to the ubiquitous use of *heuristics*. For example, classical System R-style dynamic programs often restrict their search space to certain shapes (e.g., "left-deep" plans). Query optimizers sometimes apply further heuristics to large join queries using genetic [4] or randomized [40] algorithms. In edge cases, these heuristics can break down (by definition), which results in poor plans [29].

Summary


SQL Parser
Other Parts
Logical plan
Access Method
Heap, B+ Tree, Hash
Rewriting, Search, Cost Model
Query Optimizer
Buffer Pool Manager
Prefetch, LRU
Physical plan
Select, Sort, Join
Execution Engine
Disk Manager
Heap files