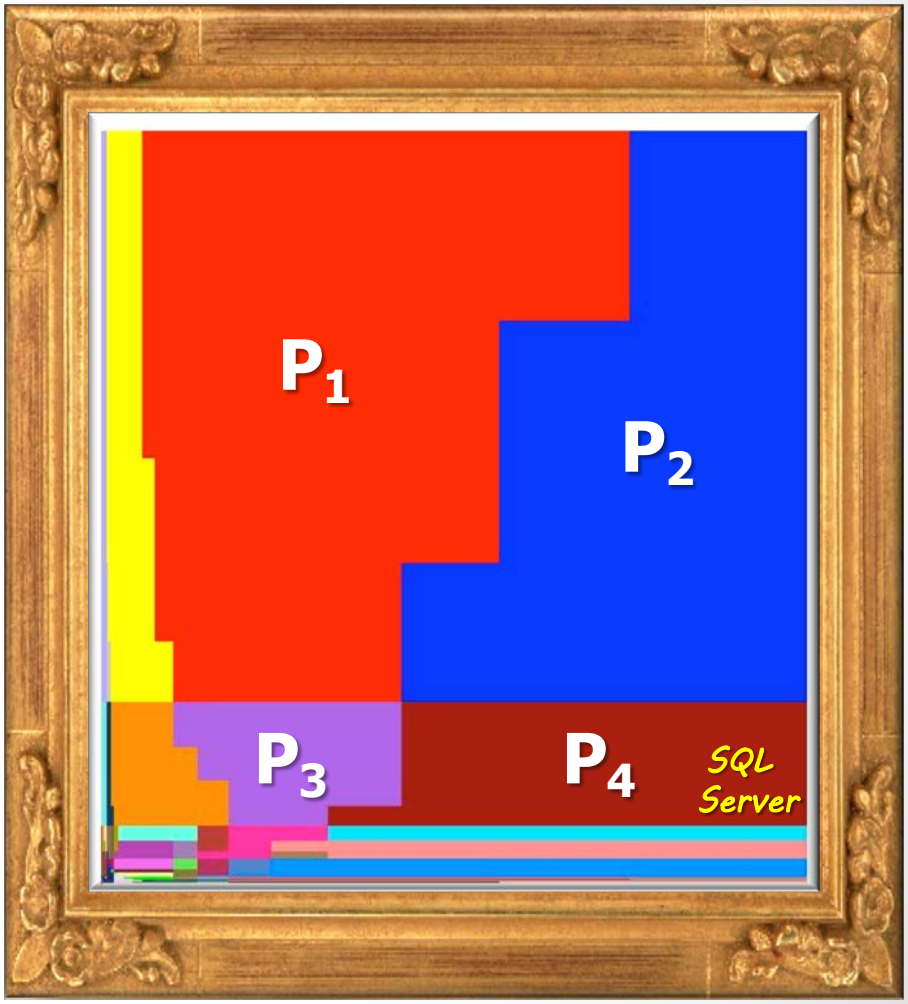
SQL Query Optimization: Why Is It So Hard To Get Right?

David J. DeWitt
MIT

Download slides and donate to a great cause:
BrentOzar.com/go/dewitt

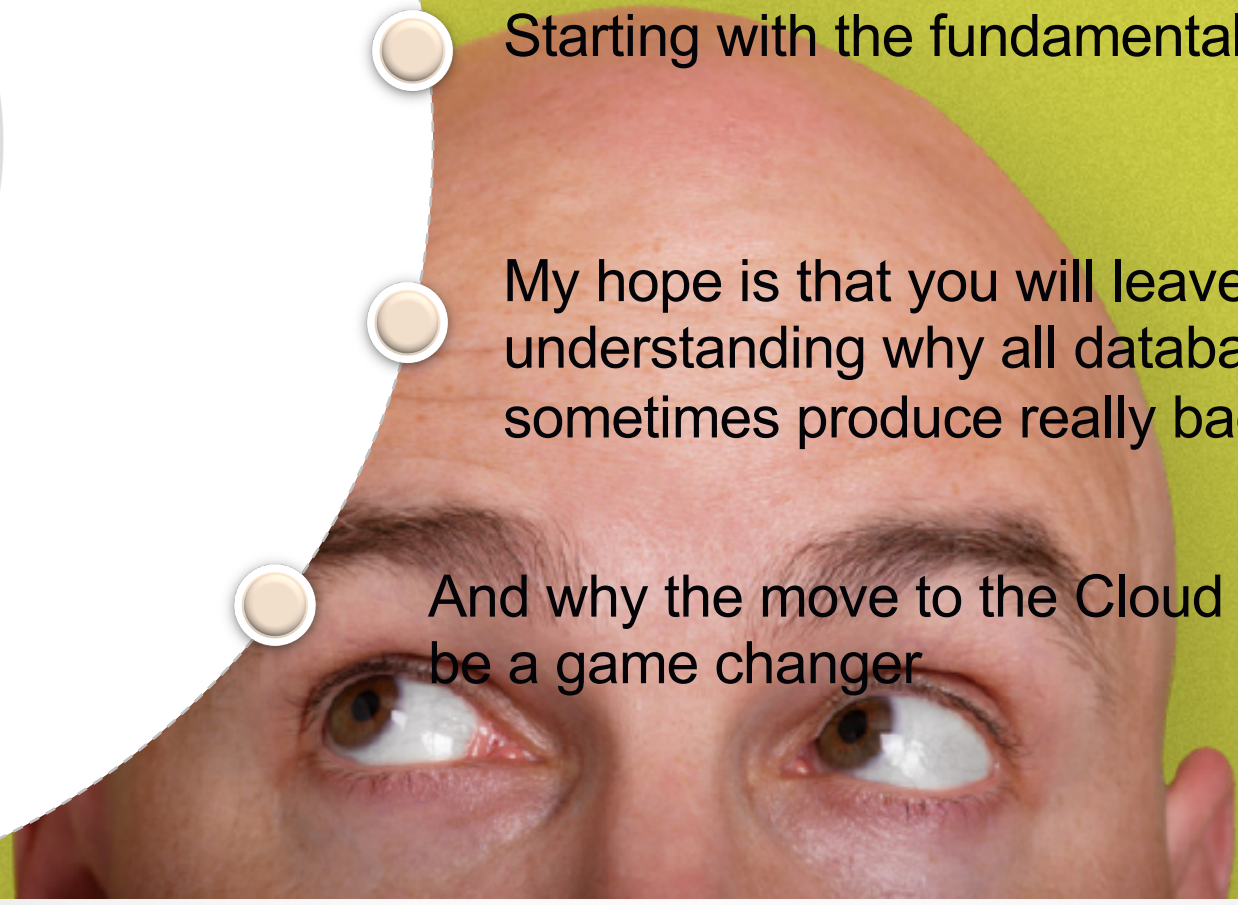
How About a Quiz to Start!

- Who painted this picture?
  - Mondrian?
  - Picasso?
  - Ingres?
- Actually it was the ***SQL Server query optimizer***!!
  - Plan space for **TPC-H query 8** as the parameter values for *Acct-Bal* and *ExtendedPrice* are varied
  - Each color represents a different query plan
  - Yikes!

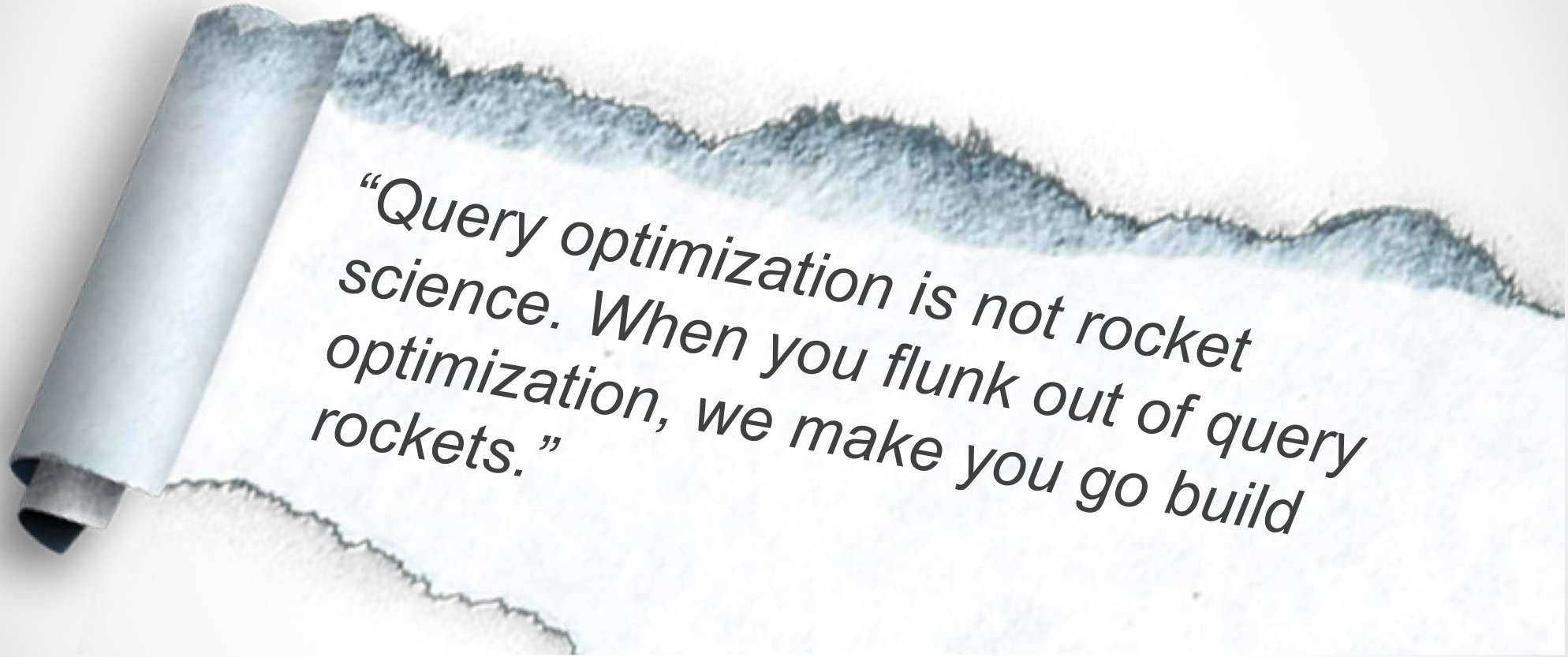


Today …

- I am going to talk about SQL query optimization
- Starting with the fundamental principals
- My hope is that you will leave understanding why all database systems sometimes produce really bad plans
- And why the move to the Cloud could be be a game changer



Anonymous Quote



The Role of the Query Optimizer

(100,000 ft view)

What's the Magic?

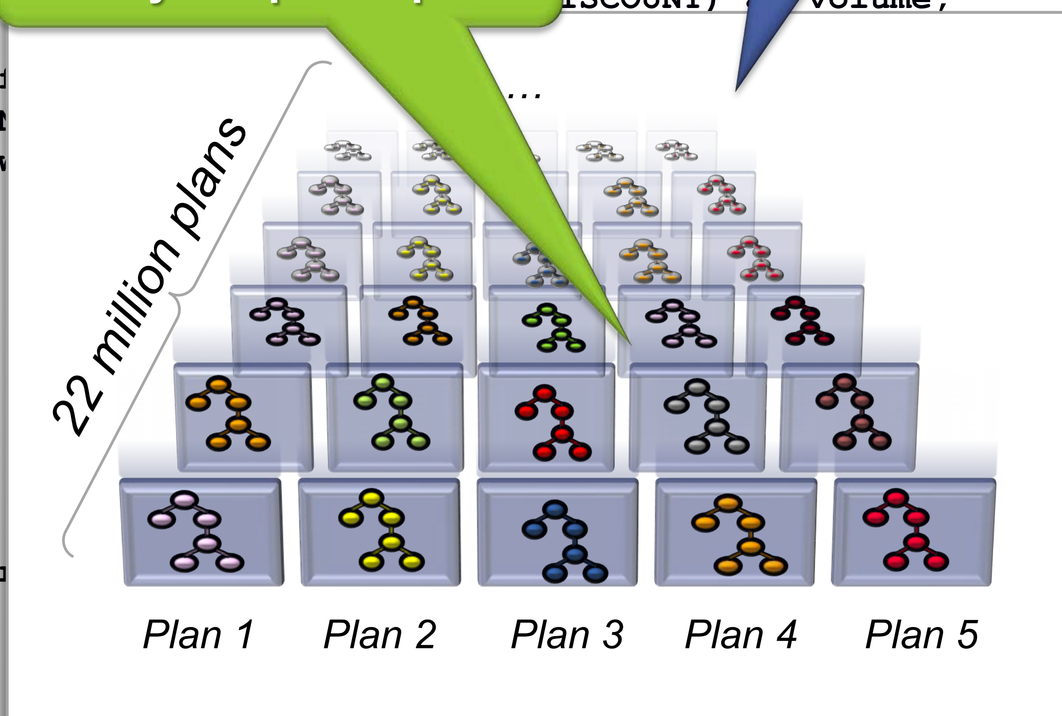
Consider Query 8 of the TPC-H benchmark:

```
Select o_year,
sum(case
 when nation = 'BRAZIL'
 else 0
end) / sum(volume)
```

```
 o_year,
ISCOUNT)    volume,
```

A very big haystack to be searching through

Should not take hours or days to pick a plan!




```
TION
STKE
KEY = R_REGIONKEY
TIONKEY
2-31'
```

There about **22 million** alternative ways of executing this query!

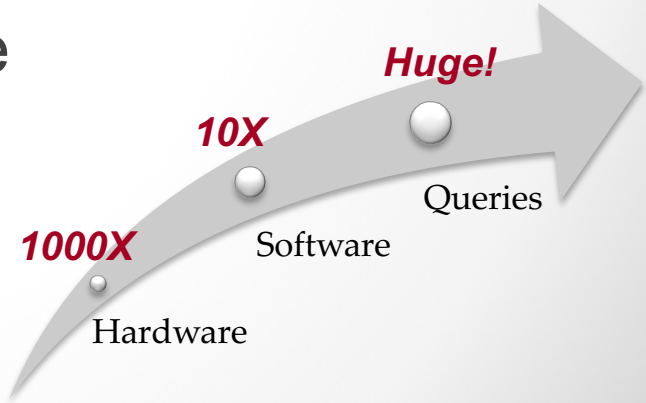
The QO must select a plan that runs in seconds or minutes, not days or weeks!

Some Historical Background

- Cost-based query optimization was invented by Pat Selinger as part of the IBM **System R** project in the late 1970s (System R became DB2)

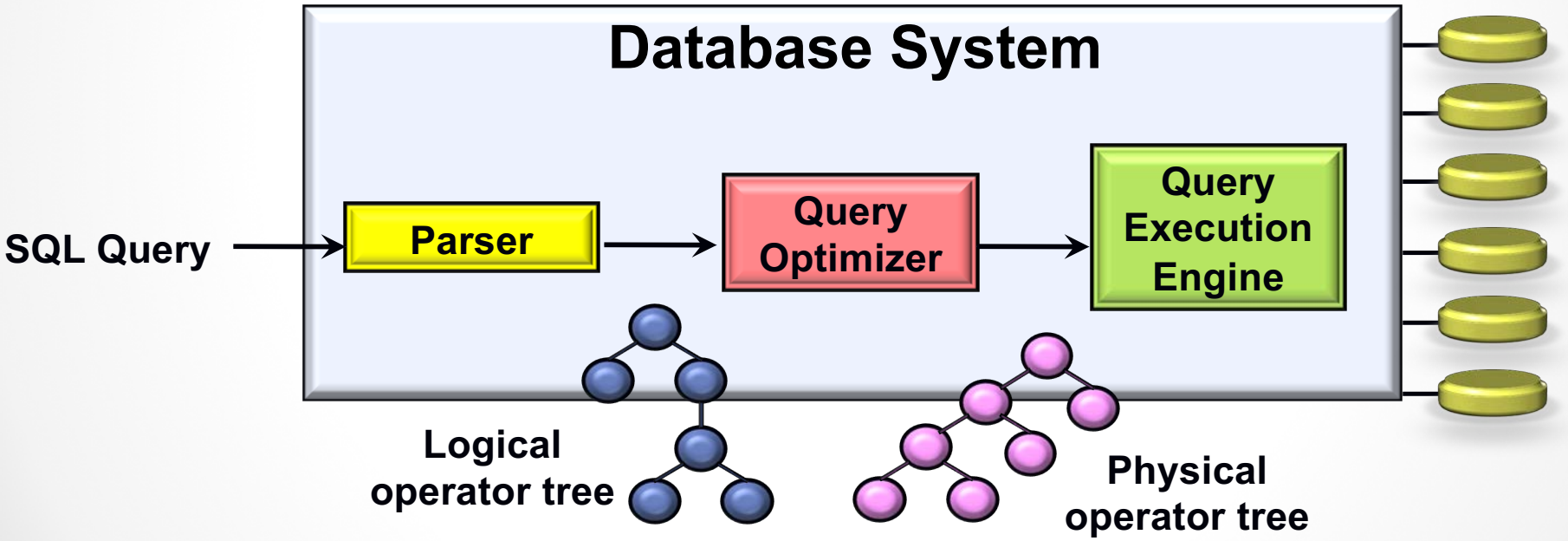
- Remains the hardest part of building a DBMS 30+ years later
  - Progress is hindered by fear of regressions
  - Far too frequently the QO picks an inefficient plan

- Situation further complicated by advances in hardware and the rest of the DBMS software
  - Hardware is *1000X* bigger and faster
  - DB software is *10X* faster
  - Queries over huge amounts of data are possible IF the QO picks the right plan



More Precisely:
The Role of the Query Optimizer

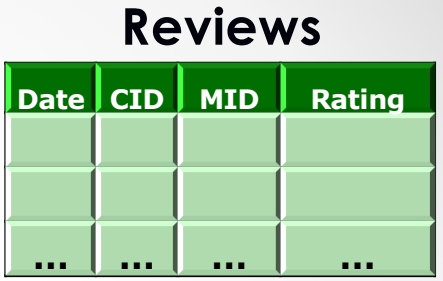
Transform SQL queries into an efficient execution plan



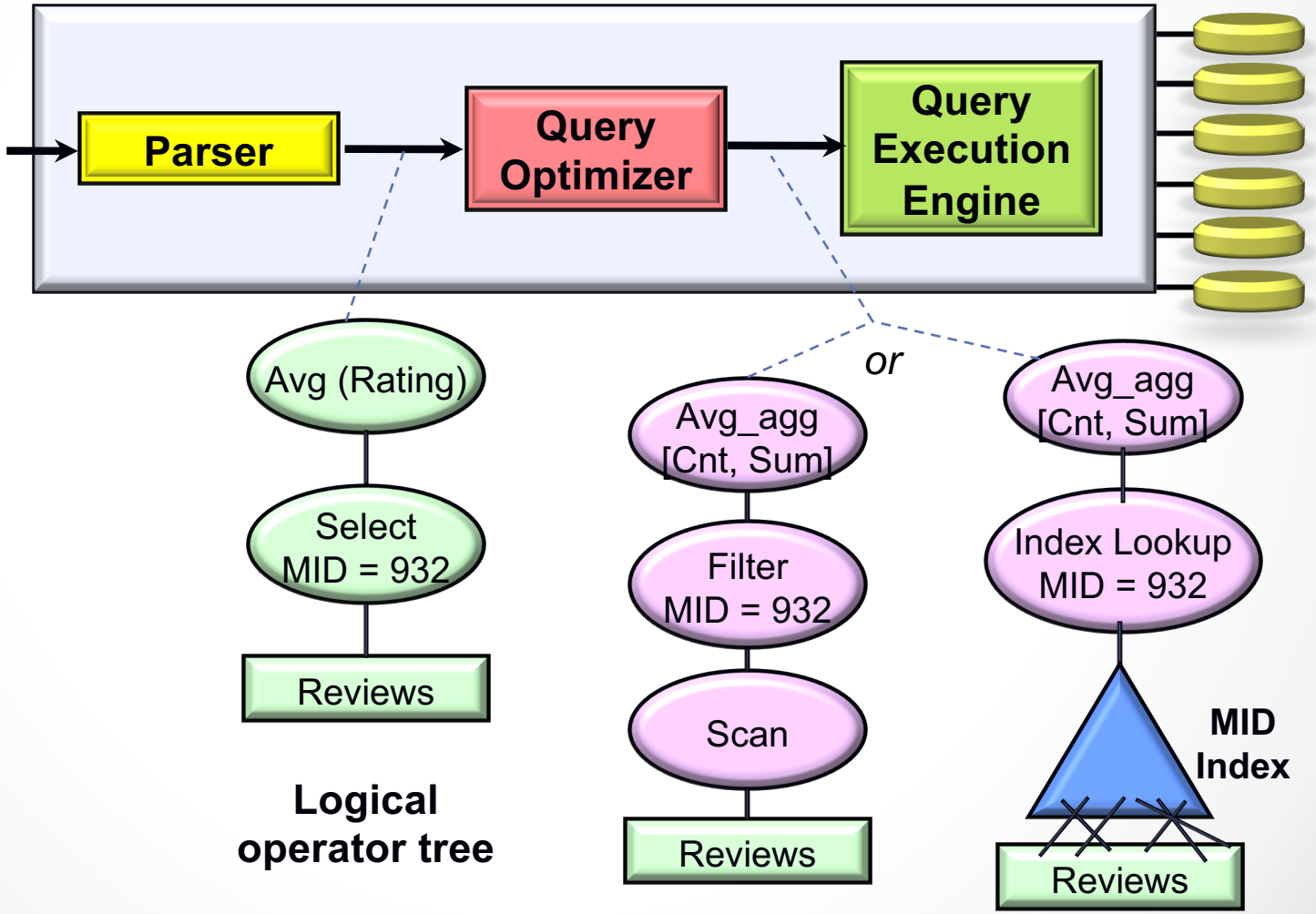
Logical operators: *what* they do
e.g., union, selection, project, join, grouping

Physical operators: *how* they do it
e.g., nested loop join, sort-merge join, hash join, index join

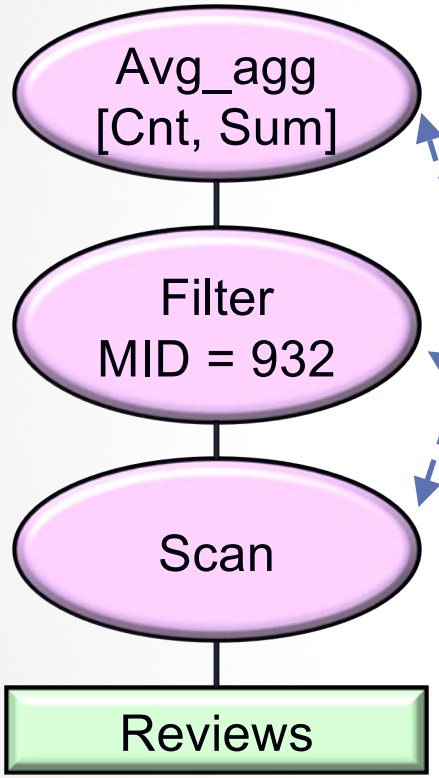
A First Example




```
SELECT
Average(Rating)
FROM Reviews
WHERE MID = 932
```

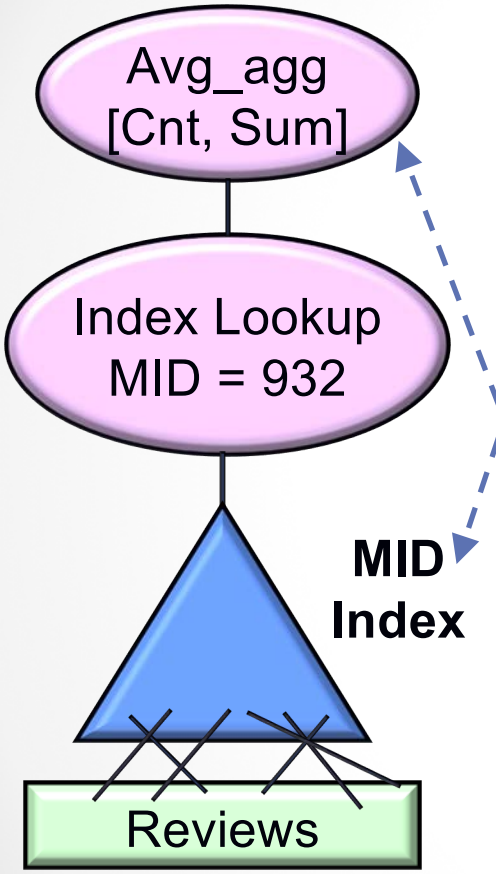


Query Plan #1



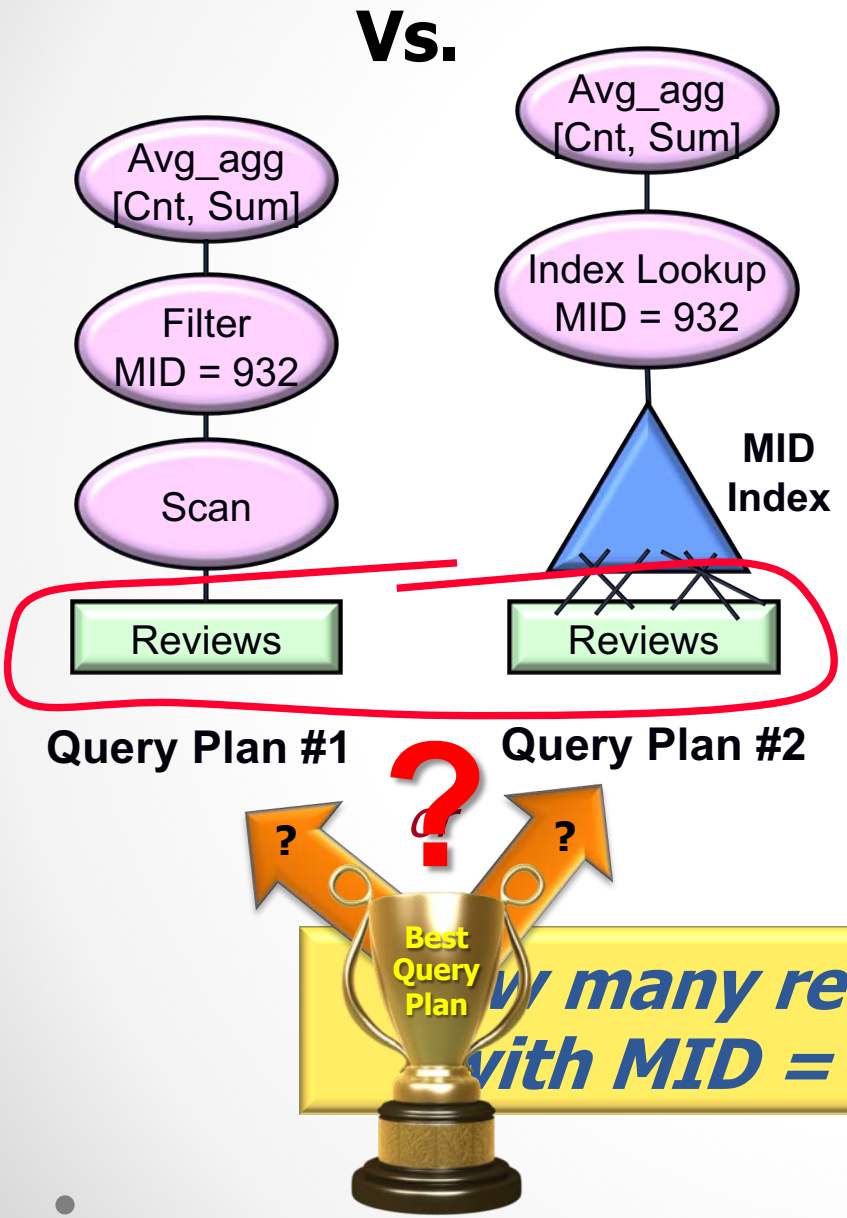
- Plan starts by scanning the entire ***Reviews*** table
  - # of disk I/Os will be equal to the # of pages in the ***Reviews*** table
  - I/Os will be sequential. Each I/O will require about 0.1 milliseconds (0.0001 seconds)
- Filter predicate "*MID = 932*" is applied to <u>all</u> rows
- Only rows that satisfy the predicate are passed on to the average computation

Query Plan #2



- MID index is used to retrieve only those rows whose *MID* field (attribute) is equal to *932*
  - Since index is not "clustered", about one disk I/O will be performed for each row
  - Each disk I/O will require a random seek and will take about 3 milliseconds (ms)
- Retrieved rows will be passed to the average computation

Which Plan Will be Faster?



- Query optimizer must pick between the two plans by estimating the cost of each
- To estimate the cost of a plan, the QO must:
  - Estimate the selectivity of the predicate *MID=932*
  - Calculate the cost of both plans in terms of CPU time and I/O time
- The QO uses statistics about each table to make these estimates
- The "best" plan depends on how many reviews there are for movie with *MID = 932*

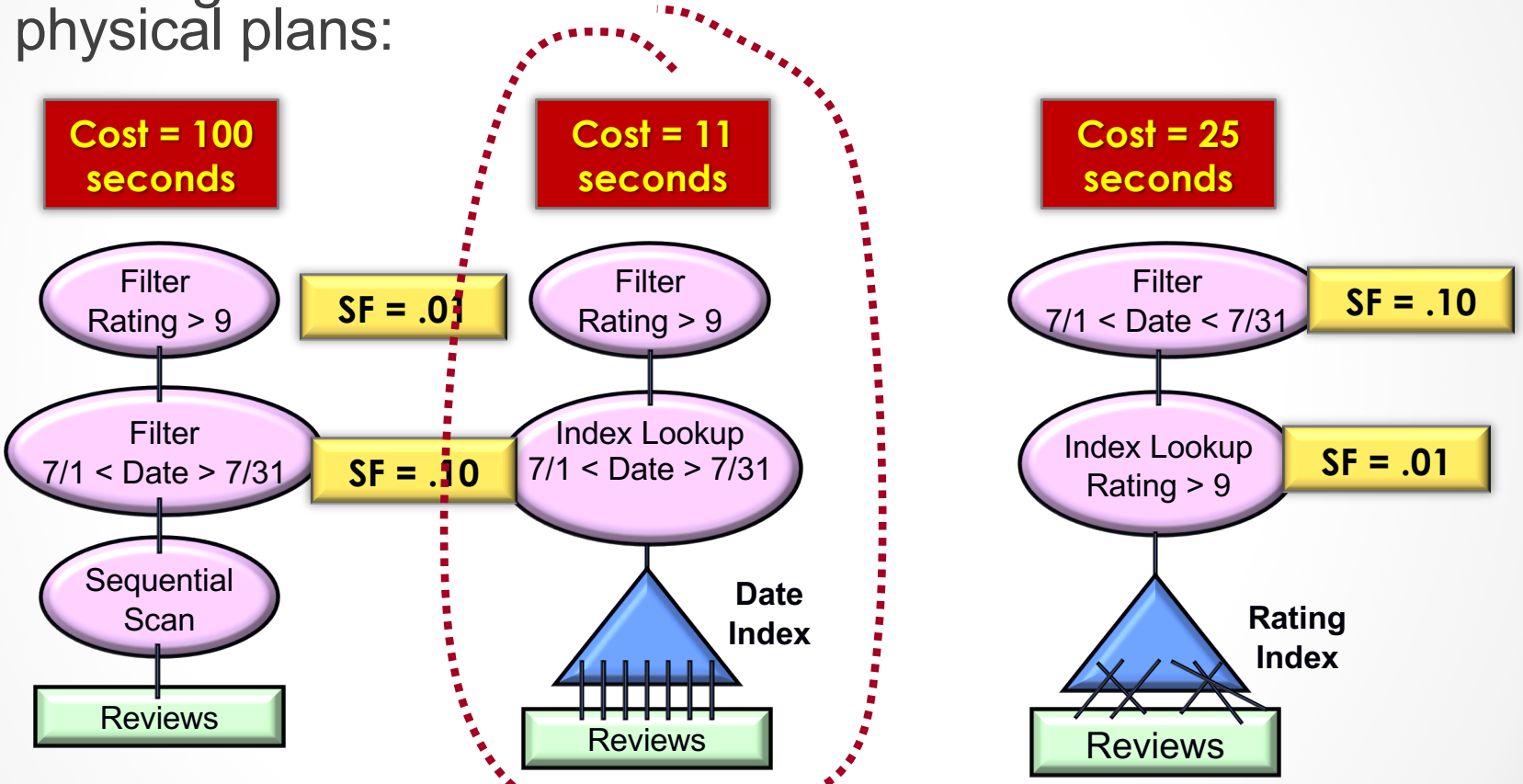
w many reviews for the movie with MID = 932 will there be?

A Slightly More Complex Query

- Consider the query:

```
SELECT *
FROM Reviews
WHERE 7/1< date < 7/31 AND
      rating > 9
```

- Optimizer might first enumerate three physical plans:



- Then, estimate selectivity factors
- Then, calculate total cost
- Finally, pick the **plan with the lowest cost**

Query Optimization: The Main Steps

✓ 1  Enumerate logically equivalent plans by applying equivalence rules

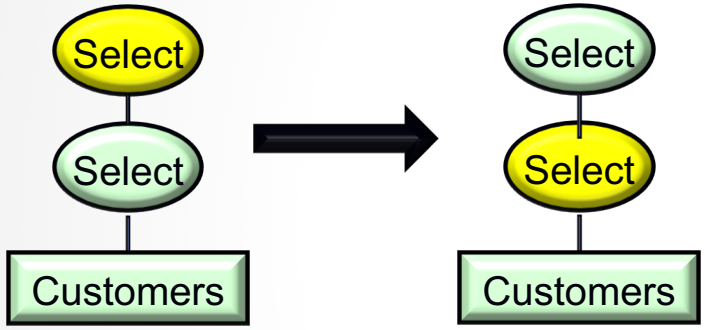
2  For each logically equivalent plan, enumerate all alternative physical query plans

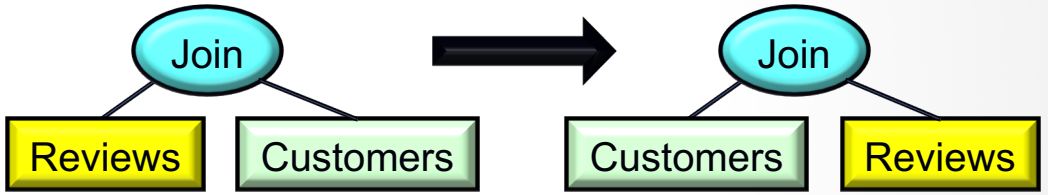
3  Estimate the cost of each of the alternative physical query plans

4  Run the plan with lowest estimated overall cost

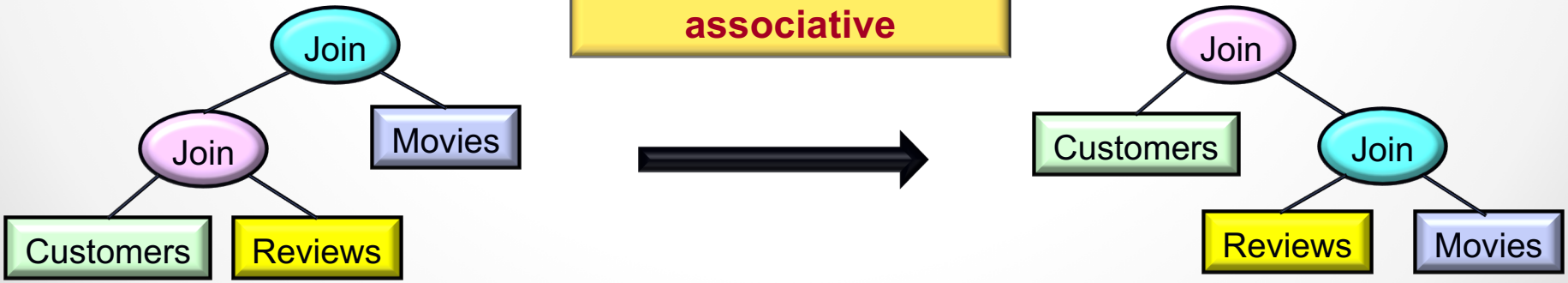
Equivalence Rules

Select and join operators commute with each other



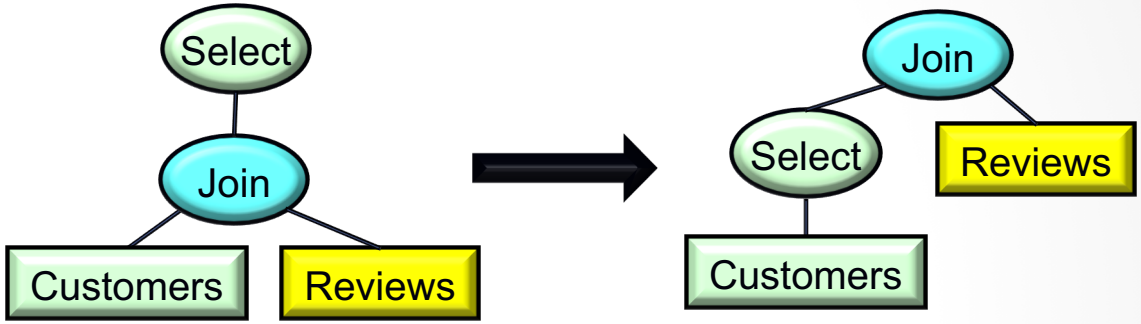


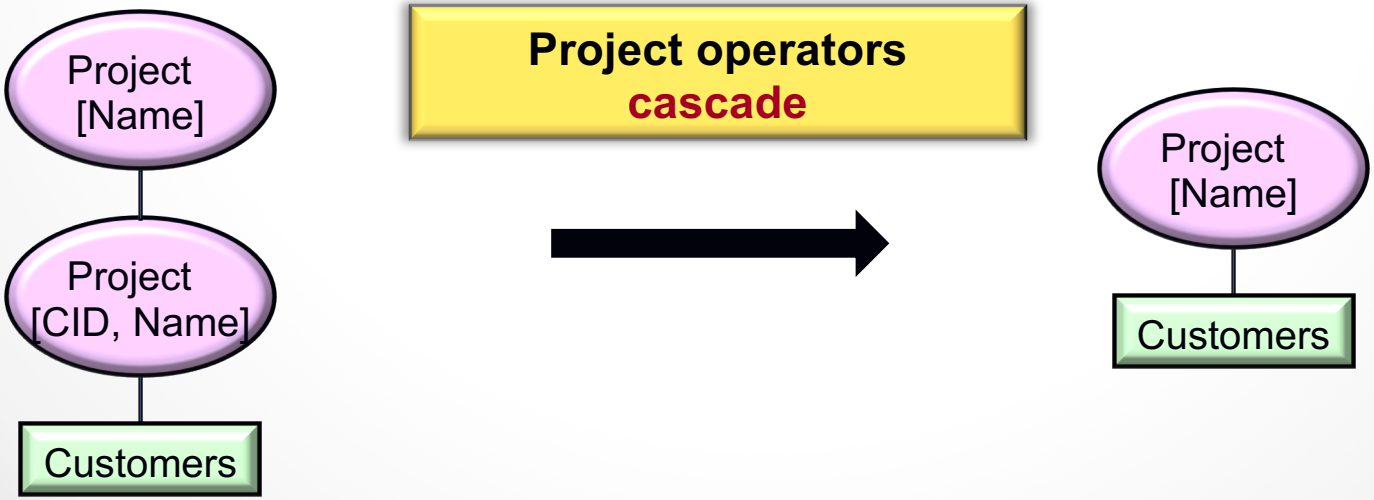
Join operators are associative



Equivalence Rules (cont.)

Select operator distributes over joins



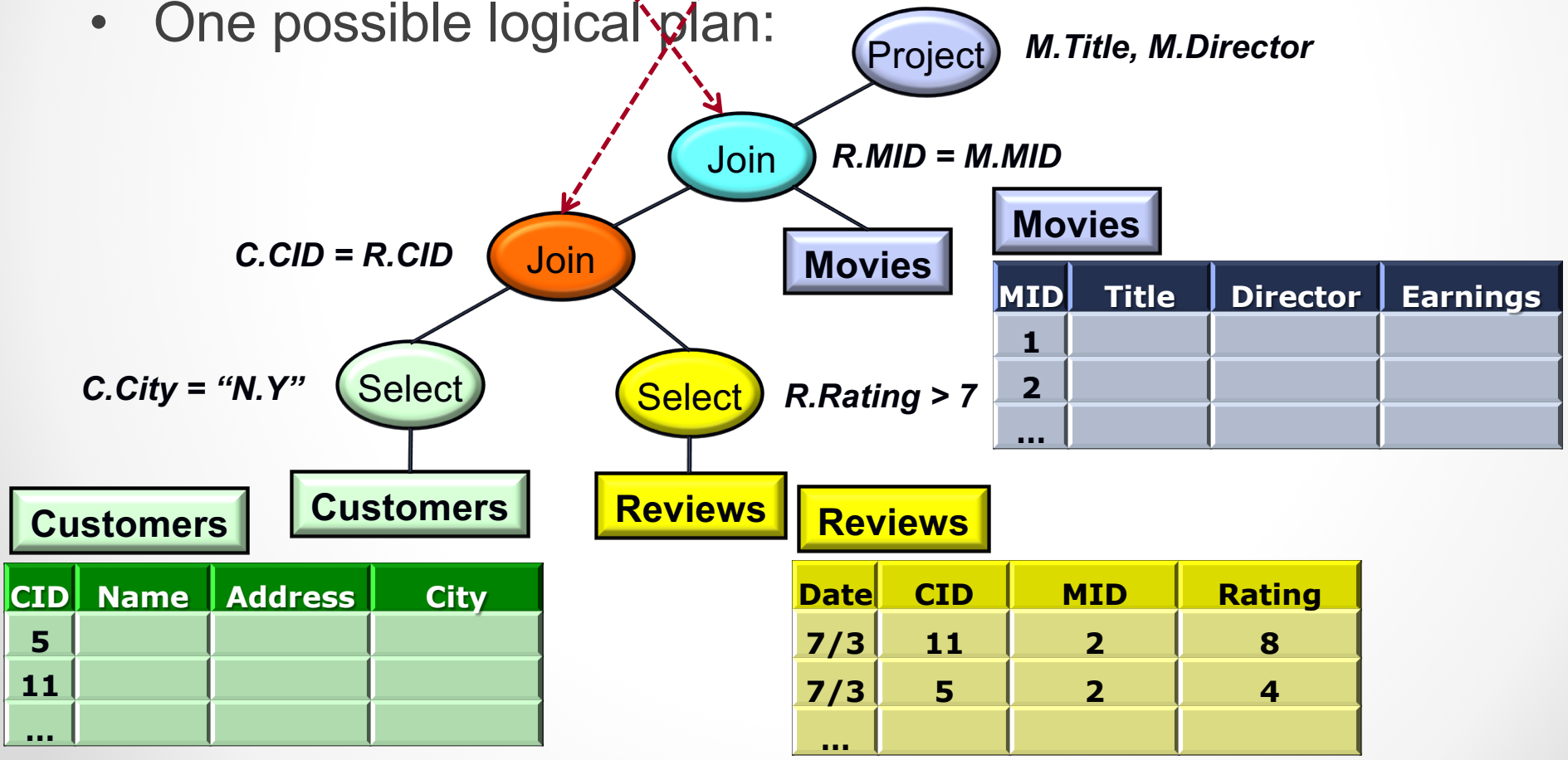


Example of Equivalent Logical Plans

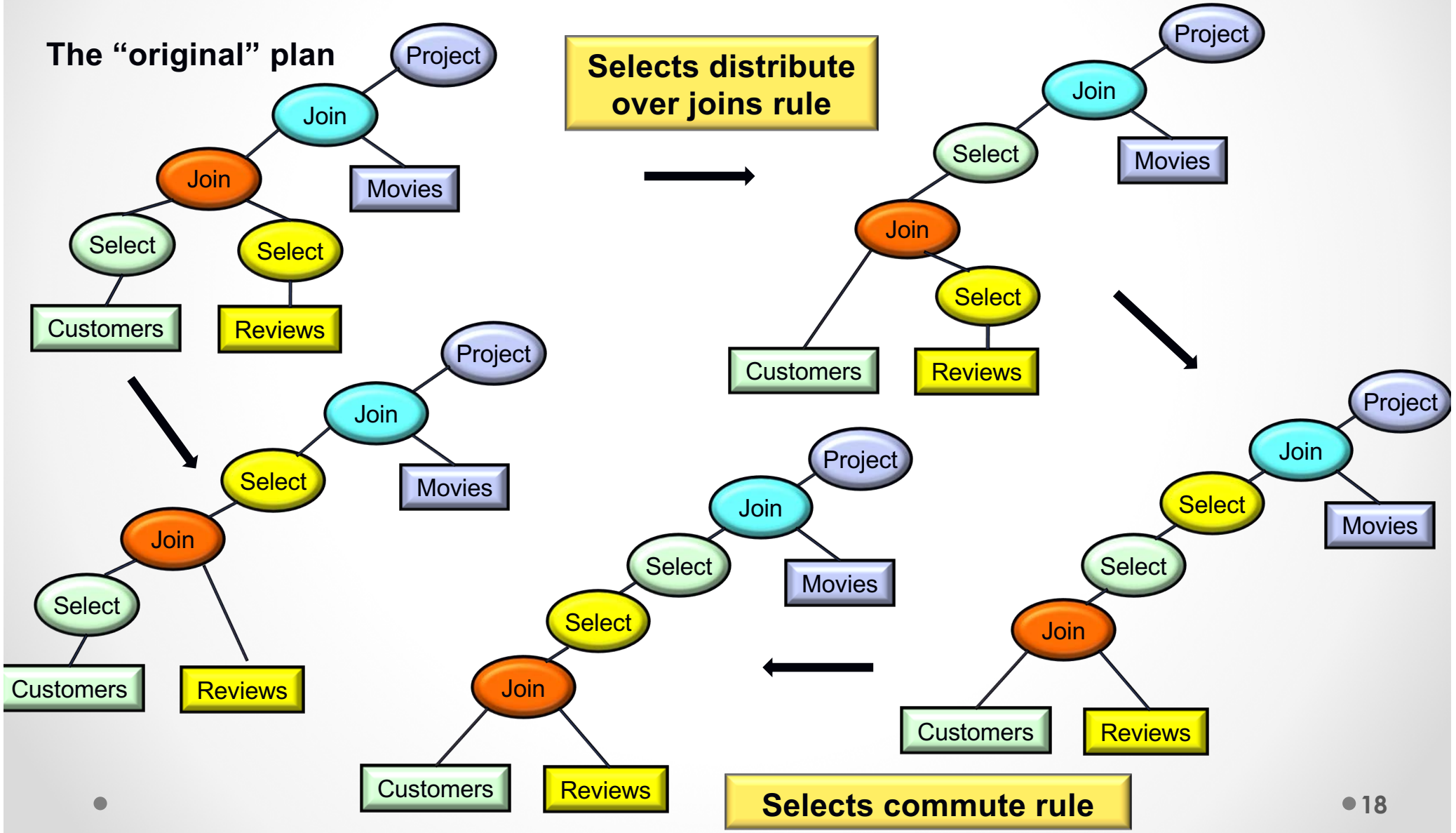
```
SELECT M.Title, M.Director
FROM Movies M, Reviews R, Customers C
WHERE C.City = "N.Y." AND R.Rating > 7
      AND M.MID = R.MID AND C.CID = R.CID
```

Find titles and director names of movies with a rating > 7 from customers residing in NYC

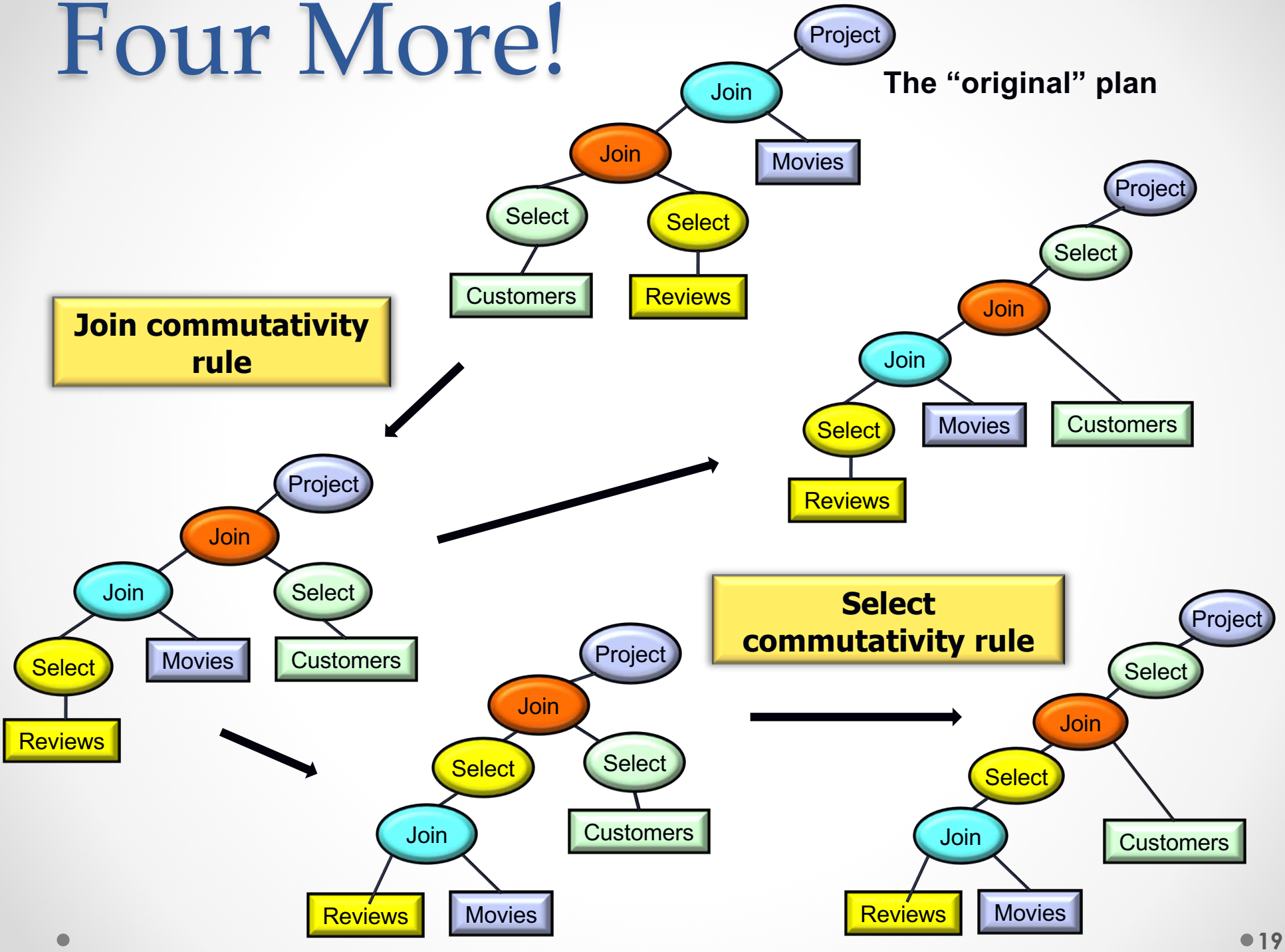
- One possible logical plan:



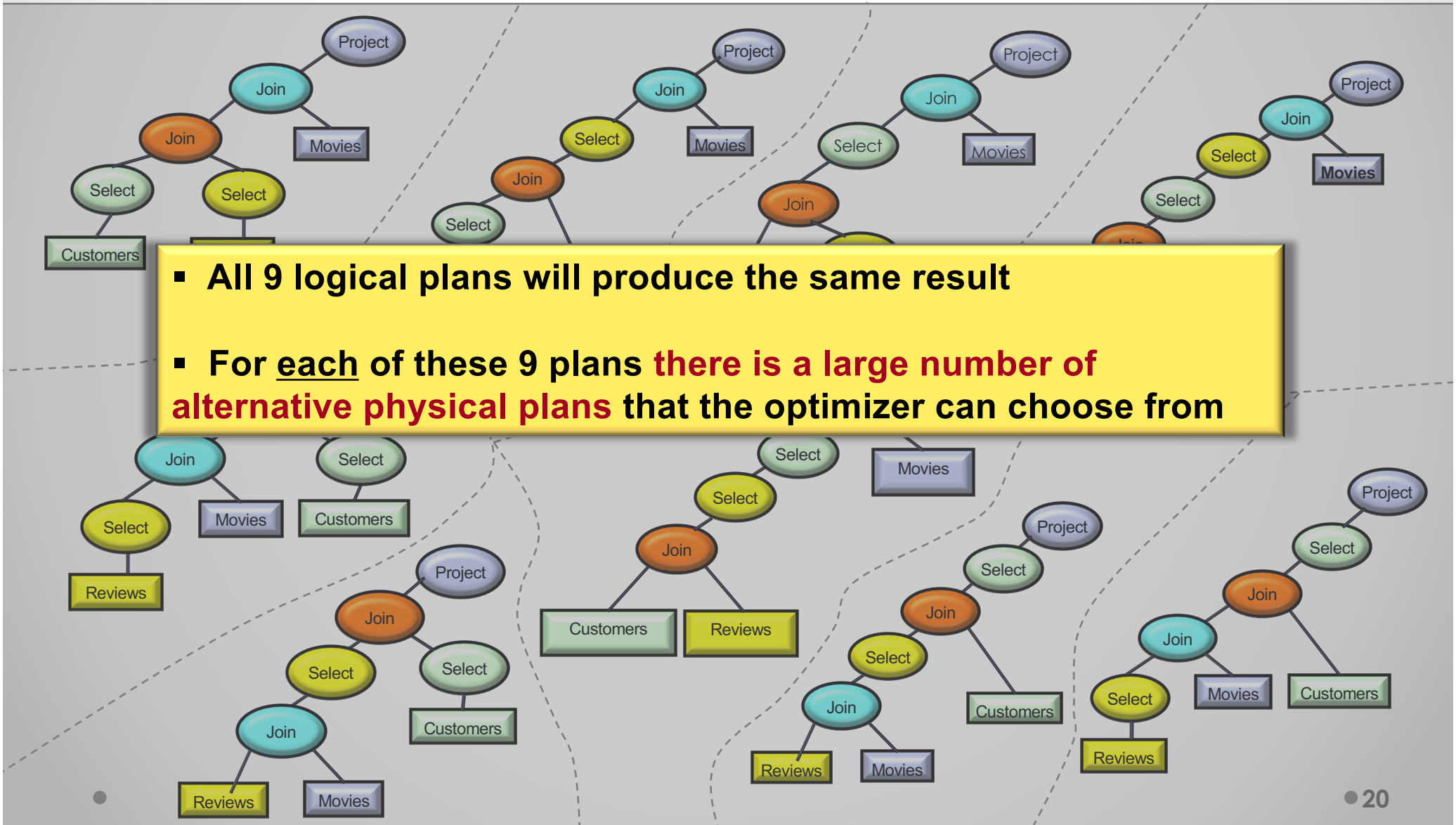
Five Logically "Equivalent" Plans



Four More!


Project
Join
Join
Movies
Select
Select
Customers
Reviews
The "original" plan
Join commutativity rule
Project
Join
Join
Select
Select
Movies
Customers
Reviews
Project
Select
Join
Join
Select
Movies
Customers
Reviews
Select commutativity rule
Project
Join
Select
Select
Join
Customers
Reviews
Movies
Project
Select
Join
Select
Join
Customers
Reviews
Movies

9 Logically Equivalent Plans, In Total



- **All 9 logical plans will produce the same result**
- **For <u>each</u> of these 9 plans there is a large number of alternative physical plans that the optimizer can choose from**

Query Optimization: The Main Steps

✓ 1  Enumerate logically equivalent plans by applying equivalence rules

✓ 2  For each logically equivalent plan, enumerate all alternative physical query plans

3  Estimate the cost of each of the alternative physical query plans

4  Run the plan with lowest estimated overall cost

Physical Plan Example

- Assume that the optimizer has:
  - Three join strategies that it can select from:
    - nested loops (NL), sort-merge join (SMJ), and hash join (HJ)
  - Two selection strategies:
    - sequential scan (SS) and index scan (IS)
- Consider JUST ONE of the 9 logical plans

- There are actually **36** possible physical alternatives for this single logical plan. (*I was too lazy to draw pictures of all 36*).
- With *9 equivalent logical plans*, there are **324** = **(9 * 36)** physical plans that the optimizer must enumerate and cost as part of the search for the best execution plan for the query

And this was a VERY simple query!

- Later we will look at how *dynamic programming* is used to explore the space of logical and physical plans w/o enumerating the entire plan space

Query Optimization: The Main Steps

✓ 1

Enumerate logically equivalent plans by applying equivalence rules

✓ 2

For each logically equivalent plan, enumerate all alternative physical query plans

✓ 3

Estimate the cost of each of the alternative physical query plans.

- Estimate the selectivity factor and output cardinality of each predicate
- Estimate the cost of each operator

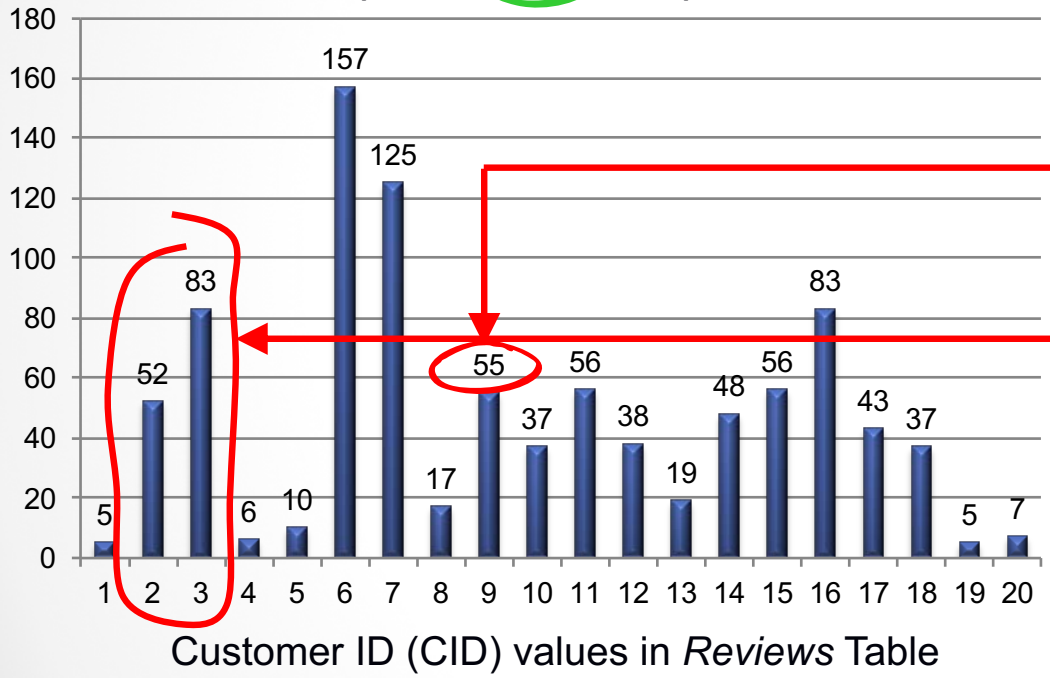
4

Run the plan with lowest estimated overall cost

Selectivity Estimation

- Task of estimating how many rows will satisfy a predicate such as ***Movies.MID***=932
- Plan quality is highly dependent on quality of the estimates that the query optimizer makes
- **Histograms** are the standard technique used to estimate selectivity factors for predicates on a single table
- Many different flavors:
  - Equi-Width
  - Equi-Height
  - Max-Diff
  - …

Histogram Motivation



Some examples:

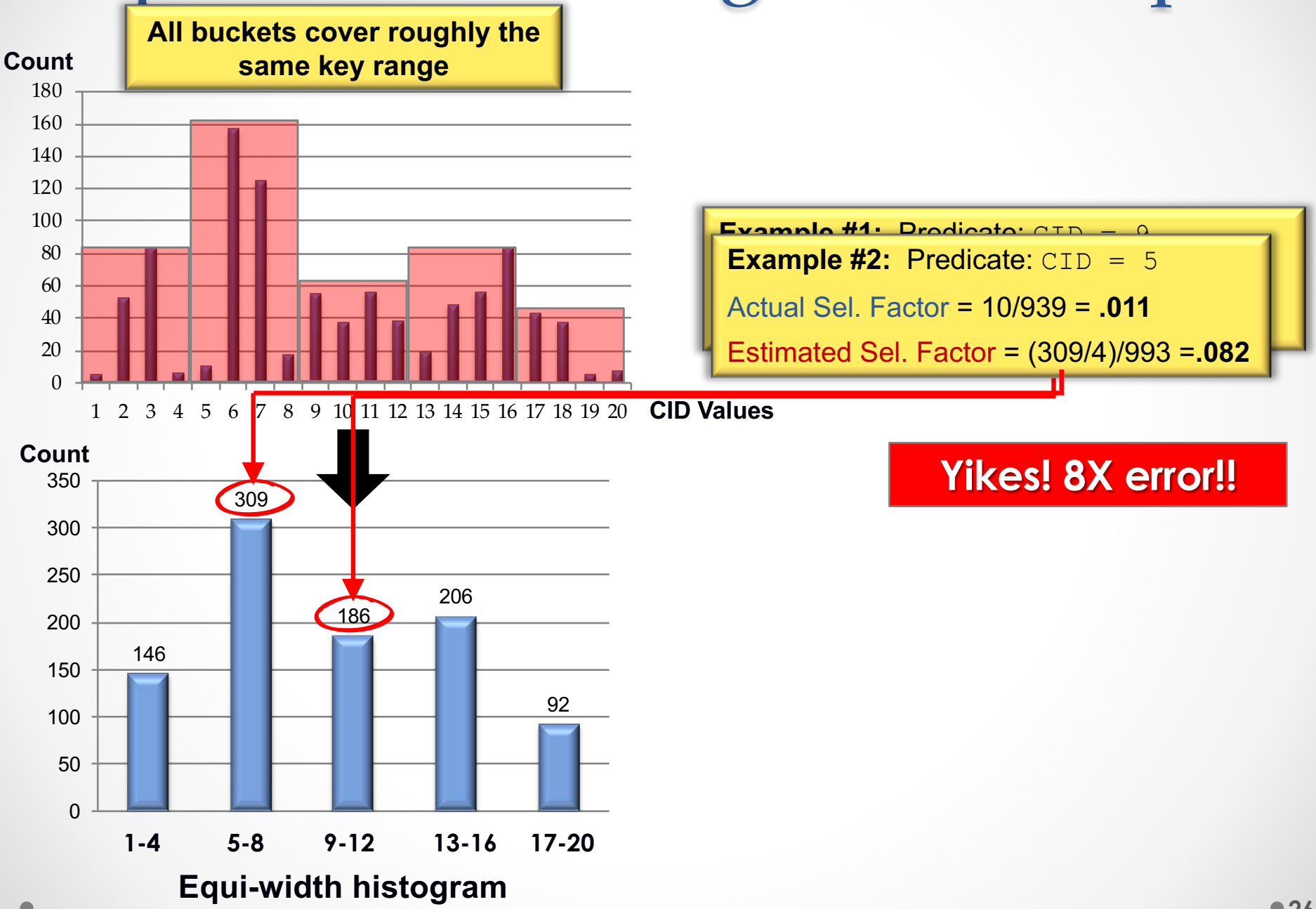
#1) Predicate: `CID = 9`
Actual Sel. Factor = 55/939 = **.059**

#2) Predicate: `2 <= CID <= 3`
Actual Sel. Factor = 135/939 = **.144**

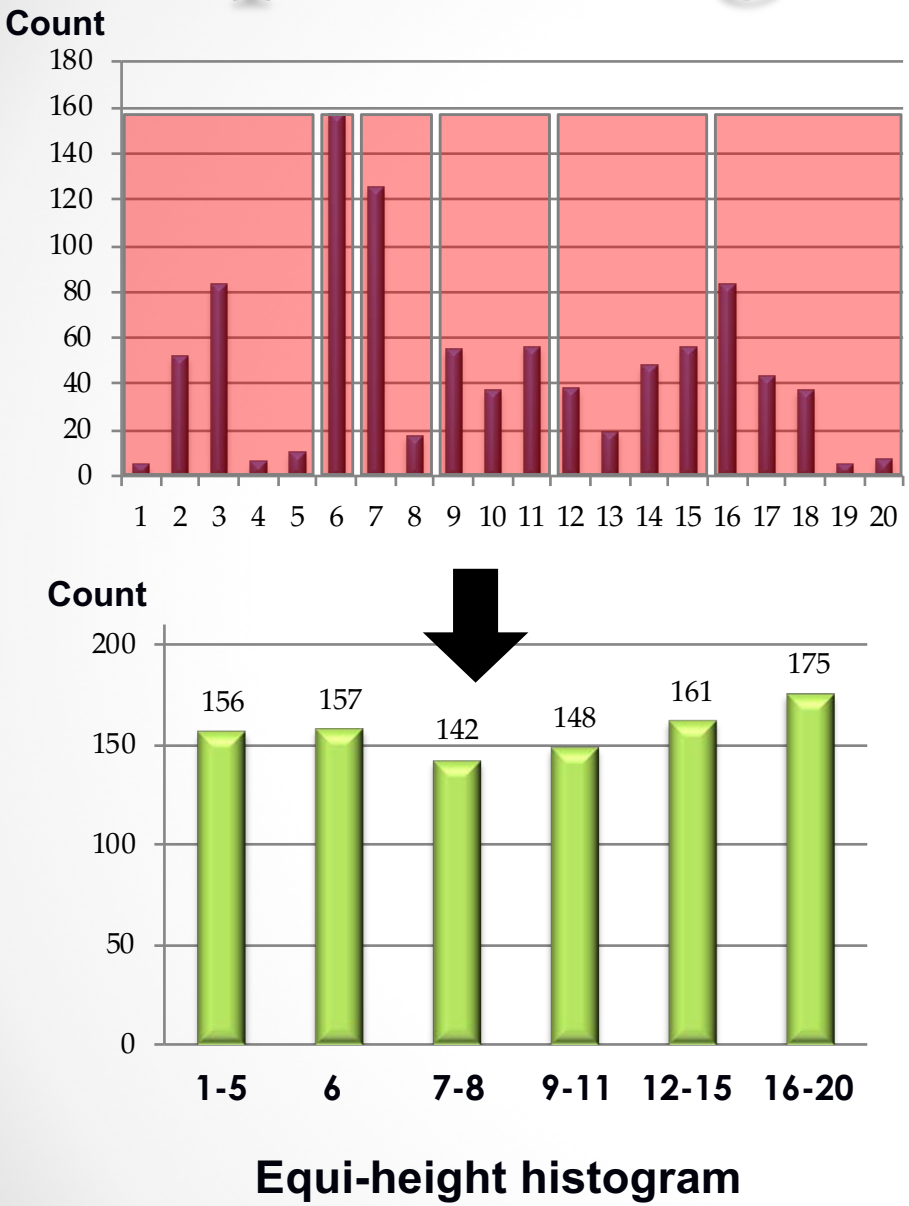
In general, there is not enough space in the catalogs to store summary statistics for each distinct attribute value

The solution: *histograms*

Equi-Width Histogram Example



Equi-Height Histograms

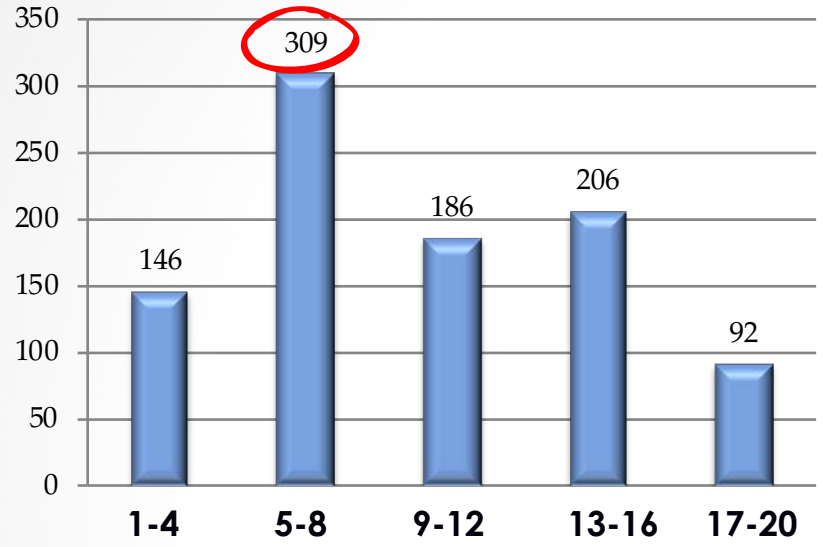


Equi-height histogram

Divide ranges so that all buckets contain roughly the same number of values

Equi-width vs. Equi-Height

Equi-width



Example #1: Predicate: CID = 5

Actual Sel. Factor = 10/939 = **.011**

Estimated Sel. Factor = (309/4)/993 =**.082**

Equi-height

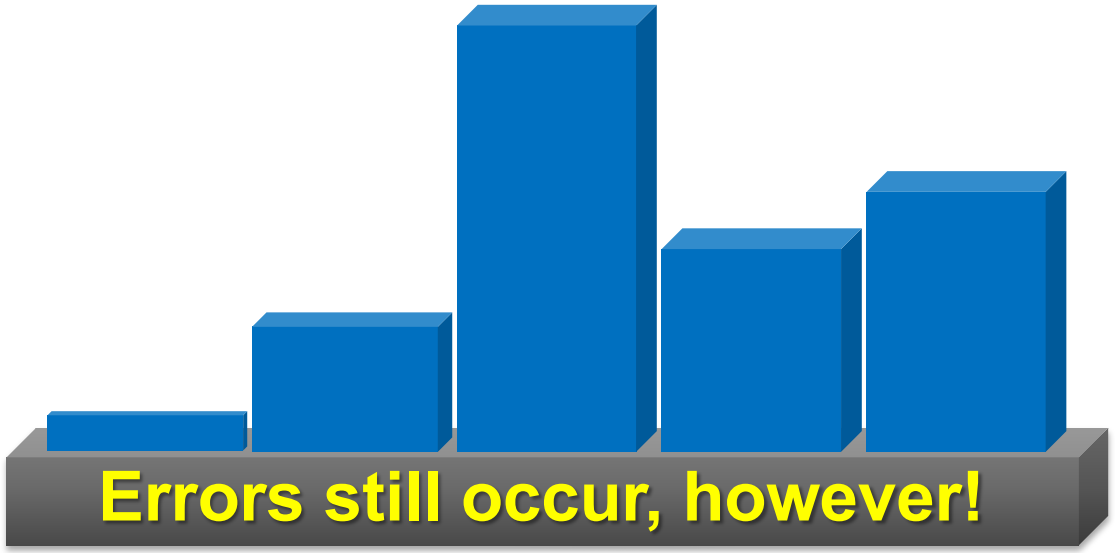
Example #1: Predicate: CID = 5

Actual Sel. Factor = 10/939 = **.011**

Estimated Sel. Factor = (156/5)/993 = **.033**

Histogram Summary

- Histograms are a critical tool for estimating selectivity factors for selection predicates



- Other statistics stored by the DBMS for each table include **# of rows**, **# of pages**, …

Query Optimization: The Main Steps

✓ 1

Enumerate logically equivalent plans by applying equivalence rules

✓ 2

For each logically equivalent plan, enumerate all alternative physical query plans

✓ 3

Estimate the cost of each of the alternative physical query plans.

- Estimate the selectivity factor and output cardinality of each predicate
- Estimate the cost of each operator

4

Run the plan with lowest estimated overall cost

Estimating Costs

- Two key costs that the optimizer considers:
  - **I/O time** – cost of reading pages from mass storage
  - **CPU time** – cost of applying predicates and operating on tuples in memory

- Actual values are highly dependent o CPU and I/O subsystem on which the query will be run
  - Further complicating the job of the query optimizer

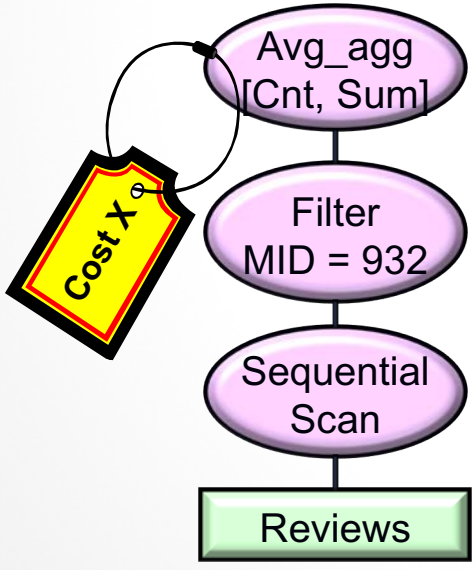
- For a parallel database system such as SQL DW, the cost of *redistributing/shuffling* rows must also be considered

An Example

Reviews

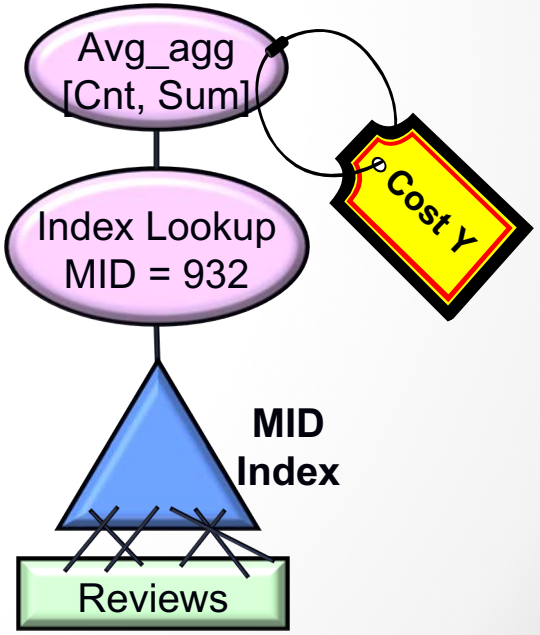
Date	CID	MID	Rating

- **Query**:
  - `SELECT Avg(Rating)`
    `FROM Reviews`
    `WHERE MID = 932`
- Two physical query plans:



Plan #1

Which plan is cheaper ???

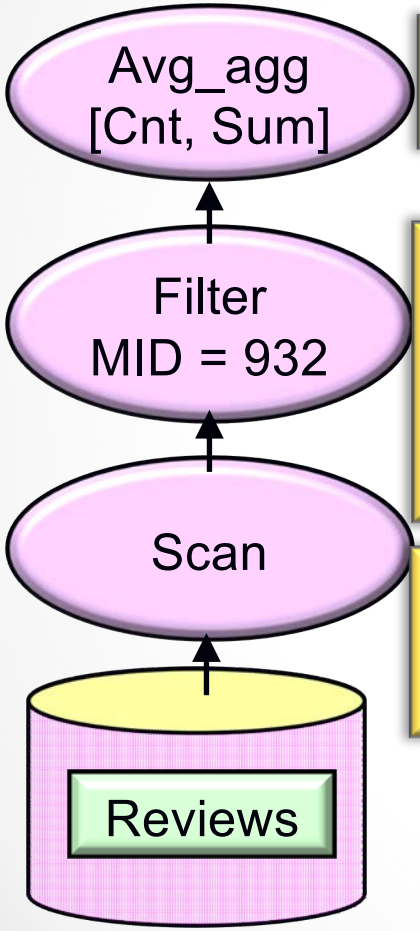


Plan #2

Plan #1

Optimizer estimates total execution time of **9 seconds**

=



- Average computation is applied to **100 rows**

- At **0.1 microseconds/row**, avg consumes **.00001 seconds** of CPU time

- Filter is applied to **10M rows**
- The optimizer **estimates that 100 rows** will satisfy the predicate

- At **0.1 microseconds/row**, filter consumes **1 second** of CPU time

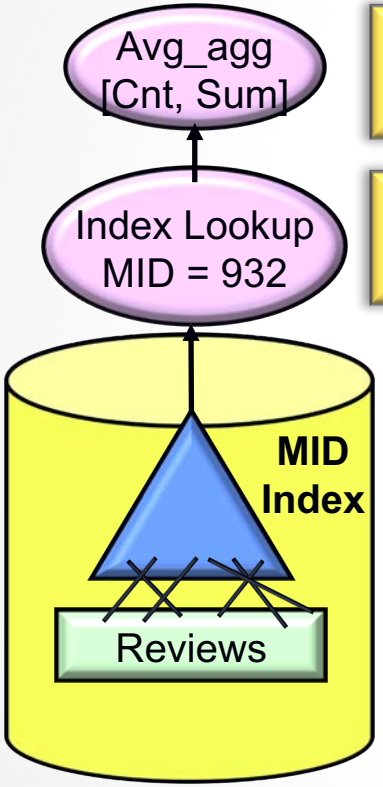
- Table is **100K pages** with **100 rows/page**
- Sorted on date

- *Reviews* is scanned sequentially at **100 MB/second**
- I/O time of scan is **8 seconds**

Plan #2

Optimizer estimates total execution time of **0.3 seconds**

=



- Average computation is applied to **100 rows**

- At **0.1 microseconds/row**, average consumes **.00001 seconds** of CPU time

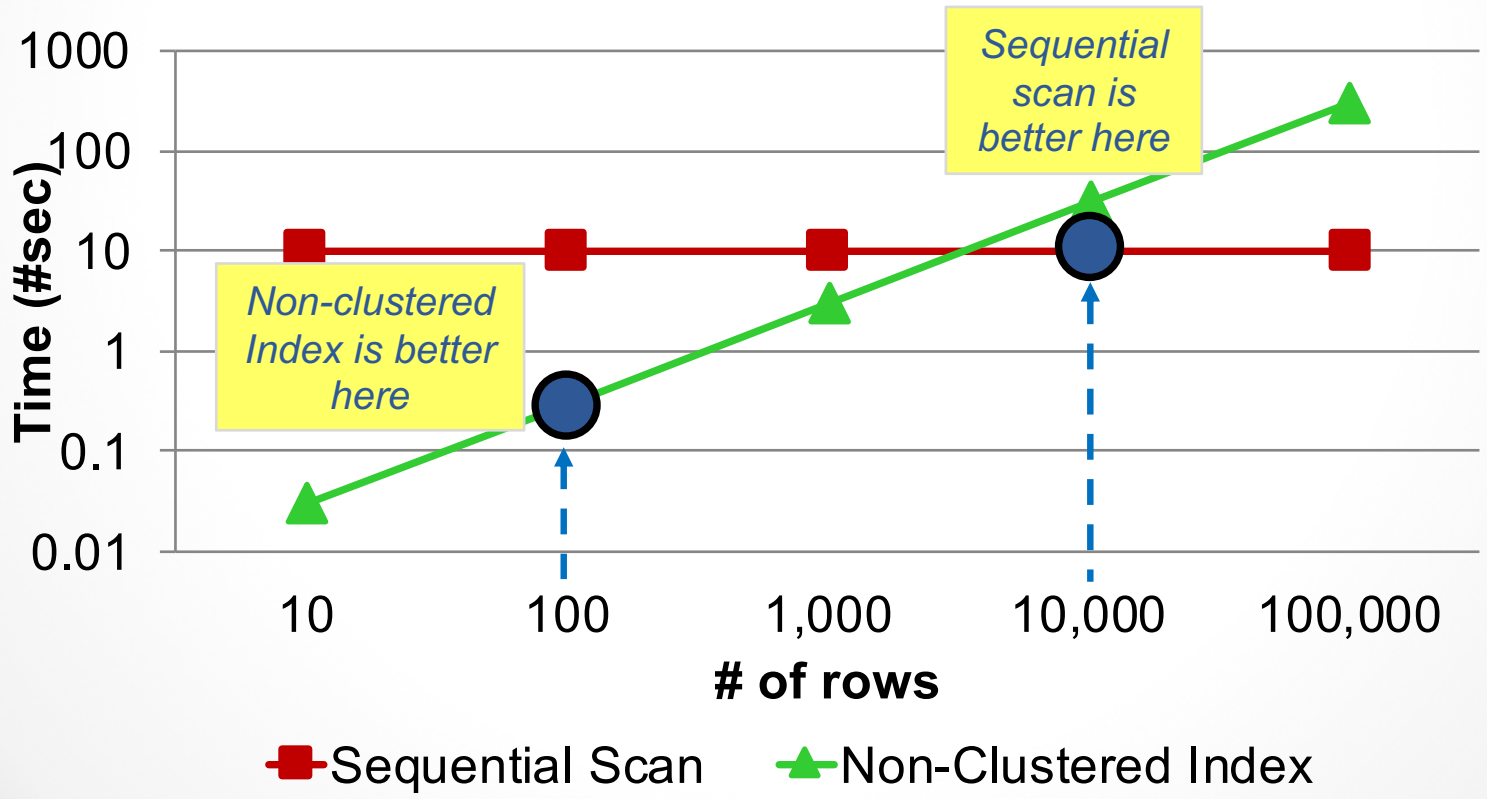
- **100 rows** are estimated to satisfy the predicate

- **100 rows** are retrieved using the MID index
- Since table is sorted on *date* field (and not *MID* field), each I/O requires a random disk I/O – about **.003 seconds** per disk I/O
- I/O time will be **.3 seconds**

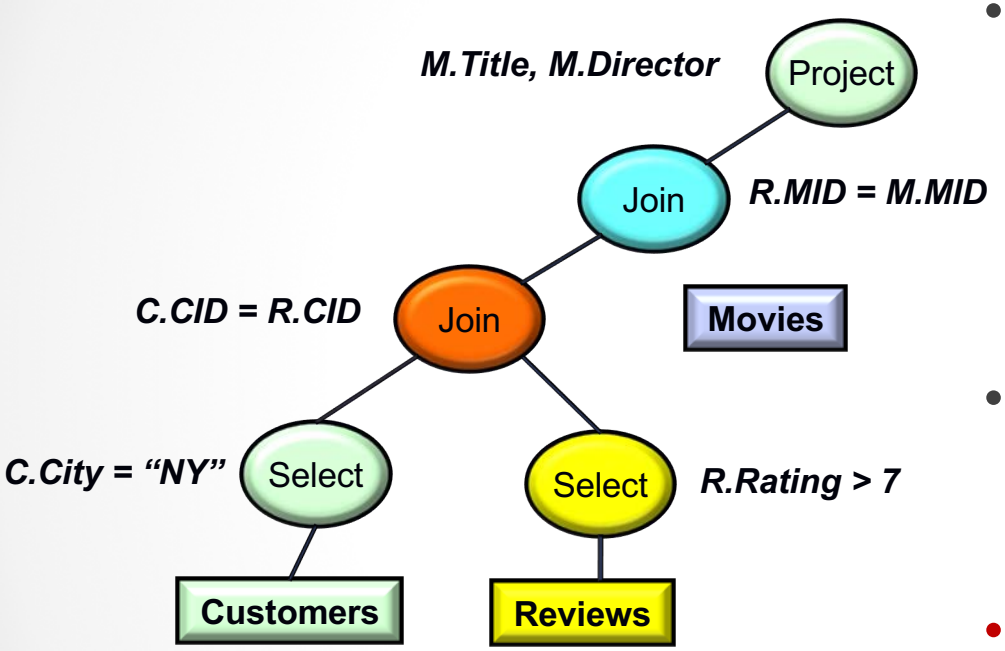
The estimate for Plan #1 was 9 seconds, so Plan #2 is clearly the better choice

But …

- **What if the estimate of the number of rows that satisfy the predicate `MID = 932` is WRONG?**
  - E.g. 10,000 rows instead of 100 rows

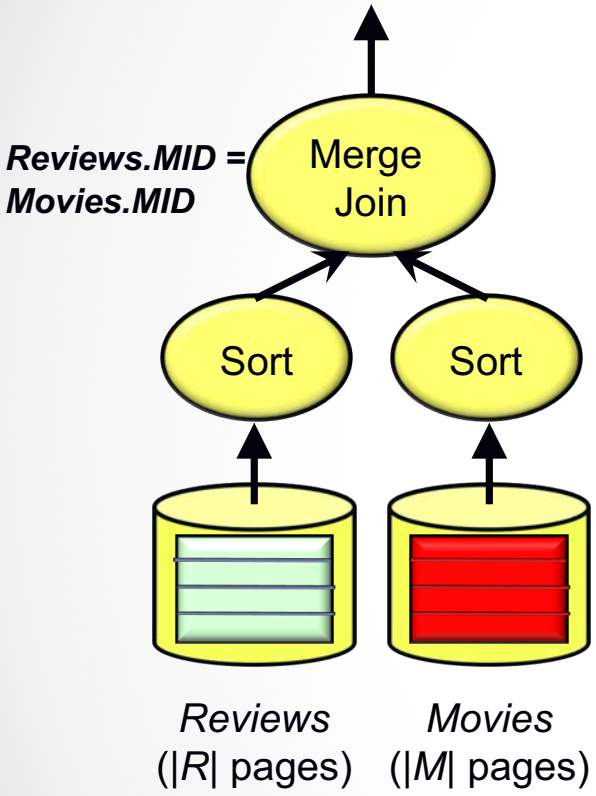


Estimating Join Costs



- Three basic join methods:
  - Nested-loops join
  - Sort-merge join
  - Hash-join
- Very different performance characteristics
- Critical for optimizer to carefully pick which method to use when

Sort-Merge Join Algorithm



Sort *Reviews* on *MID* column
(unless already sorted)
Sort *Movies* on *MID* column
(unless already sorted)
"Merge" two sorted tables:
Scan each table sequential in tandem
{
For current row *r* of *Reviews*
For current row *m* of *Movies*
if *r.MID* = *m.MID* produce output row
Advance *r* and *m* cursors
}

Cost = 4 * |*R*| I/Os

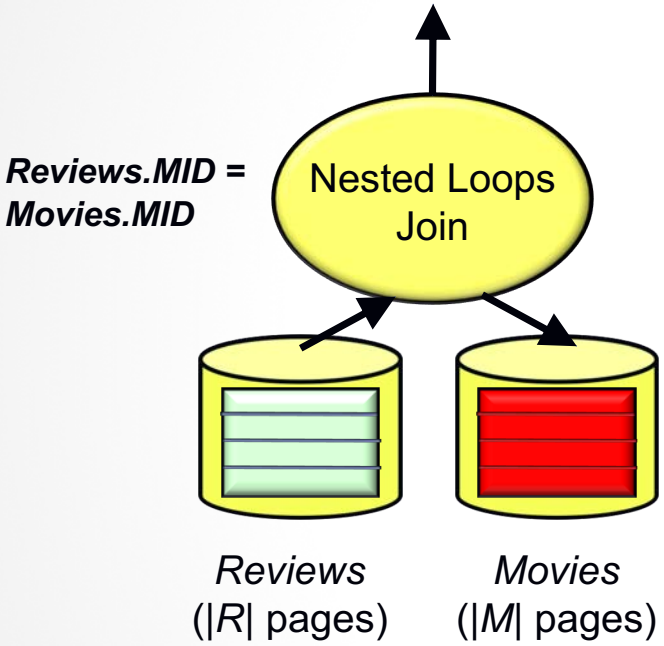
Cost = 4 * |*M*| I/Os

Cost = |*R*| + |*M*| I/Os

Total I/O cost = 5*|*R*| + 5*|*M*| I/Os

Main Idea: Sort *R* and *M* on the join column (*MID*), then scan them to do a ``merge'' (on join column), and output result tuples.

Nested-Loops Join

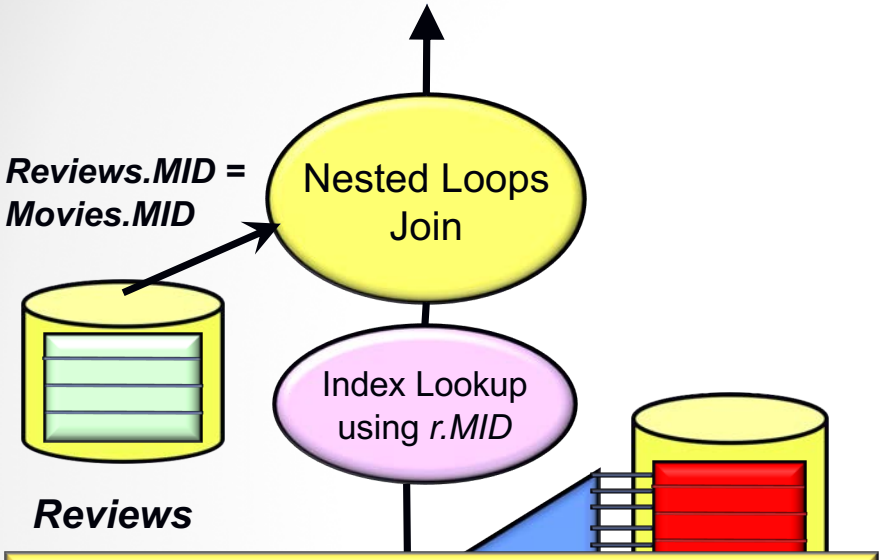



```
For each page R_i, 1≤ i ≤ |R|, of Reviews
{
    Read page R_i from disk
    For each M_j, 1≤ j ≤ |M|, of Movies
    {
        Read page M_j from disk
        For all rows r on page R_i
        {
            For all rows m on page M_j
            {
                if r.MID = m.MID produce output row
            }
        }
    }
}
```

I/O Cost = |R| + |R| * |M|

Main Idea: Scan R, and for each tuple in R probe tuples in M (by scanning it). Output result tuples.

Index-Nested Loops




```
For each page R_i, 1≤ i ≤ |R|, of Reviews
{
    Read page R_i from disk
    For all rows r on page R_i
    {
        Use MID index on Movies
        to fetch rows with MID attributes = r.MID
        Form output row for each returned row
    }
}
```

Notice that since *Reviews* is ordered on the *Date* column (and not *MID*), so each row of the *Movies* table retrieved incurs **two random disk I/Os:**

- **one to the index** and
- **one to the table**

Cost = |*R*| + |*R*| * (||*R*||/|*R*|) * 2

- **2 I/Os:** 1 index I/O + 1 movie I/O as *Reviews* table is sorted on date column
- **||*R*||** is # of rows in *R*
- **||*R*||/|*R*|** gives the average number of rows of R per page

Main Idea: Scan R, and for each tuple in R probe tuples in M (by probing its index). Output result tuples.

Estimating Result Cardinalities

- Consider the query

```
SELECT *
FROM Reviews
WHERE 7/1 < date < 7/31 AND rating > 9
```

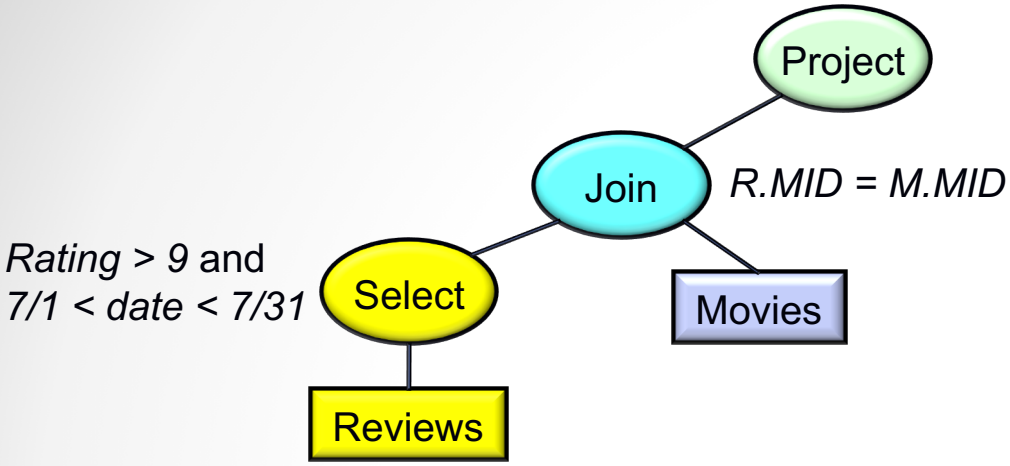
- Assume *Reviews* has **1M rows**
- Assume following selectivity factors:

	Sel. Factor	# of qualifying rows
7/1 < date < 7/31	0.1	100,000
Review > 9	0.01	10,000

- **How many output rows will the query produce?**
  - If predicates are not correlated
    - .1 * .01 * 1M = **1,000 rows**
  - If predicates are correlated could be as high as
    - .1 * 1M = **100,000 rows**

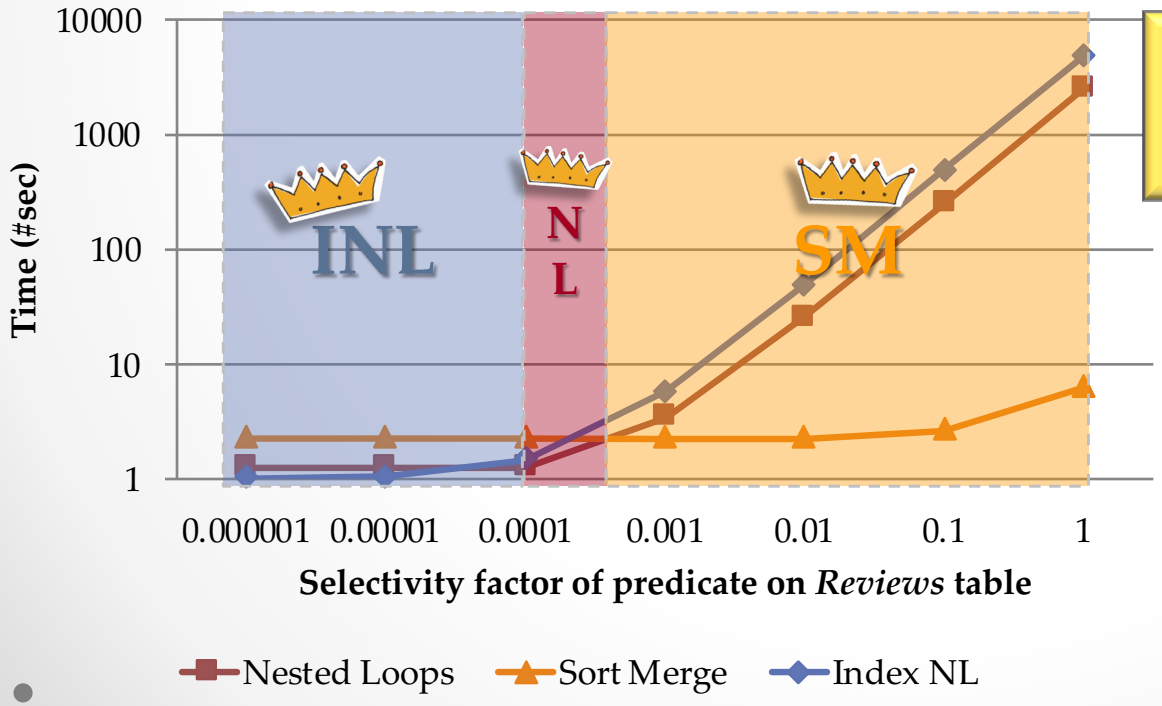
Why does this matter?

This is Why!



Assume that:

- *Reviews* table is **10,000 pages** with **80 rows/page**
- *Movies* table is **2,000 pages**
- The primary index on *Movies* is on the *MID* column

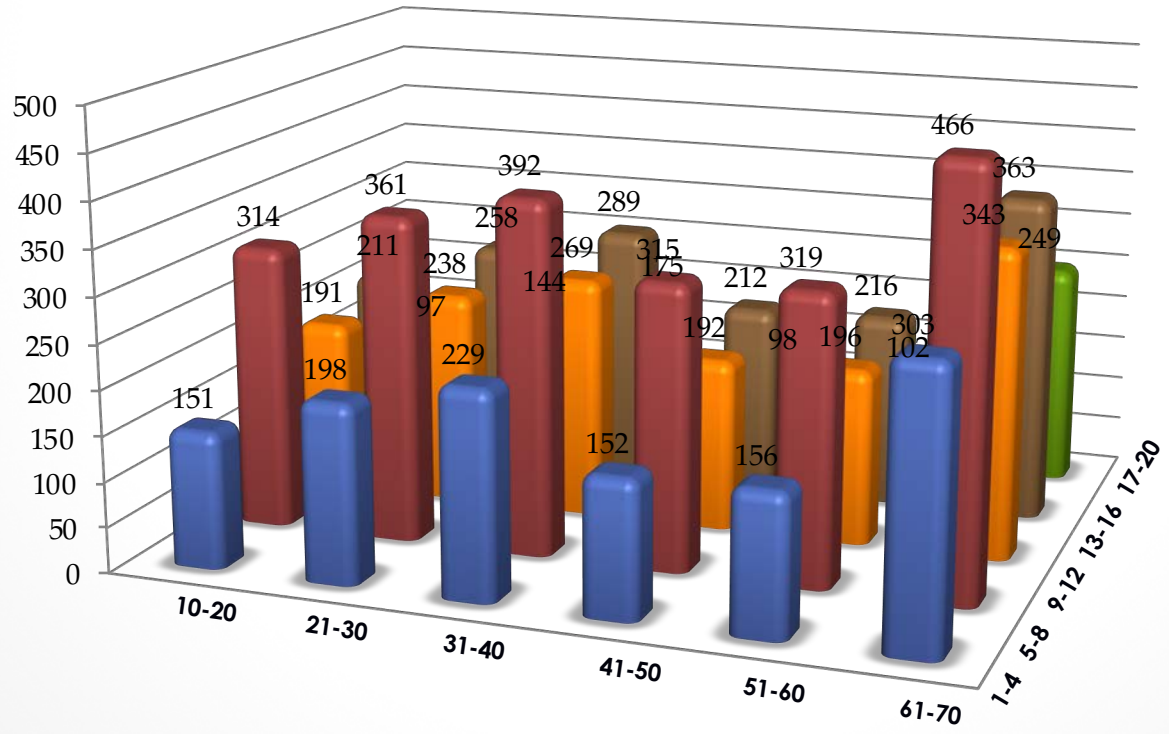


Note that ***each join algorithm has a region where it provides the best performance***

The consequences of incorrectly estimating the selectivity of the predicate on *Reviews* can be HUGE

Multidimensional Histograms

- Used to capture correlation between attributes
- A 2-D example



A Little Bit About Estimating Join Cardinalities

- **Question**: Given a join of *R* and *S*, what is the range of possible result sizes (in #of tuples)?
  - Suppose the join is on a key for *R* and *S*

    Students(*sid*, sname, did), *Dorm*(*did*,*d.addr*)

```
Select S.sid, D.address
From Students S,  Dorms D
Where S.did = D.did
```

What is the cardinality?

A student can only live in at most 1 dorm:
- each *S* tuple can match with at most 1 *D* tuple
- **cardinality (*S* join *D*) = cardinality of *S***

Estimating Join Cardinality

- **General case**: join on {*A*} (where {*A*} is key for neither)
  - estimate each tuple *r* of *R* generates uniform number of matches in *S* and each tuple *s* of *S* generates uniform number of matches in *R*, e.g.

SF = min(||**R**|| * ||**S**|| / NKeys(A,**S**), **= 100*20/10 = 200**

||**S**|| * ||**R**|| / NKeys(A,**R**)) **= 20*100/75 = 26.6**

e.g., `SELECT M.title, R.title`
`FROM Movies M, Reviews R`
`WHERE M.title = R.title`

Movies: 100 tuples, 75 unique titles → 1.3 rows for each title

Reviews: 20 tuples, 10 unique titles → 2 rows for each title

Query Optimization: The Main Steps

✓ **1**  **Enumerate** logically equivalent plans by applying equivalence rules

✓ **2**  For each logically equivalent plan, **enumerate** all alternative physical query plans

✓ **3** 

How **big is the plan space** for a query involving ***N*** tables?

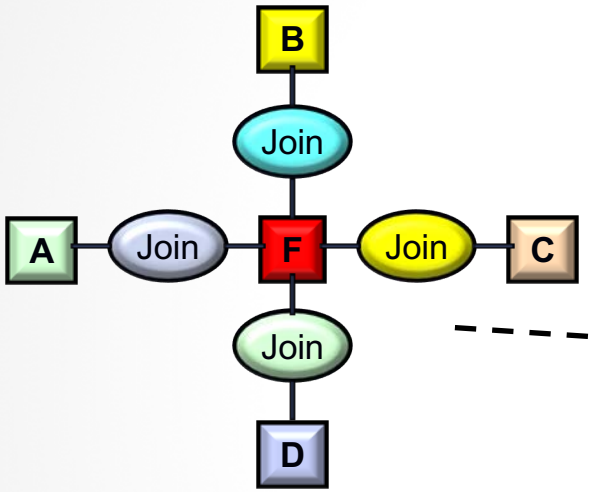
- Estimate the cost of each operator

It turns out that the answer depends on the "*shape*" of the query

4  R ... erall cost

Two Common Query "Shapes"

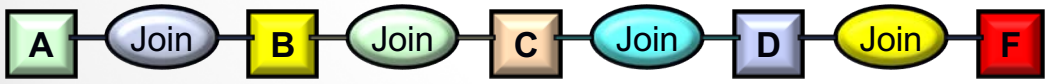
"Star" Join Queries



Number of logically equivalent alternatives

# of Tables	Star	Chain
2	2	2
4	48	40
5	384	224
6	3,840	1,344
8	645,120	54,912
10	18,579,450	2,489,344

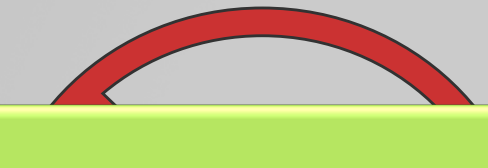
"Chain" Join Queries



In practice, "typical" queries fall somewhere between these two extremes

Pruning the Plan Space

- Consider only left-deep query plans to reduce the search space



Solution:

Use some form of **dynamic programming**
(either bottom up or top down)
to search the plan space heuristically

Sometimes these heuristics will cause the best plan to be missed!!

ii) 22,044,960 different physical plans!!

				32
8	645,120	10,080	54,912	128
10	18,579,450	725,760	2,489,344	512

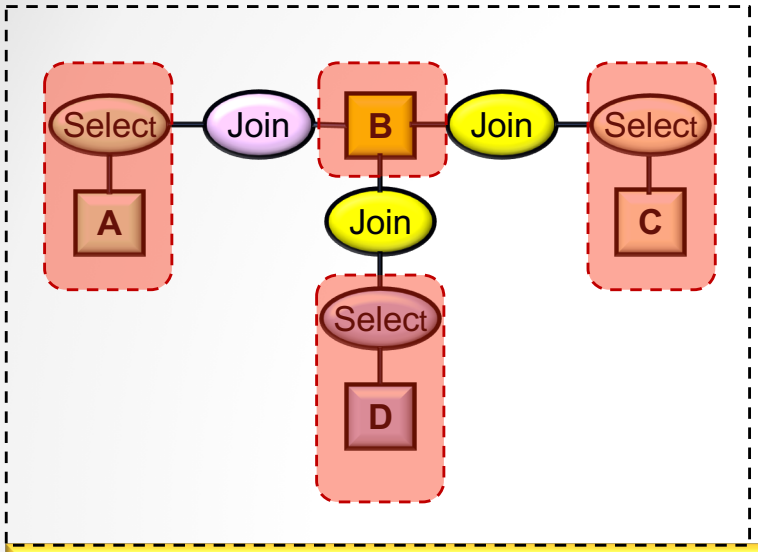
Bottom-Up QO Using Dynamic Programming

- Optimization is performed in *N* passes (if *N* relations are joined):
  - **Pass 1:** Find the best (lowest cost) 1-relation plan for each relation.
  - **Pass 2:** Find the best way to join
    outer/left table) to another relation
    all 2-relation plans.
  - **Pass *N*:** Find best way to join res
    N'th relation to generate *all N-rela*
- At each pass, for each subset of re
  - Lowest cost plan overall, plus
  - Lowest cost plan for each *interesting order* of the rows
- Order by, group by, aggregates etc. handled as the final step

Interesting orders include orders that facilitate the execution of joins, aggregates, and order by clauses subsequently by the query

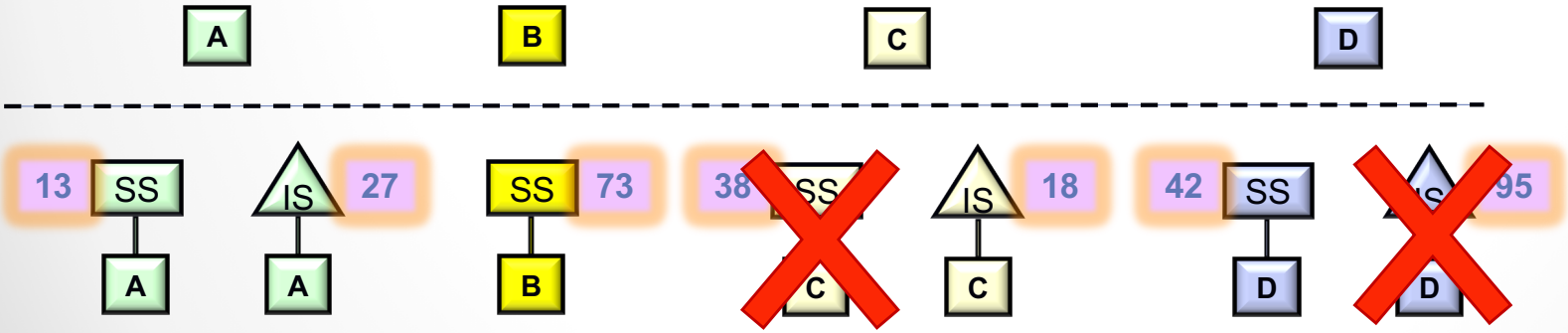
In spite of pruning plan space, this approach is still exponential in the # of tables.

An Example:



Legend:
SS – sequential scan
IS – index scan
5 – cost

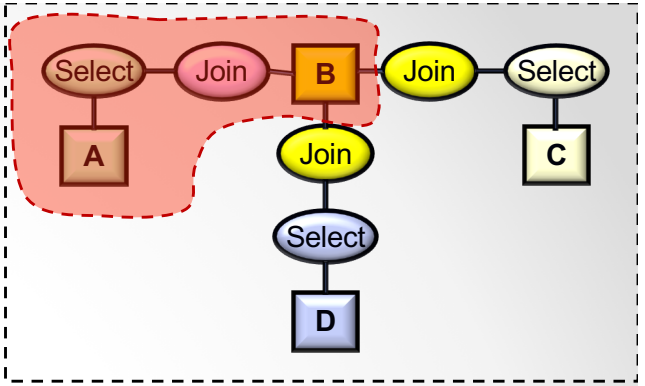
First, generate all single relation plans:



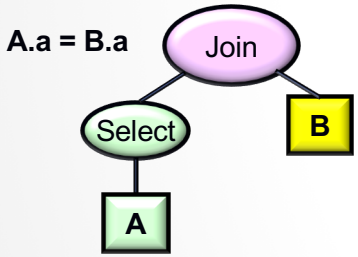
Then, All Two Relation Plans

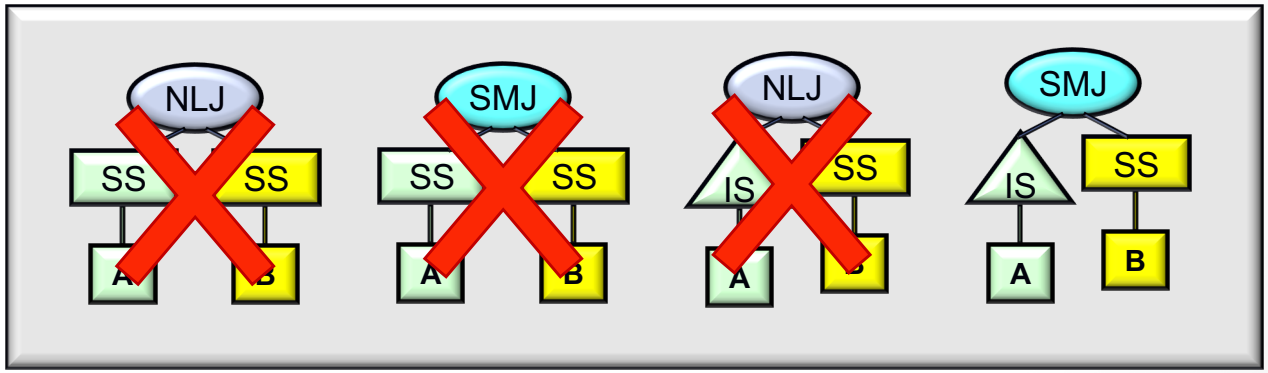
All single relation plans after pruning

Two Relation Plans Starting With A



Single relation plans





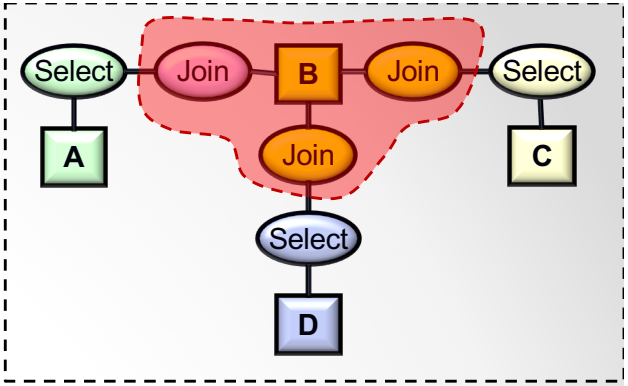
1013 315 822 293

Prune

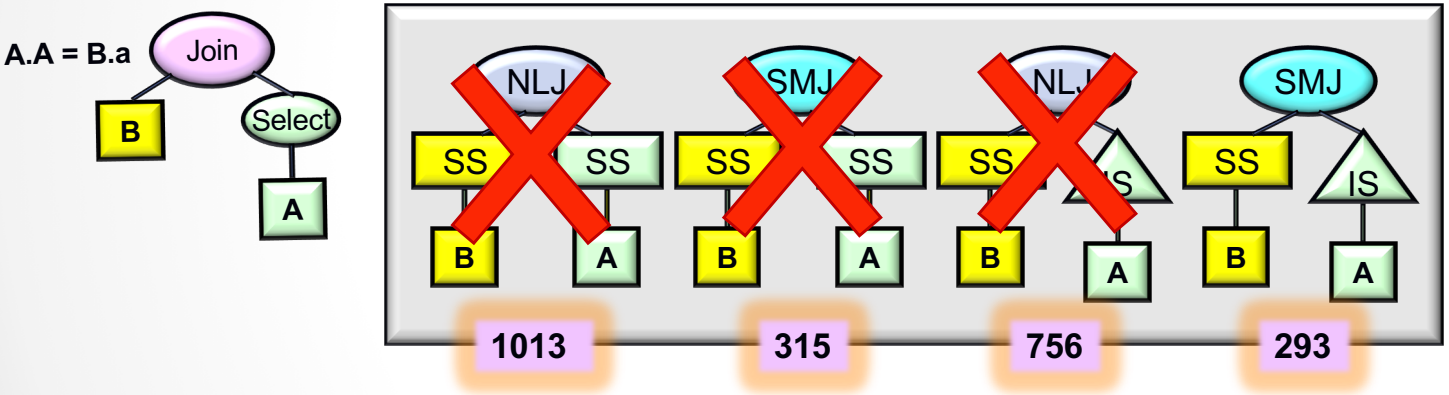
Let's assume there are 2 alternative join methods for the QO to select from:

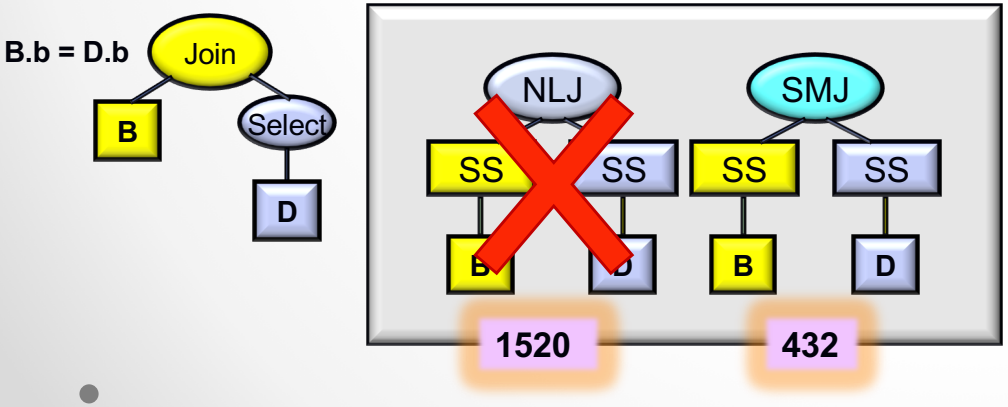
1. **NLJ** = Nested Loops Join
2. **SMJ** = Sort Merge Join

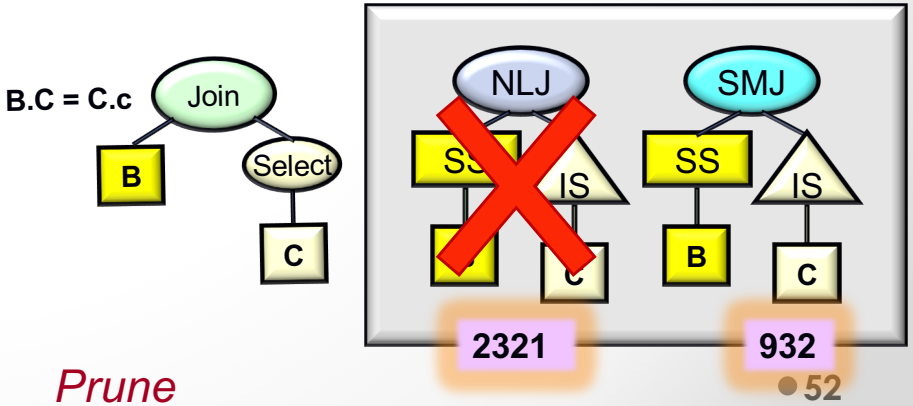
Two Relation Plans Starting With B



Single relation plans

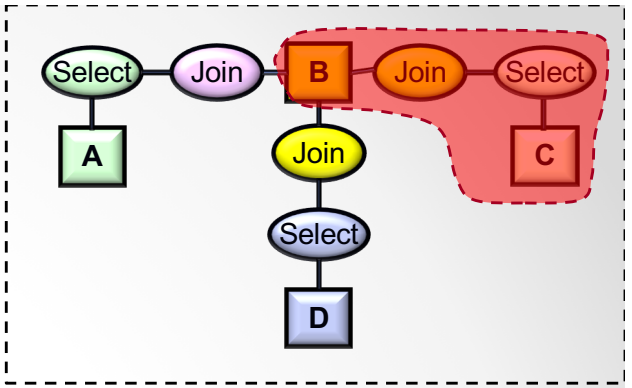




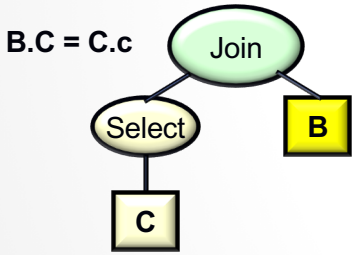


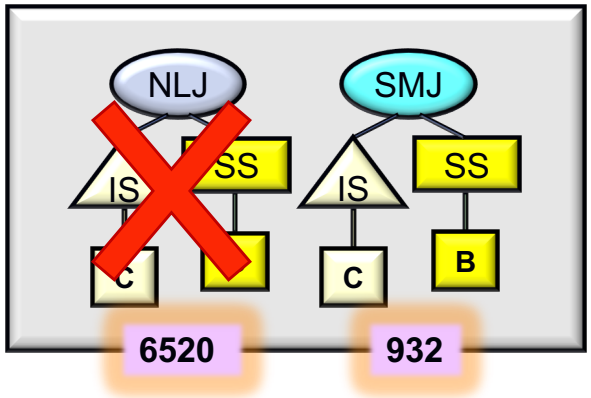
Prune

Two Relation Plans Starting With C



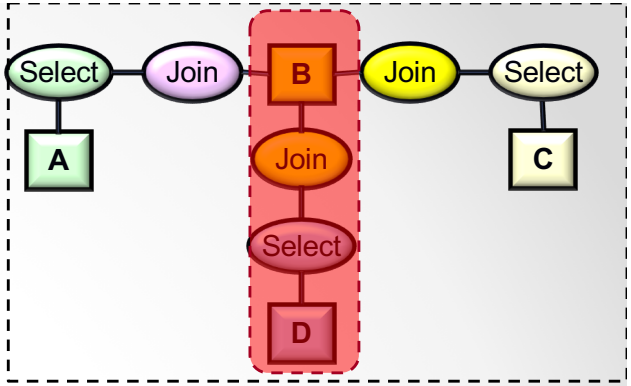
Single relation plans



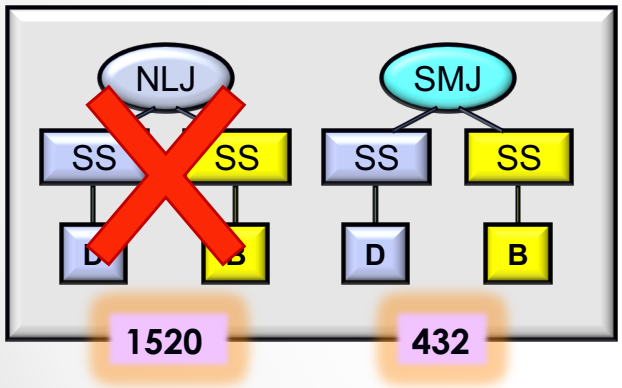


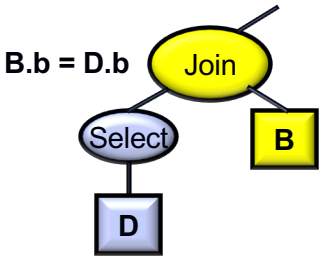
Prune

Two Relation Plans Starting With D



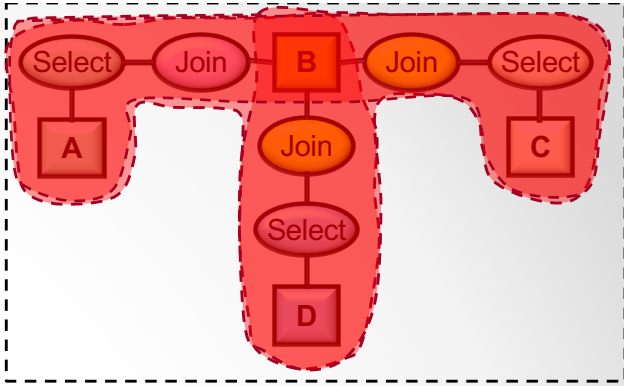
Single relation plans

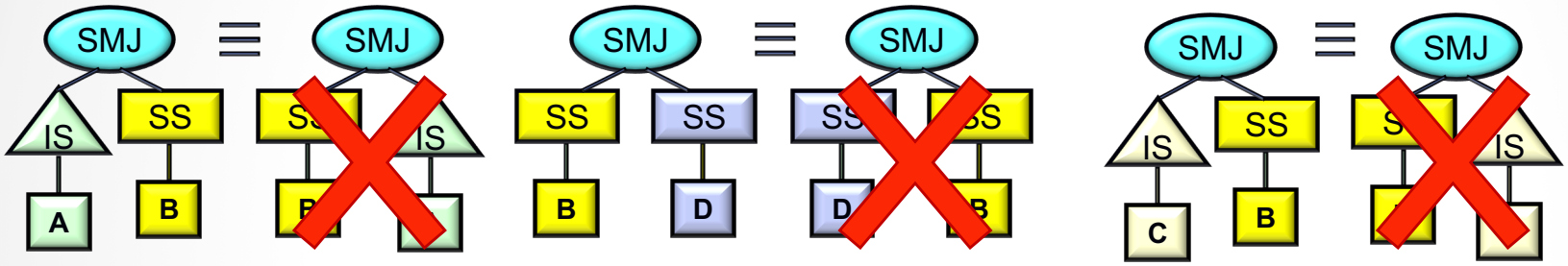




Prune

Further Prune Two Relation Plans

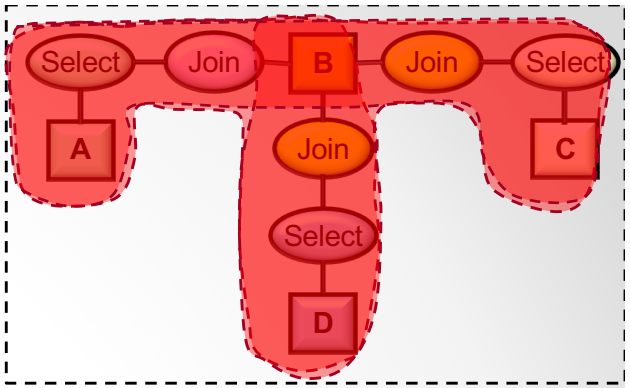


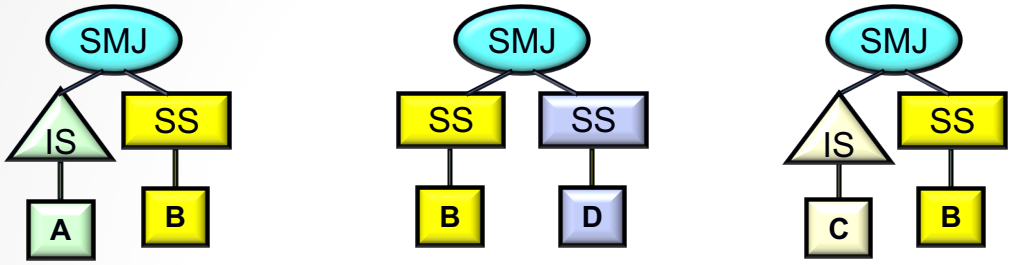


Pruned two relation plans

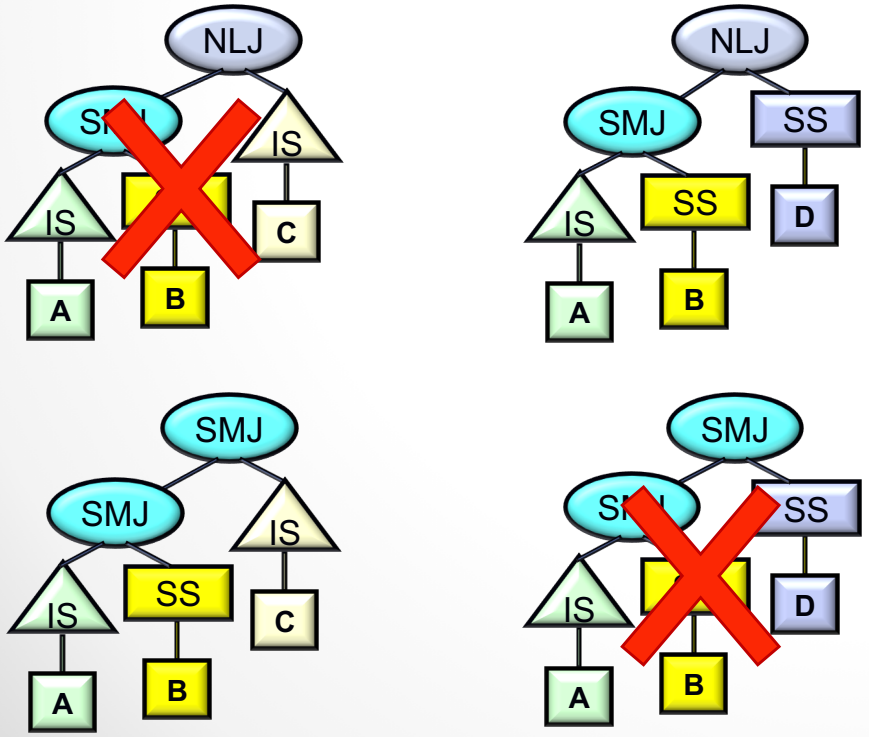
Next, All Three ...on Plans

1) Consider the Two Relation Plans That Start With A



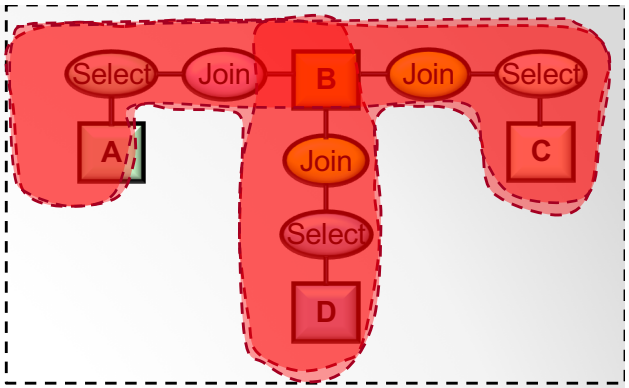


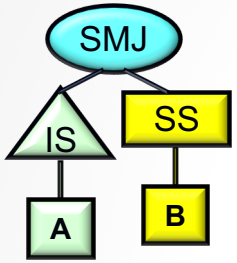
Fully pruned two relation plans

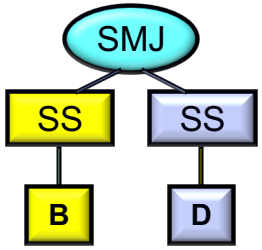


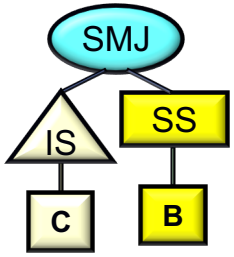
Next, All Three Rel lans

2) Consider the Two Relation Plans That Start With B

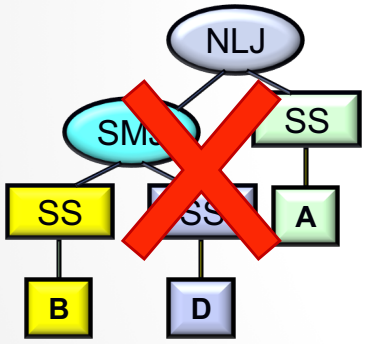


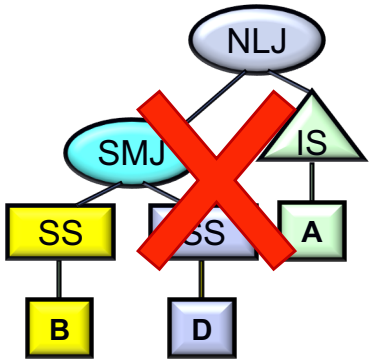


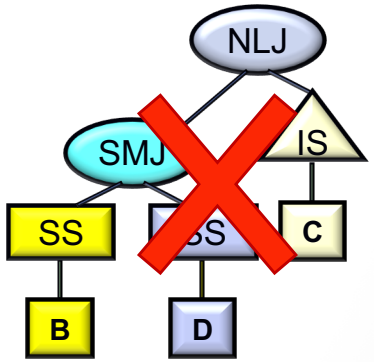


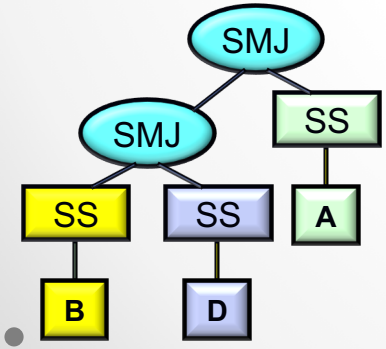


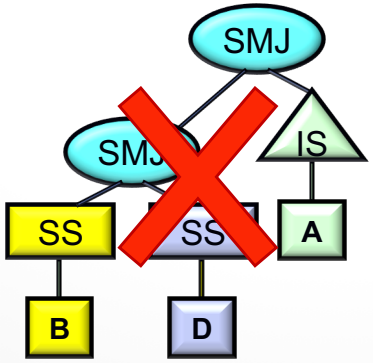
Fully pruned two relation plans

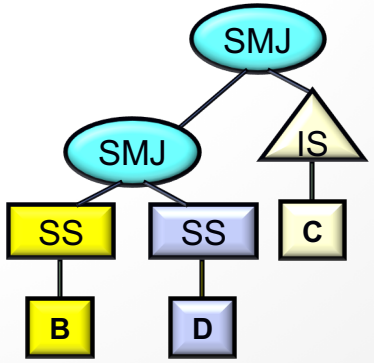






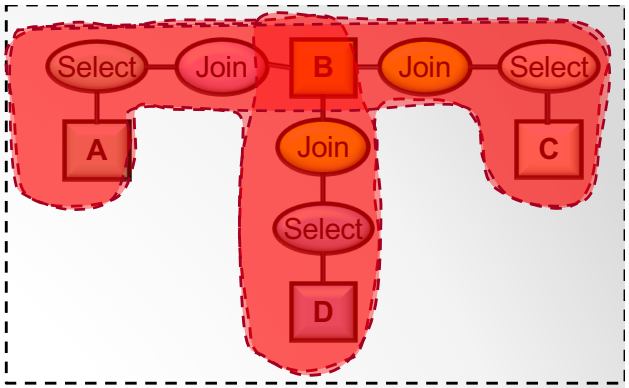


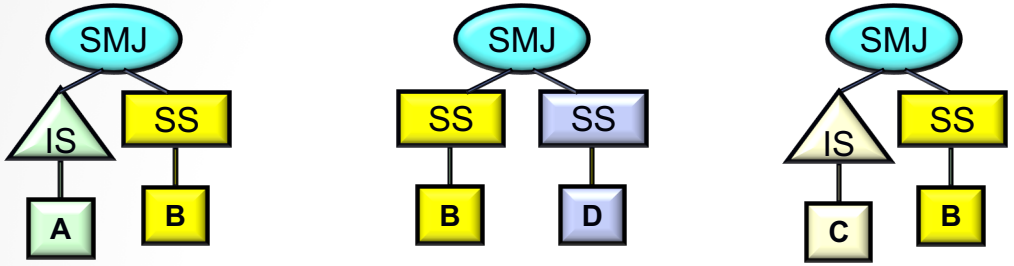




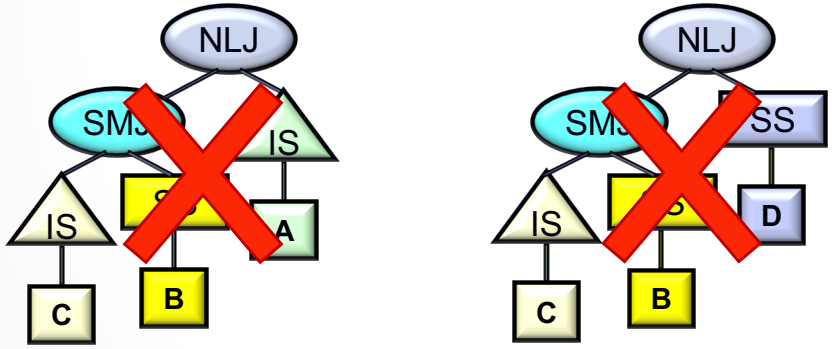
Next, All Three Relation P

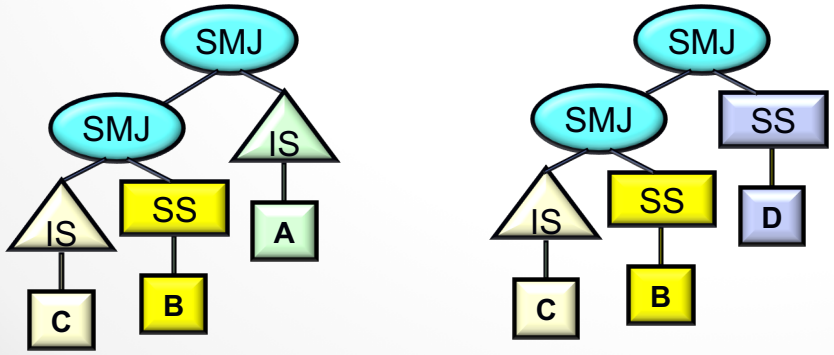
3) Consider the Two Relation Plans That Start With C





Fully pruned two relation plans





You Have Now Seen the Theory

- But the reality is:
  - **Optimizer still pick bad plans too frequently** for a variety of reasons:
    - Statistics can be missing, out-of-date, incorrect
    - Cardinality estimates assume uniformly distributed values but data values are skewed
    - Attribute values are *correlated* with one another:
      - *Make = "Honda"* and *Model = "Accord"*
    - Cost estimates are based on formulas that do not take into account the characteristics of the machine on which the query will actually be run
  - Regressions happen due hardware and software upgrades

What can be done to improve the situation?

Opportunities for Improvement

- Develop tools that give us a better understanding of what goes wrong
- Improve plan stability
- Use of feedback from the QE to the QO to improve statistics and cost estimates

Towards a Better Understanding of QO Behavior

- **Picasso Project** – Jayant Haritsa, IIT Bangalore
  - Bing "Picasso Haritsa" to find the project's web site
  - Tool is available for SQL Server, Oracle, PostgreSQL, DB2, Sybase
- Simple but powerful idea:
- For a given query such as

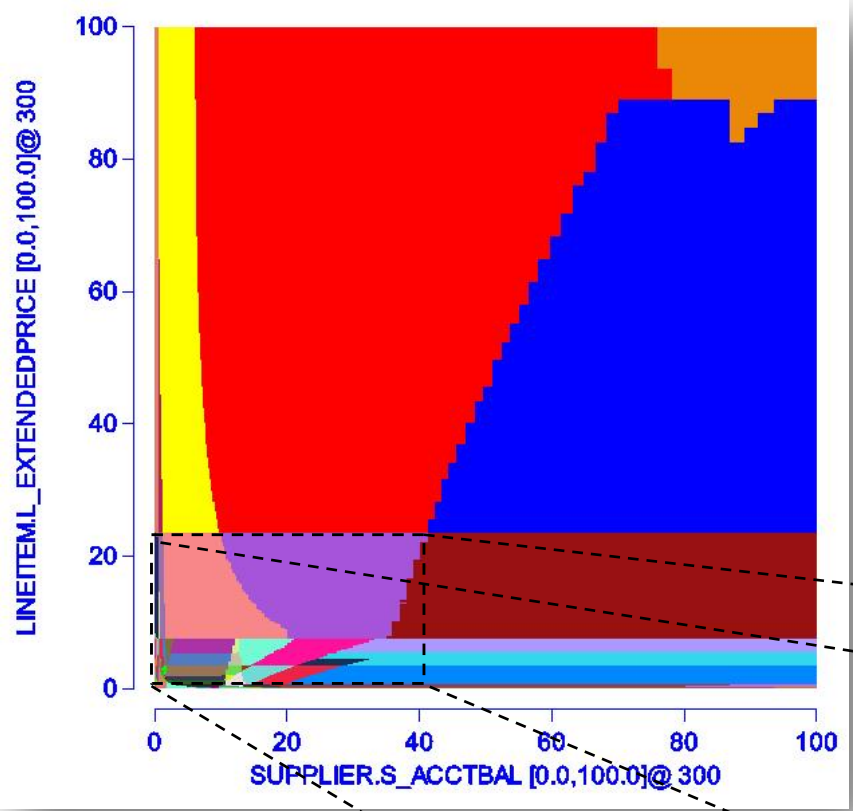
```
SELECT * from A, B
WHERE A.a = B.b and
      A.c <= constant-1 and
      B.d <= constant-2
```

- Systematically vary *constant-1* and *constant-2*
- Obtain query plan and estimated cost from the query optimizer for each combination of input parameters
- Plot the results

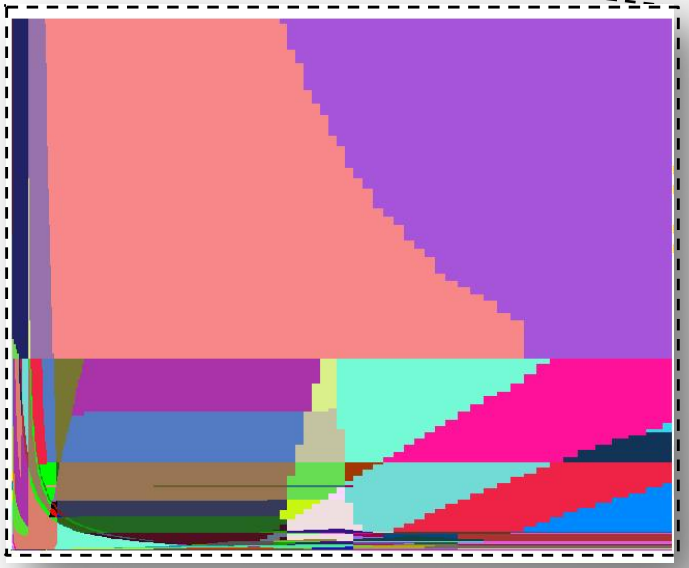
Example: TPC-H Query 8

```
select
        o_year,
        sum(case
                when nation = 'BRAZIL' then volume
                else 0
        end) / sum(volume)
from
        (
          select YEAR(O_ORDERDATE) as o_year,
                   L_EXTENDEDPRICE * (1 - L_DISCOUNT) as volume, n2.N_NAME as nation
          from PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION n1, NATION n2, REGION
        where
                P_PARTKEY = L_PARTKEY and S_SUPPKEY = L_SUPPKEY
                and L_ORDERKEY = O_ORDERKEY and O_CUSTKEY = C_CUSTKEY
                and C_NATIONKEY = n1.N_NATIONKEY and n1.N_REGIONKEY = R_REGIONKEY
                and R_NAME = 'AMERICA' and S_NATIONKEY = n2.N_NATIONKEY
                and O_ORDERDATE between '1995-01-01' and '1996-12-31'
                and P_TYPE = 'ECONOMY ANODIZED STEEL'
                and S_ACCTBAL <= constant-1
                and L_EXTENDEDPRICE <= constant-2
        ) as all_nations
group by o_year
order by o_year
```

Resulting Plan Space



- **SQL Server 2008 R2**
- A total of 90,000 queries
  - 300 different values for both *L_ExtendedPrice* and *S_AcctBal*
- 204 different plans!!
  - Each distinct plan is assigned a unique color
- Zooming in to the [0,20:0,40] region:

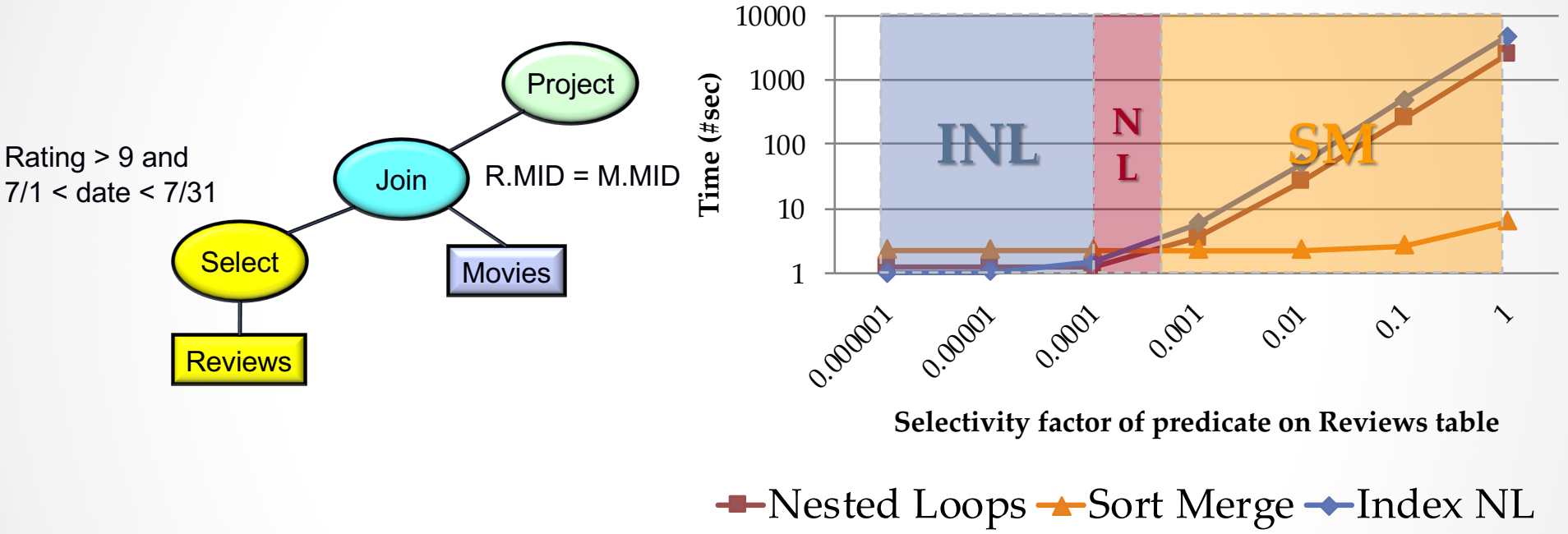


Key takeaway: If plan choice is so sensitive to the constants used, it will undoubtedly be sensitive to errors in statistics and cardinality estimates ☹

Intuitively, this seems very bad!

How Might We Do Better?

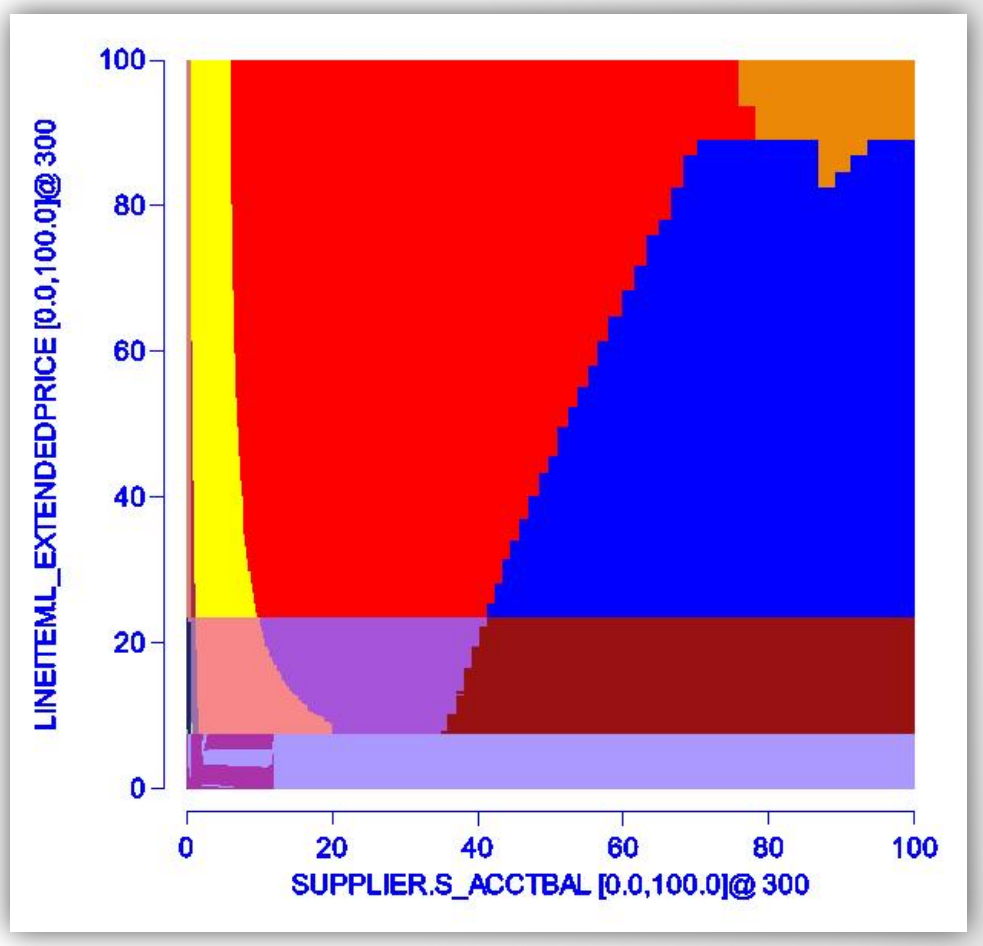
- Recall this graph of join algorithm performance



- While the two "nested loops" algorithms are faster at low selectivity factors, they are not as "stable" across the entire range of selectivity factors

"Reduced" Plan Diagrams

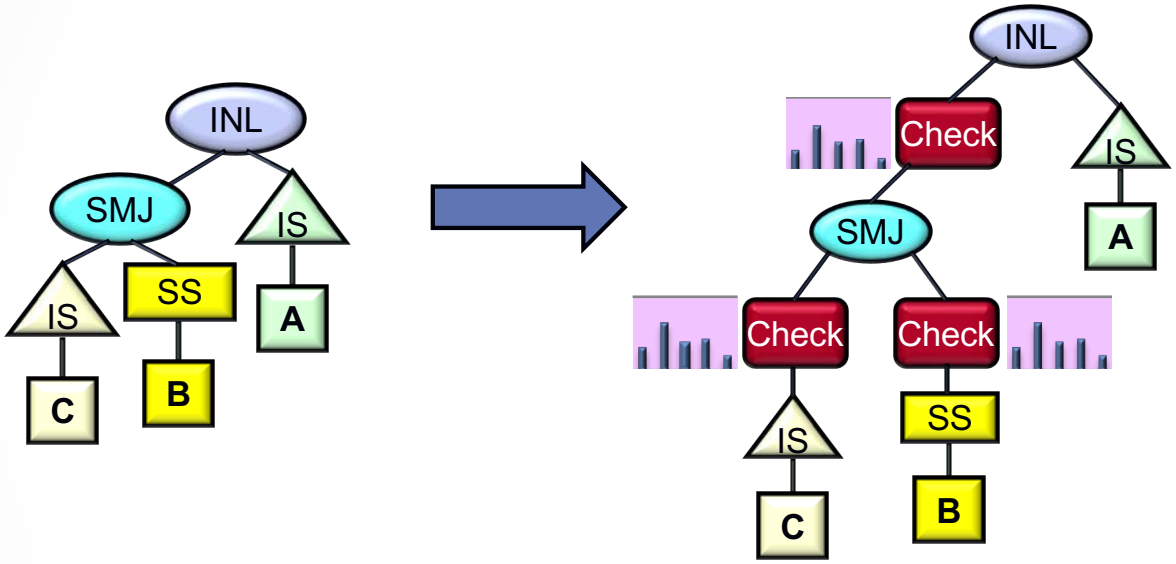
- Robustness is somehow tied to the number of plans
  - Fewer plans => more robust plans
- For TPC-H query 8, it is possible to **use only 30 plans** (instead of 204) by picking more robust plans that are slightly slower (10% max, 2% avg)
- Since each plan covers a larger region it will be less sensitive to errors in estimating cardinalities and costs



Reduced plan space for TPC-H query 8

How Might We Do Better?

- **At QO time**, have the QO annotate compiled query plans with **statistics** (e.g. expected cardinalities) and **check** operators



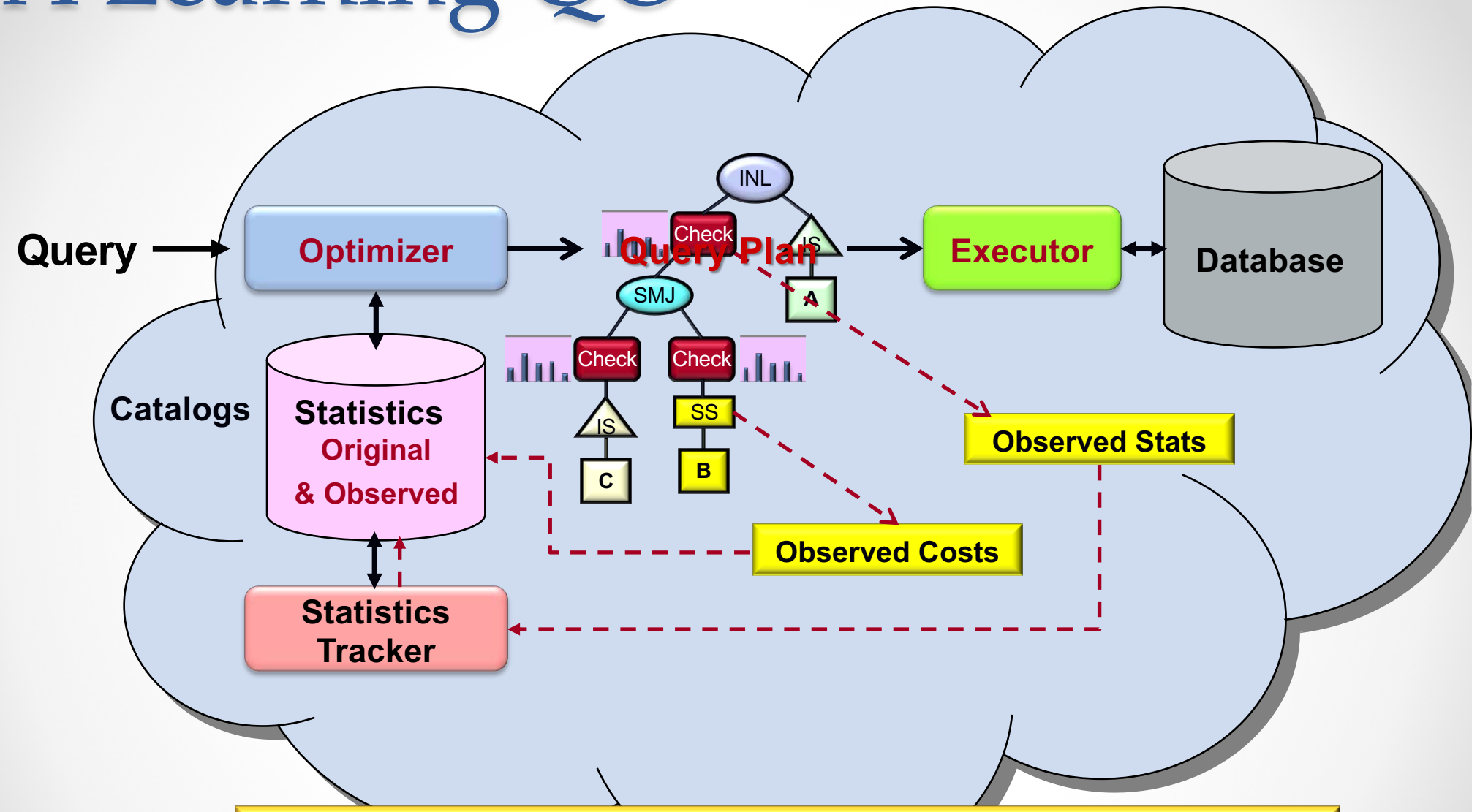
- **At runtime**, check operators collect the actual statistics and compare actual vs. predicted
- Opens up a number of avenues for improving QO performance

Especially in the CLOUD!

QO In the Cloud

- What is different?
  - On prem, a DB vendor has essentially no insight to how its product is used
  - In the cloud, vendor knows
    - Schema information (tables, indices, …)
    - The hardware being used
    - The complete query workload
    - For each query, the optimized plan & its estimated cost, the actually running cost, and the selectivity of each operator
- Use this information to build an optimizer that learns.

A Learning QO



Optimization of subsequent queries benefits from the observed statistics and operator costs

Key Points To Remember For The Quiz

- Query optimization is harder than rocket science
  - The other components are trivial in comparison
- Three key phases of QO
  - Enumeration of logical plan space
  - Enumeration of alternative physical plans
  - Selectivity estimation and costing
- The QO team of every DB vendor lives in fear of regressions
  - How exactly do you expect them to make forward progress?