

香港中文大學(深圳)  
The Chinese University of Hong Kong, Shenzhen



Ack: Prof. Jignesh Patel @ CMU  
Prof. Andy Pavlo @CMU

# CSC3170

## 14: Query Planing & Optimization

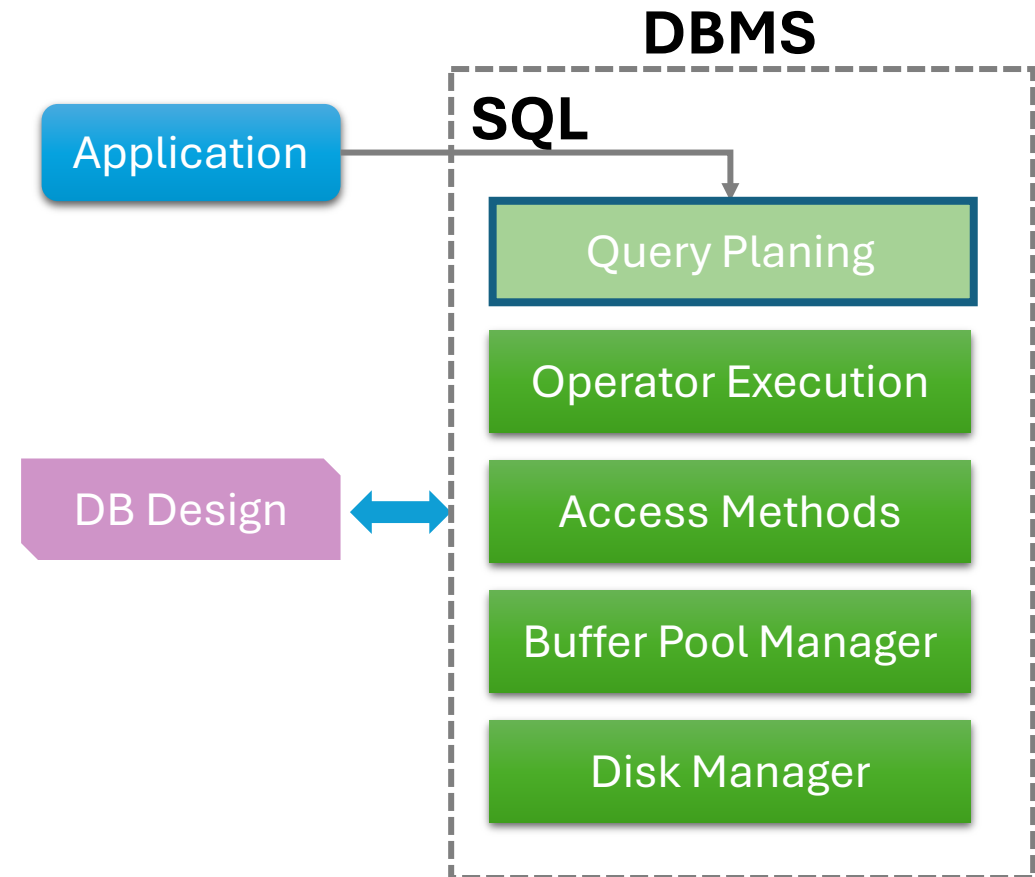
Chenhao Ma

School of Data Science

The Chinese University of Hong Kong, Shenzhen

# This Lecture

- Query Planing & Optimization








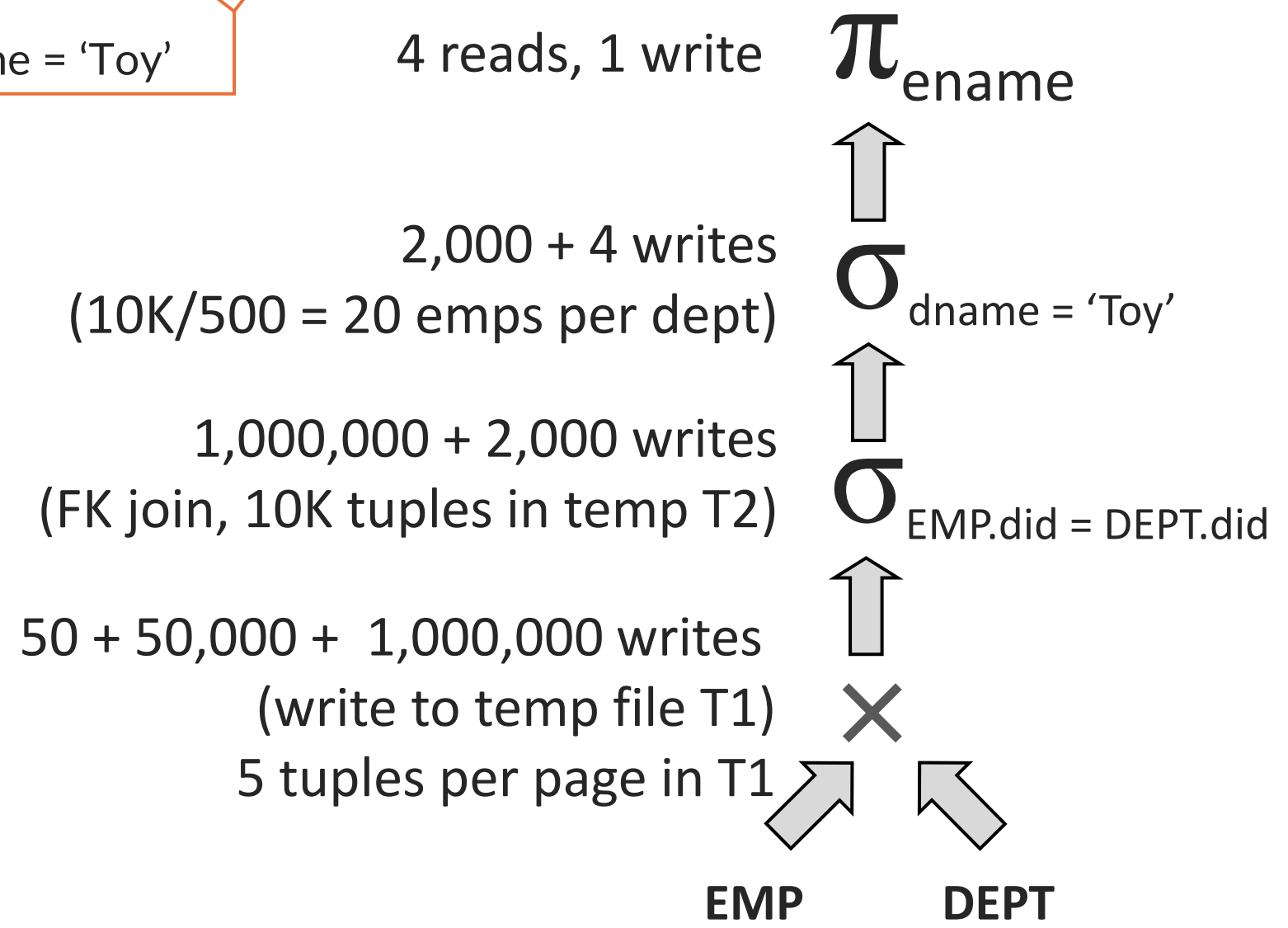


**Query**

```
SELECT distinct ename  
FROM Emp E, Dept D  
WHERE E.did = D.did AND D.dname = 'Toy'
```

**Total: 2M I/Os**

Catalog	
<div><div>clustered</div></div> <div><div>unclustered</div></div> <div><div>unclustered</div></div>	
EMP (ssn, ename, addr, sal, did)	
10,000 records	
1,000 pages	
<hr/>	
<div><div>clustered</div></div> <div><div>unclustered</div></div>	
DEPT (did, dname, floor, mgr)	
500 records	
50 pages	





**Query**

```
SELECT distinct ename  
FROM Emp E, Dept D  
WHERE E.did = D.did AND D.dname = 'Toy'
```

**Total: 54K I/Os**

## Catalog

clustered      unclustered      unclustered

▲      ▲      ▲

EMP (ssn, ename, addr, sal, did)

10,000 records  
1,000 pages

clustered      unclustered

▲      ▲

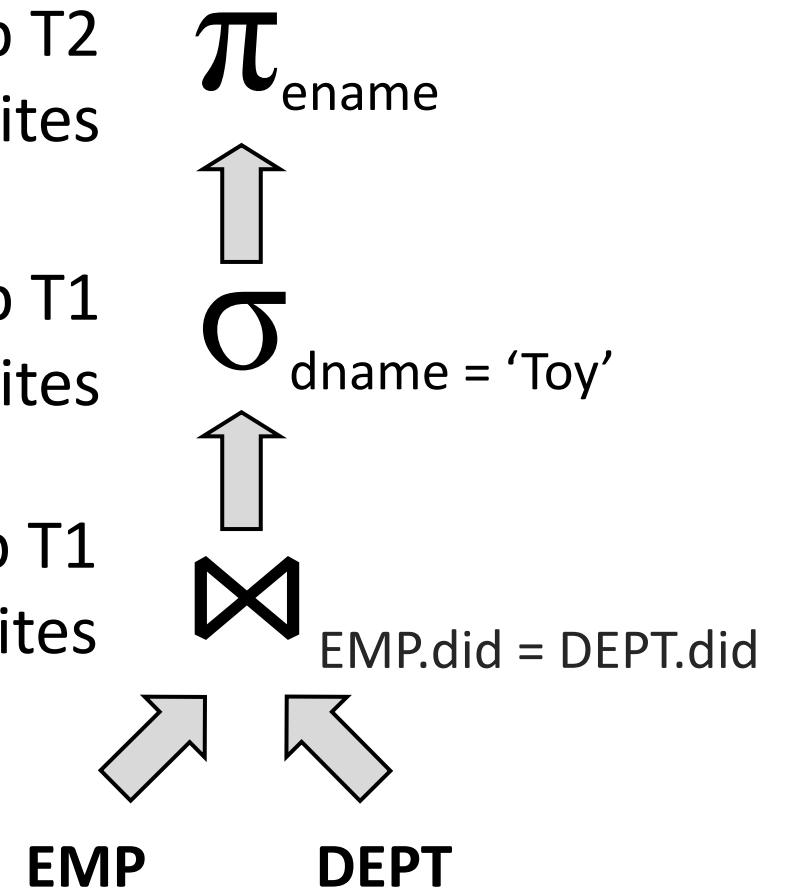
DEPT (did, dname, floor, mgr)

500 records  
50 pages

Read temp T2  
4 reads + 1 writes

Read temp T1  
2,000 reads + 4 writes

Page NL, write to temp T1  
50 + 50,000 + 2000 writes








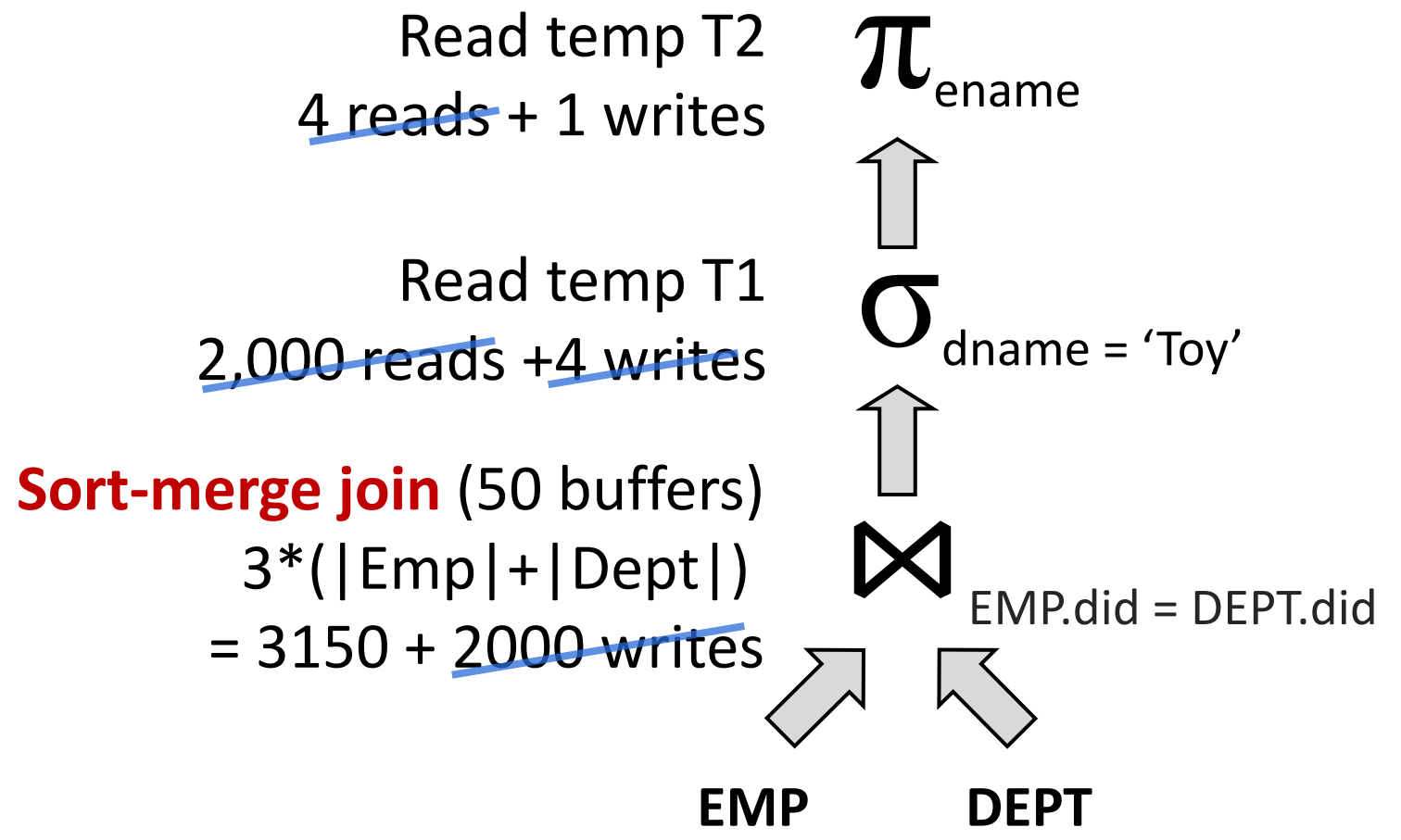
Query

SELECT distinct ename  
FROM Emp E, Dept D  
WHERE E.did = D.did AND D.dname = 'Toy'

w/ Materialization **Total: 7,159 I/Os**

w/ Pipelining **Total: 3,151 I/Os**

Catalog	
<div> <div>clustered</div> <div>unclustered</div> <div>unclustered</div> </div> <div>    </div> <div>EMP (ssn, ename, addr, sal, did)</div> <div>10,000 records</div> <div>1,000 pages</div>	
<div> <div>clustered</div> <div>unclustered</div> </div> <div>   </div> <div>DEPT (did, dname, floor, mgr)</div> <div>500 records</div> <div>50 pages</div>	





Query

```
SELECT distinct ename  
FROM Emp E, Dept D  
WHERE E.did = D.did AND D.dname = 'Toy'
```

Total: 37 I/Os

## Catalog

clustered      unclustered      unclustered

▲      ▲      ▲

EMP (ssn, ename, addr, sal, did)  
10,000 records  
1,000 pages

clustered      unclustered

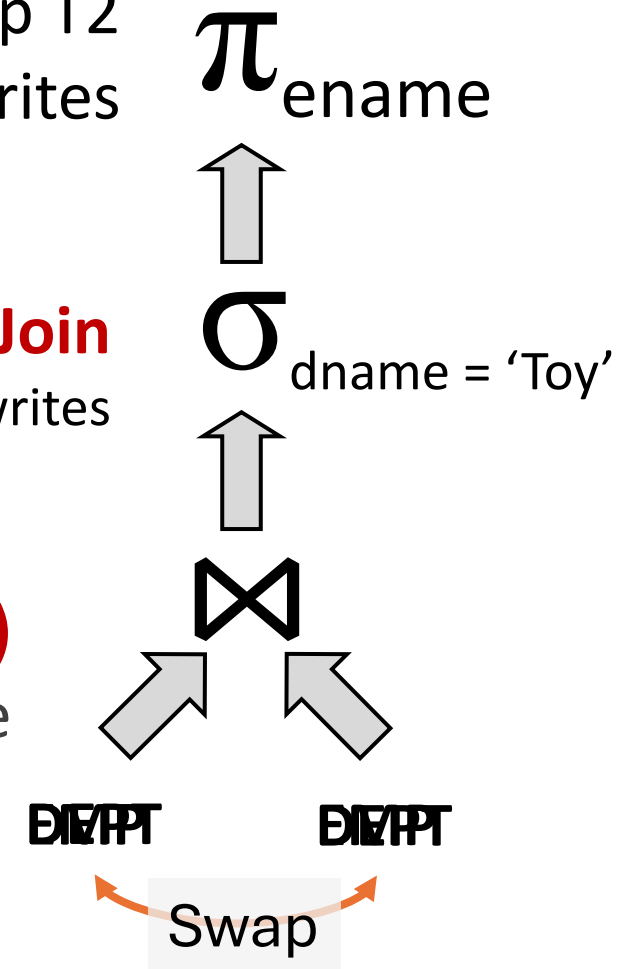
▲      ▲

DEPT (did, dname, floor, mgr)  
500 records  
50 pages

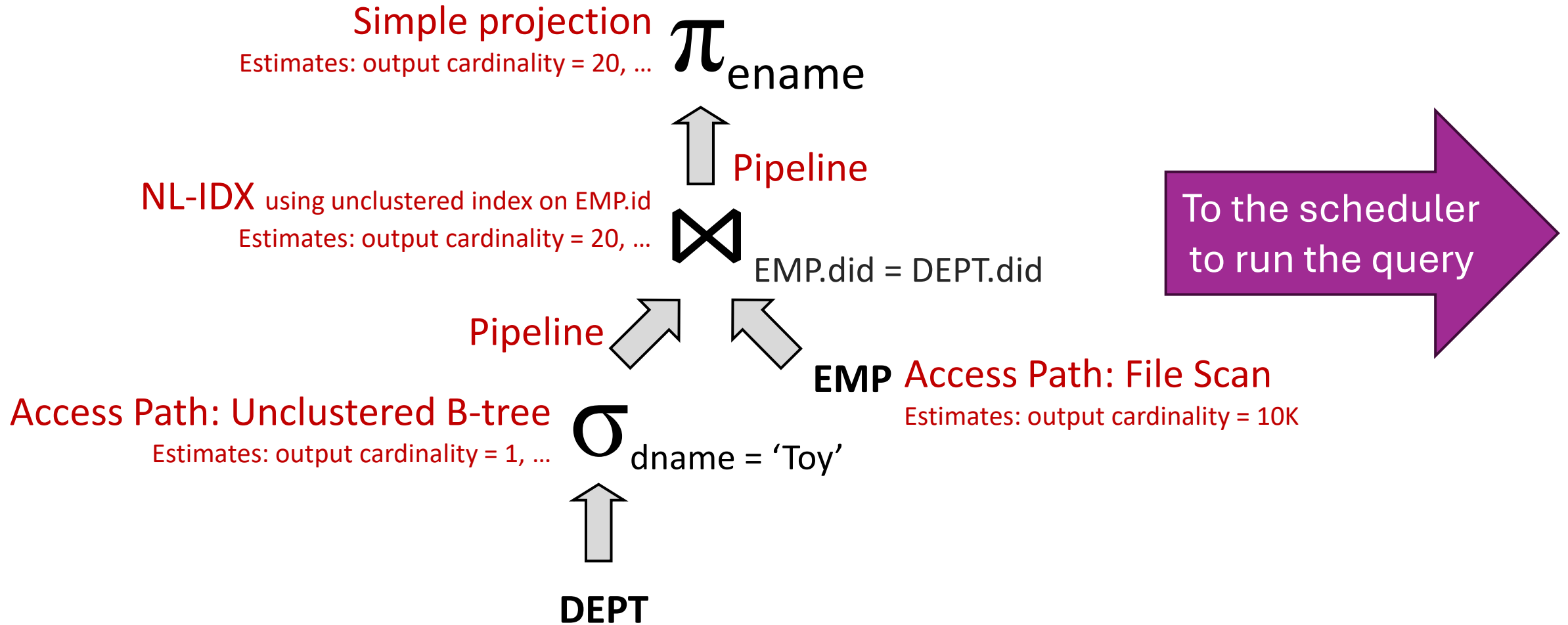
Read temp T2  
4 reads + 1 writes

Read temp T1, **NL-IDX Join**  
1 + 3 (idx) + 20 (ptr chase) + 4 writes

Access: **Index (name)**  
3 reads + 1 write

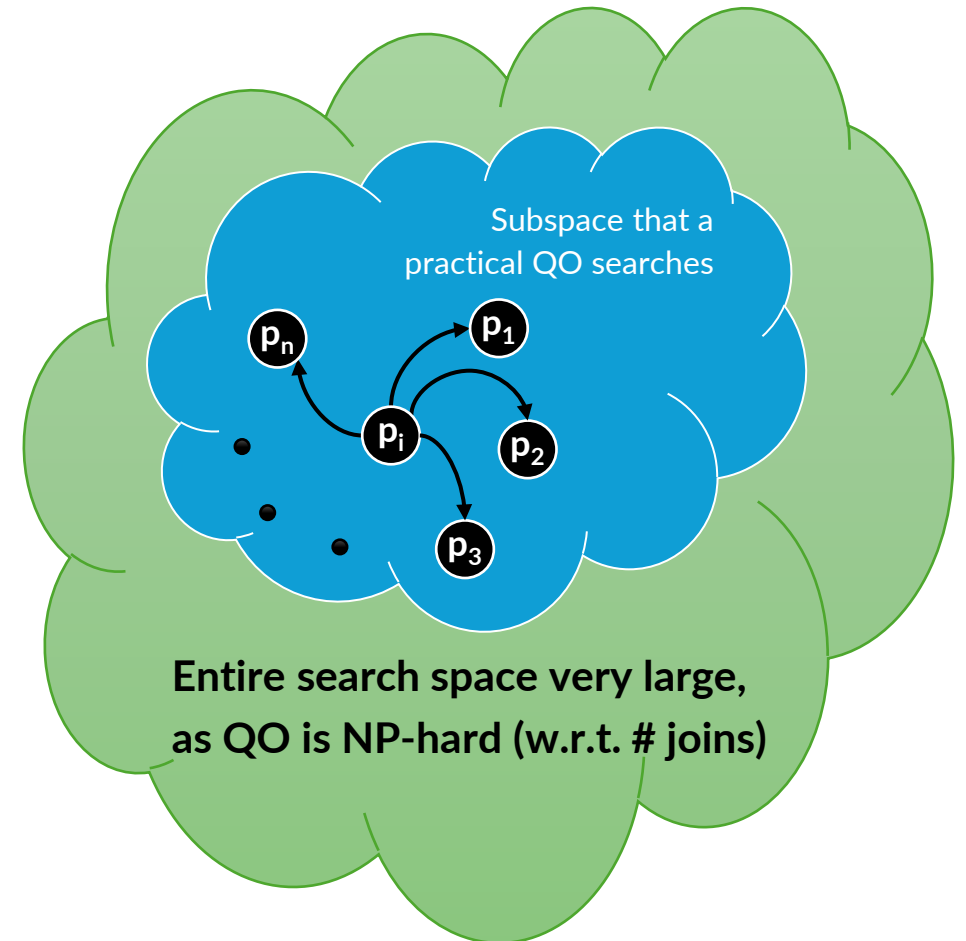


# Annotated RA Tree a.k.a. The Physical Plan



# Query Optimization (QO)

1. Identify candidate equivalent trees (logical). It is an NP-hard problem, so the space is large.
  2. For each candidate, find the execution plan tree (physical). We need to **estimate** the cost of each plan.
  3. Choose the best overall (physical) plan.
- **Practically: Choose from a subset of all possible plans.**







# Logical VS. Physical Plans

- The optimizer generates a mapping of a **logical** algebra expression to the optimal equivalent physical algebra expression.
- **Physical** operators define a specific execution strategy using an access path.
  - They can depend on the physical format of the data that they process (i.e., sorting, compression).
  - Not always a 1:1 mapping from logical to physical.



# Query Optimization

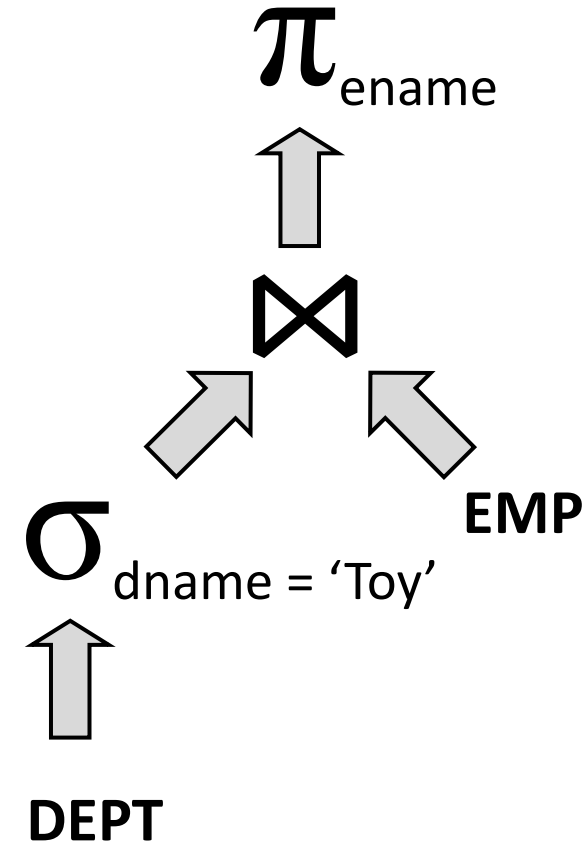
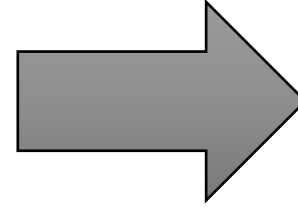
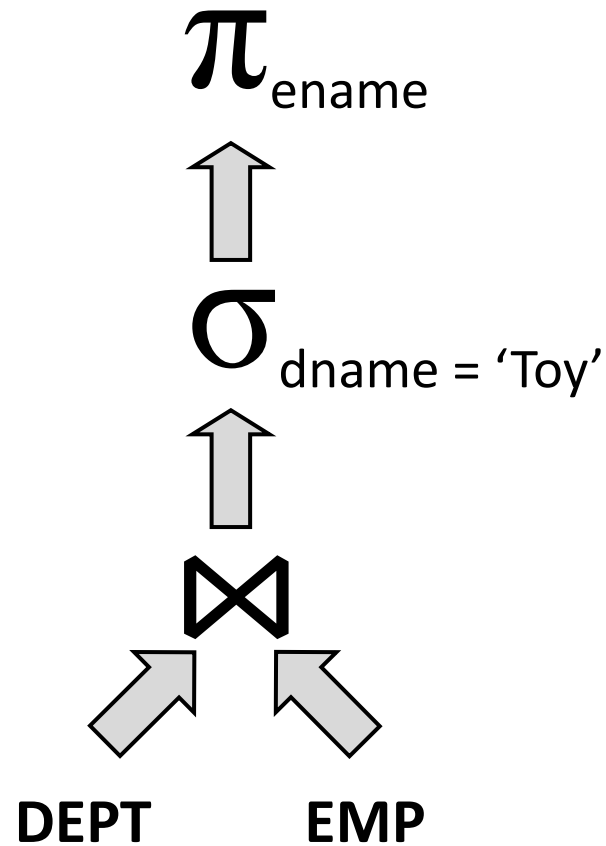
- **Heuristics / Rules**

- Rewrite the query to remove (guessed) inefficiencies; e.g, always do selections first, or push down projections as early as possible.
- These techniques may need to examine catalog, but they do not need to examine data.

- **Cost-based Search**

- Use a model to estimate the cost of executing a plan.
- Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

# Predicate Pushdown

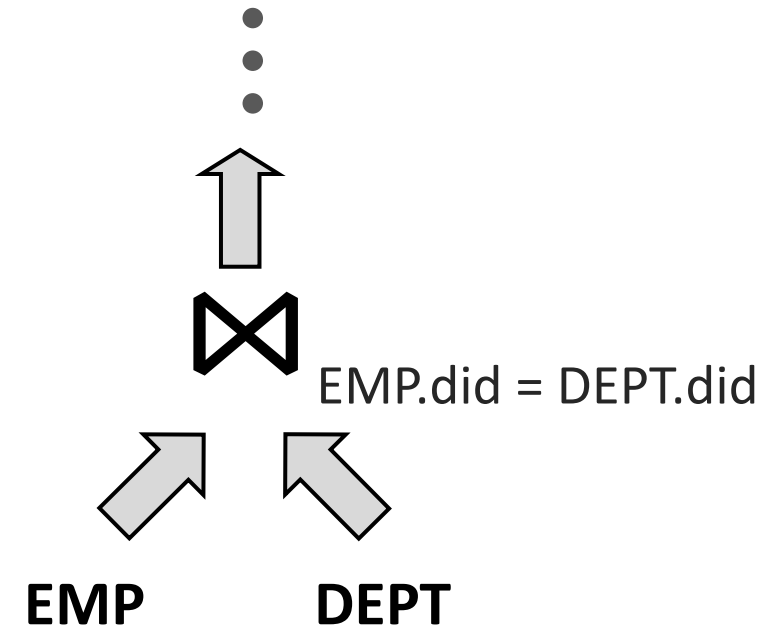
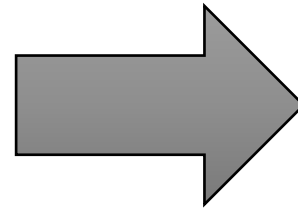
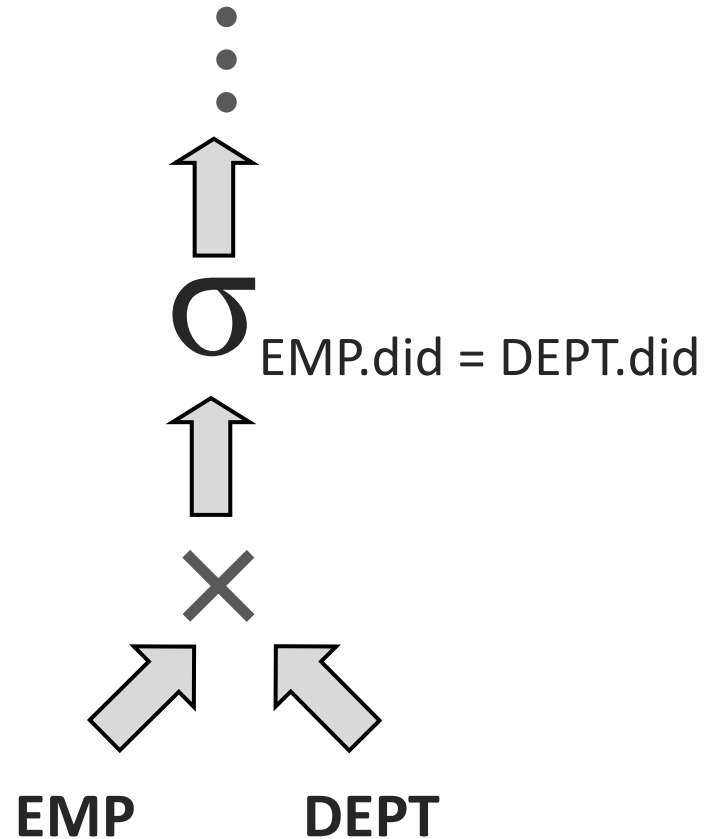


$\pi_{\text{ename}} (\sigma_{\text{dname} = \text{'Toy'}} (\text{DEPT} \bowtie \text{EMP}))$

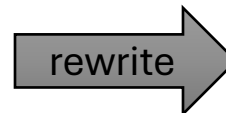
rewrite

$\pi_{\text{ename}} (\text{EMP} \bowtie \sigma_{\text{dname} = \text{'Toy'}} (\text{DEPT}))$

# Replace Cartesian Product

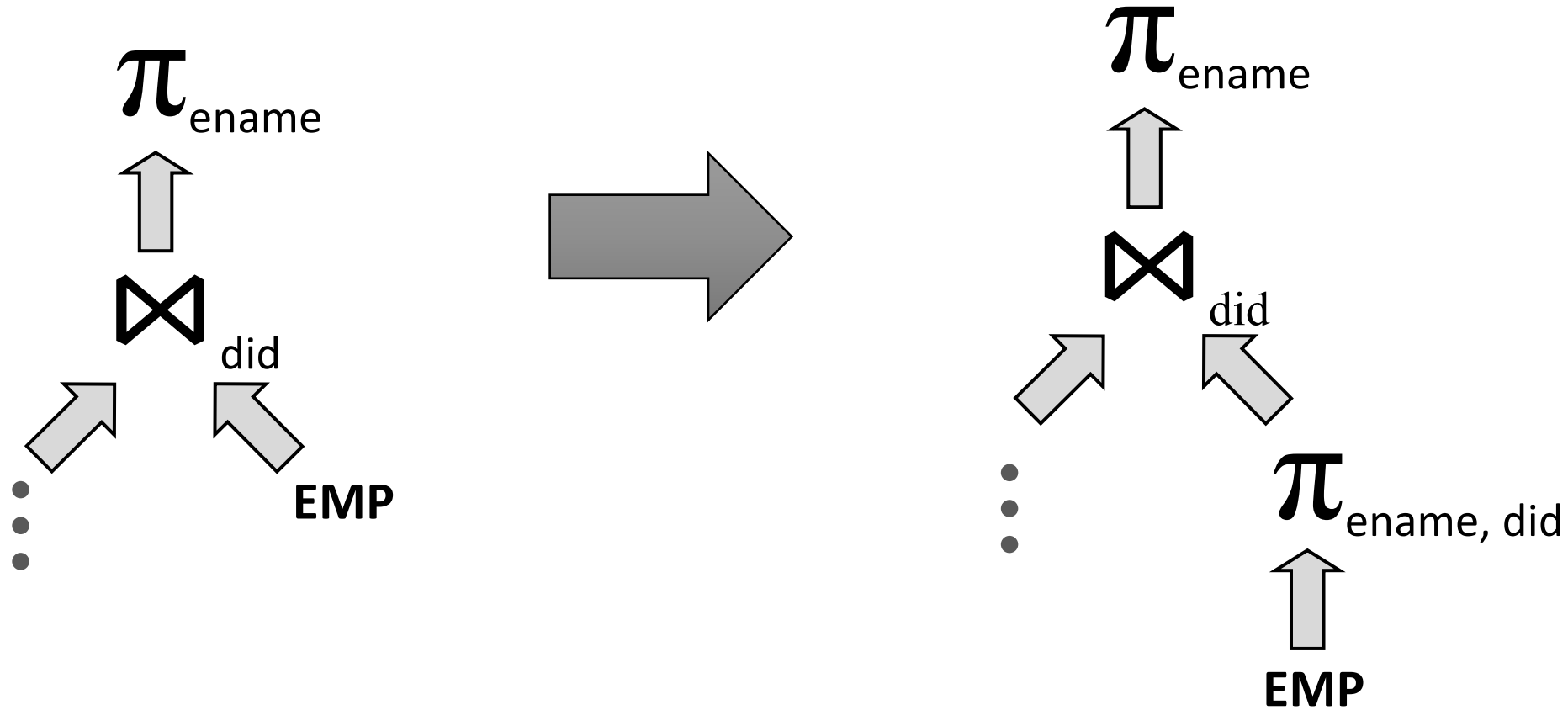


$\dots (\sigma_{DEPT.did = EMP.did} (DEPT \times EMP))$



$\dots (EMP \bowtie_{did} DEPT)$

# Projection Pushdown



$$\pi_{\text{EMP.ename}} \left( \dots \bowtie_{\text{did}} \text{EMP} \right)$$

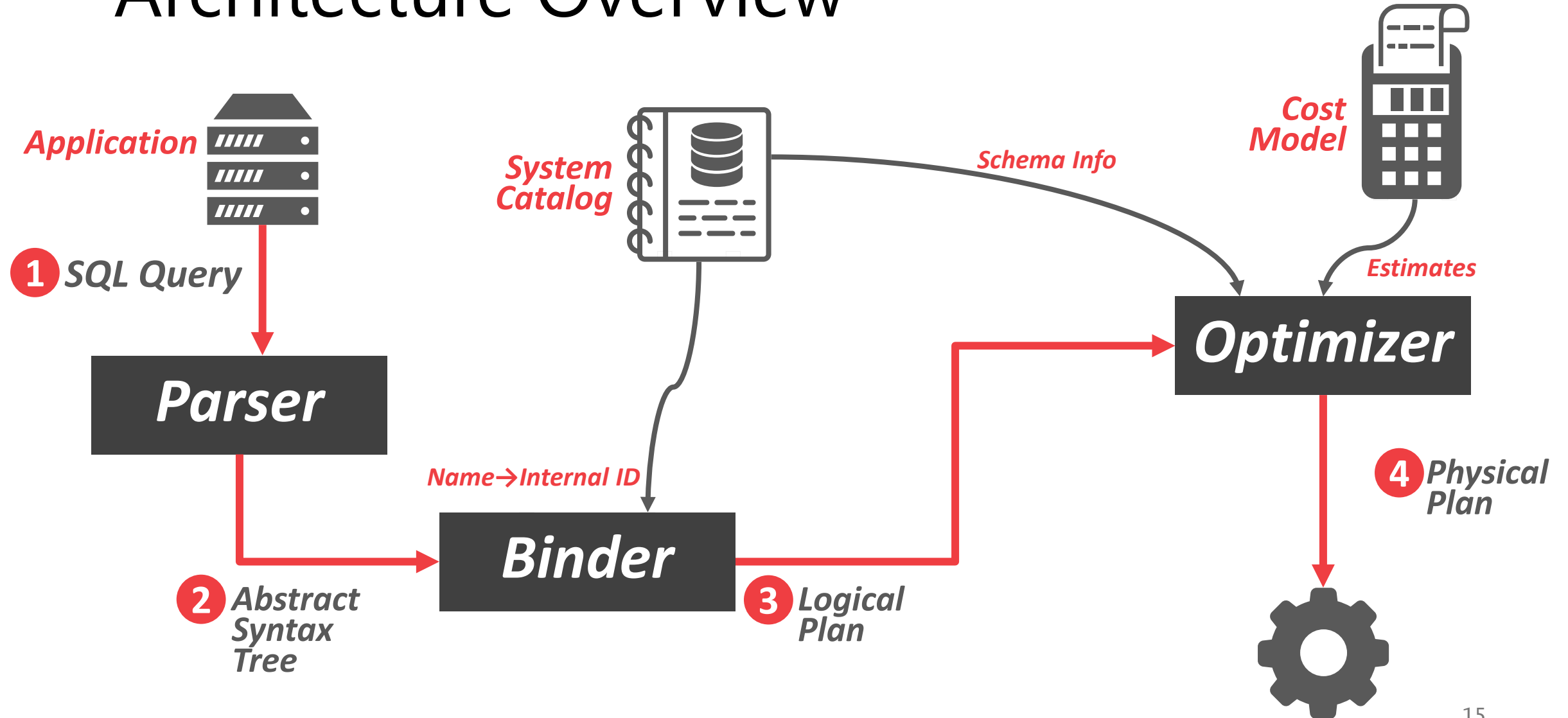
rewrite

$$\pi_{\text{EMP.ename}} \left( \dots \bowtie_{\text{did}} \left( \pi_{\text{ename, did}} \text{EMP} \right) \right)$$

# Equivalence

- $\sigma_{P_1}(\sigma_{P_2}(R)) \equiv \sigma_{P_2}(\sigma_{P_1}(R))$  ( $\sigma$  commutativity)
- $\sigma_{P_1 \wedge P_2 \dots \wedge P_n}(R) \equiv \sigma_{P_1}(\sigma_{P_2}(\dots \sigma_{P_n}(R)))$  (cascading  $\sigma$ )
- $\Pi_{A_1}(R) \equiv \Pi_{A_1}(\Pi_{A_2}(\dots \Pi_{A_k}(R)\dots))$ ,  $A_i \subseteq A_{i+1}$  (cascading  $\Pi$ )
- $R \bowtie S \equiv S \bowtie R$  (join commutativity)
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$  (join associativity)
- $\sigma_P(R \bowtie S) \equiv (R \bowtie_P S)$ , if  $P$  is a join predicate
- $\sigma_P(R \bowtie S) \equiv \sigma_{P_1}(\sigma_{P_2}(R) \bowtie_{P_4} \sigma_{P_3}(S))$ , where  $P = p_1 \wedge p_2 \wedge p_3 \wedge p_4$
- $\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(R)) \equiv \Pi_{A_1, A_2, \dots, A_n}(\sigma_P(\Pi_{A_1, \dots, A_n, B_1, \dots, B_M} R))$ , where  $B_1 \dots B_M$  are columns in  $P$
- ...

# Architecture Overview



# Query Optimization

- **Heuristics / Rules**

Examples: predicate pushdown, replace cartesian product, projection pushdown ...

- Rewrite the query to remove (guessed) inefficiencies; e.g, always do selections first, or push down projections as early as possible.
- These techniques may need to examine catalog, but they do not need to examine data.

- **Cost-based Search**

- Use a model to estimate the cost of executing a plan.
- Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.



# Cost-based Search



# Cost-based Query Optimization

- Let's start with a certain style of QO: cost-based, bottom-up QO (the classic System-R optimizer approach)
- Approach: Enumerate different plans for the query and estimate their costs.
  - Single relation.
  - Multiple relations.
  - Nested sub-queries.
- It chooses the best plan it has seen for the query after exhausting all plans or some timeout.

# Single-relation Query Planing

- Pick the best access method.
  - Sequential Scan
  - Binary Search (clustered indexes)
  - Index Scan
- Predicate evaluation ordering.
- Simple heuristics are often good enough for this.

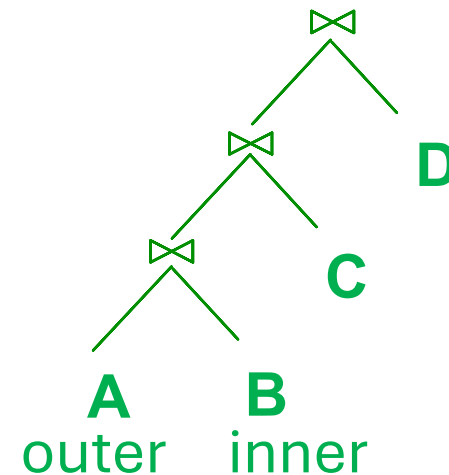
# System R Optimizer

- Break the query into blocks and generate the logical operators for each block.
- For each logical operator, generate a set of physical operators that implement it.
  - All combinations of join algorithms and access paths
- Then, iteratively construct a “left-deep” join tree that minimizes the estimated amount of work to execute the plan.

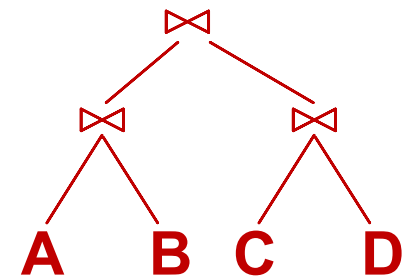


Selinger

A left-deep tree



A bushy tree



System-R optimizer does  
NOT consider this “shape”

# System R Optimizer

```
SELECT ARTIST.NAME
FROM ARTIST, APPEARS, ALBUM
WHERE ARTIST.ID=APPEARS.ARTIST_ID
AND APPEARS.ALBUM_ID=ALBUM.ID
AND ALBUM.GENRE="Blues"
ORDER BY ARTIST.ID
```

ARTIST: Sequential Scan

APPEARS: Sequential Scan

ALBUM: Index Look-up on GENRE

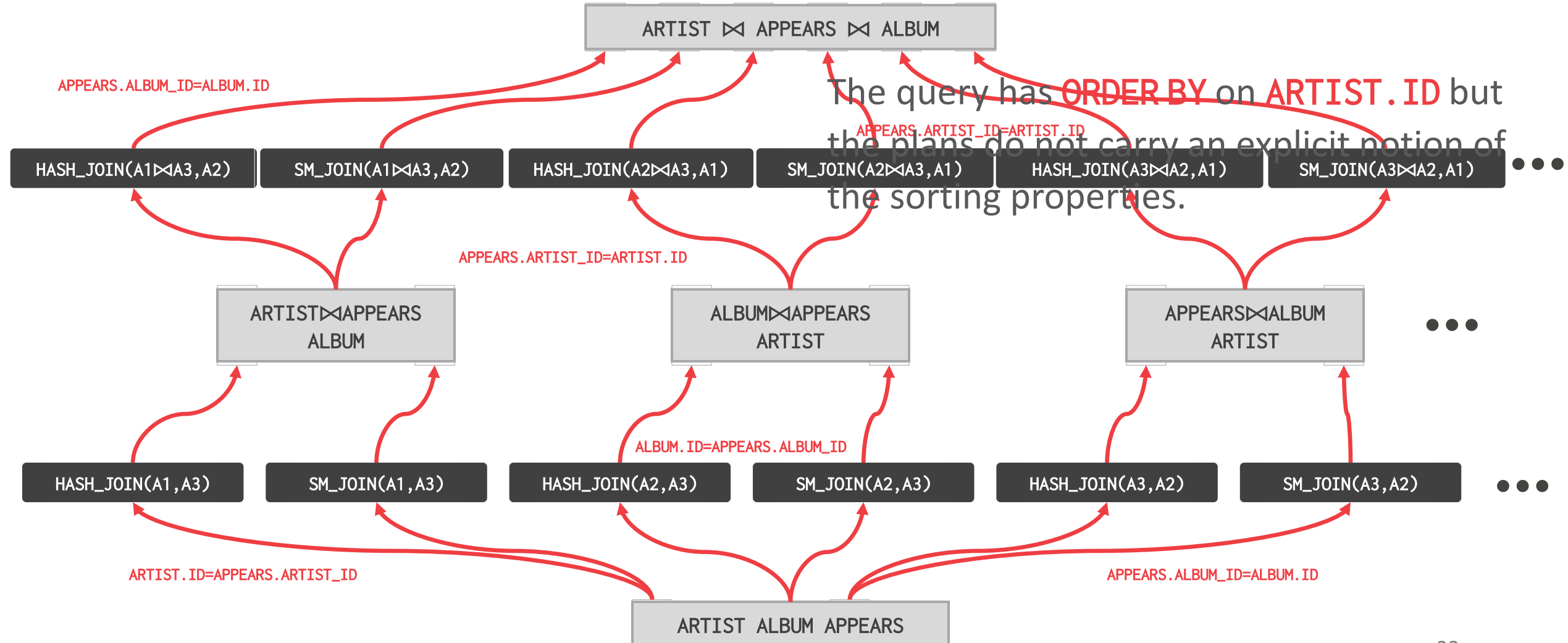
**Step #1:** Choose the best access paths to each table

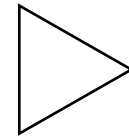
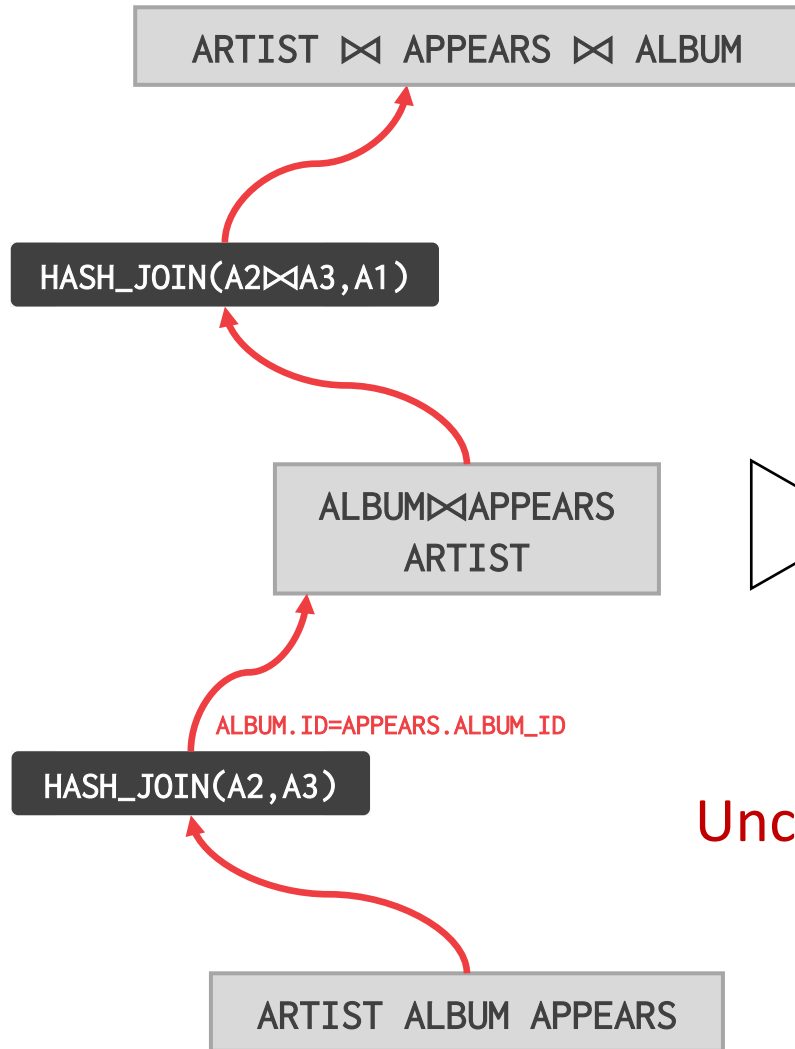
**Step #2:** Enumerate all possible join orderings for tables

**Step #3:** Determine the join ordering with the lowest cost

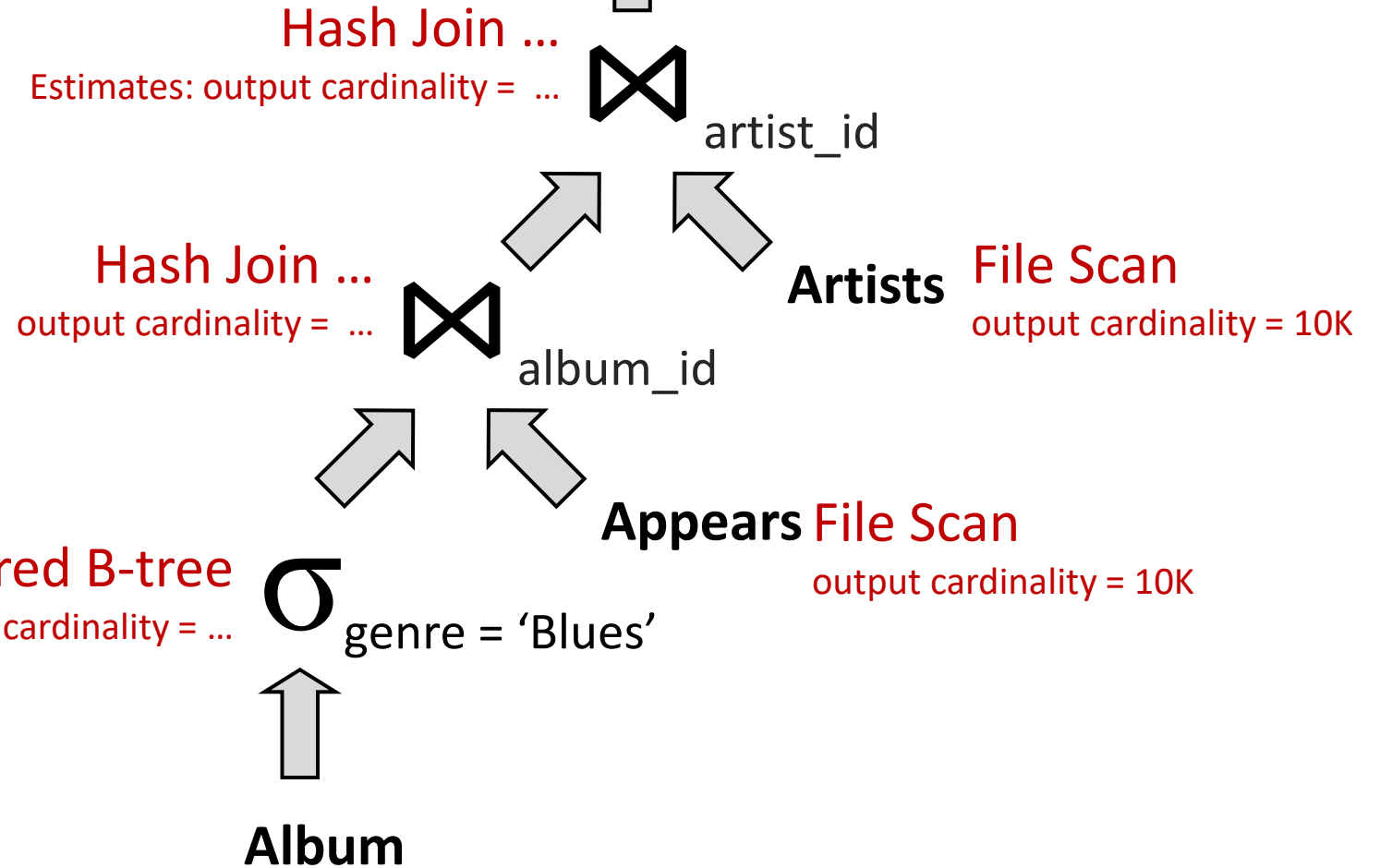
ARTIST	⋈	APPEARS	⋈	ALBUM
APPEARS	⋈	ALBUM	⋈	ARTIST
ALBUM	⋈	APPEARS	⋈	ARTIST
APPEARS	⋈	ARTIST	⋈	ALBUM
ARTIST	×	ALBUM	⋈	APPEARS
ALBUM	×	ARTIST	⋈	APPEARS
⋮		⋮		⋮

# System R Optimizer





Unclustered B-tree  
output cardinality = ...





# Multi-Relation Query Planning

- **Choice #1: Bottom-up Optimization**

- Start with nothing and then build up the plan to get to the outcome that you want.

We just saw an example of this, the System R approach

- **Choice #2: Top-down Optimization**

- Start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.





# Bottom-up Optimization

- Use static rules to perform initial optimization. Then use dynamic programming to determine the best join order for tables using a divide-and-conquer search method
- **Examples:** IBM System R, DB2, MySQL, Postgres, most open-source DBMSs.



# Top-down Optimization

- Start with a logical plan of what we want the query to be.  
Perform a branch-and-bound search to traverse the plan tree by converting logical operators into physical operators.
- Keep track of global best plan during search.
- Treat physical properties of data as first-class entities during planning.
- **Example:** MSSQL, Greenplum, CockroachDB

# Top-down Optimization

Invoke rules to create new nodes and traverse the tree.

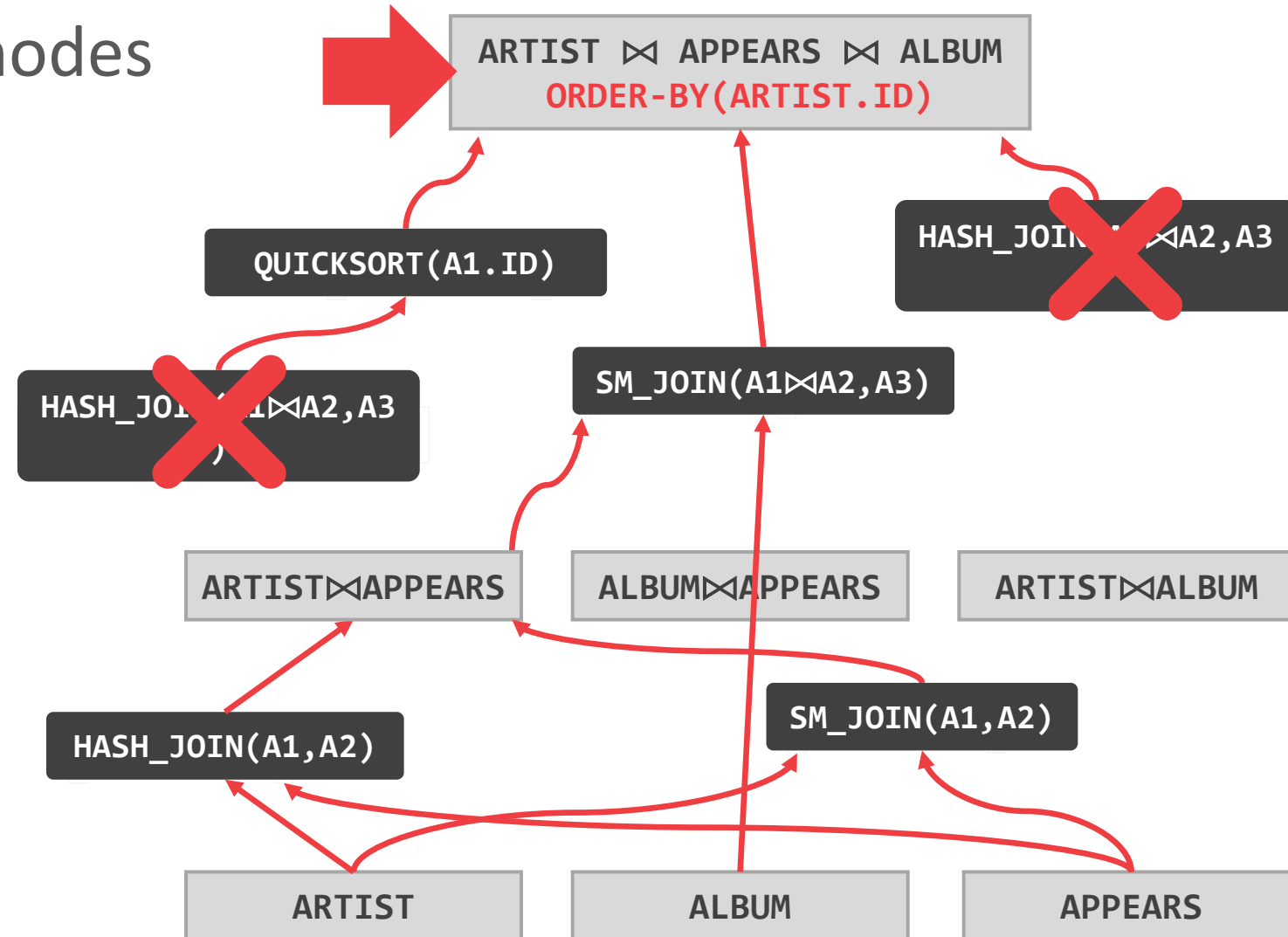
→ **Logical** → **Logical**:

JOIN(A, B) to JOIN(B, A)

→ **Logical** → **Physical**:

JOIN(A, B) to HASH\_JOIN(A, B)

Can create “enforcer” rules that require input to have certain properties.



# Life so far ... single block QO

- Often, we get nested queries.
  - We could optimize each block using the methods we have discussed.
  - However, this may be inefficient since we optimize each block separately without a global approach.
- What if we could flatten a nested query into a single block and optimize it?
  - Then, apply single-block query optimization methods.
  - Even if one can't flatten to a single block, flattening to fewer blocks is still beneficial.

# Nested Queries

# Nested Queries

- The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.
- Two Approaches:
  - Rewrite to de-correlate and/or flatten them.
  - Decompose nested query and store results in a temporary table.

# Nested Sub-queries: Rewrite

```
SELECT name FROM sailors AS S
WHERE EXISTS (
  SELECT * FROM reserves AS R
  WHERE S.sid = R.sid
  AND R.day = '2022-10-25'
)
```



```
SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2022-10-25'
```

# Decomposing Queries

- For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.
- Sub-queries are written to temporary tables that are discarded after the query finishes.



# Decomposing Queries

**Inner Block** `SELECT MAX(rating) FROM sailors`

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

**Outer Block**

**Nested Block**

# Expression Rewriting

# Expression Rewriting

- An optimizer transforms a query's expressions (e.g., **WHERE/ON** clause predicates) into the minimal set of expressions.
- Implemented using if/then/else clauses or a pattern-matching rule engine.
  - Search for expressions that match a pattern.
  - When a match is found, rewrite the expression.
  - Halt if there are no more rules that match.

# Expression Rewriting

- Impossible / Unnecessary Predicates

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE false;
```

```
SELECT * FROM A WHERE RANDOM() IS NULL;
```

- Merging Predicates

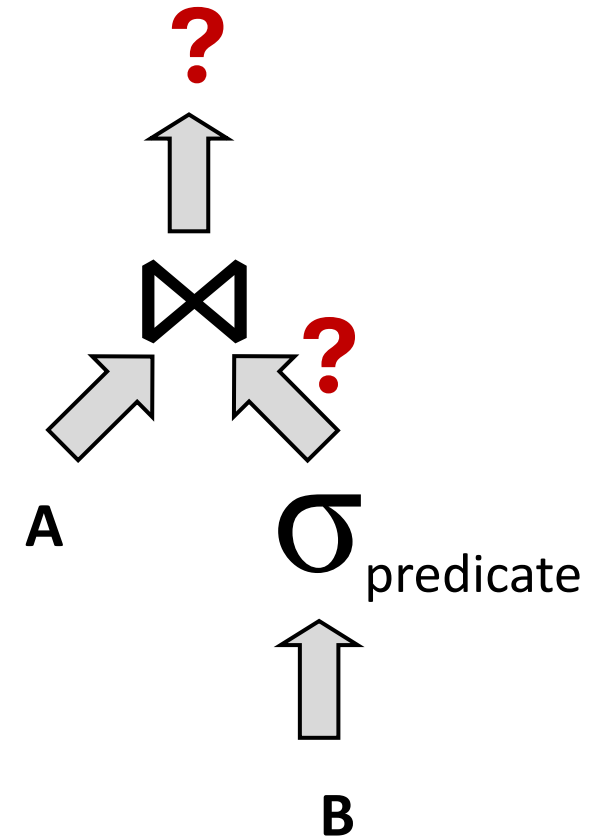
```
SELECT * FROM A
```

```
WHERE val BETWEEN 1 AND 150;  
OR val BETWEEN 50 AND 150;
```

# Cost Estimation

# How do we calculate the cost of the plans?

- We have formulas for the operator algorithms (e.g. the cost formulae for hash join, sort merge join, ...), but we also need to estimate the size of the output that an operator produces.



# Cost Estimation

- The DBMS uses a cost model to predict the behavior of a query plan given a database state.
  - This is an internal cost that allows the DBMS to compare one plan with another.
- It is too expensive to run every possible plan to determine this information, so the DBMS needs a way to derive this information.

# Cost Model Components

- **Choice #1: Physical Costs**

- Predict CPU cycles, I/O, cache misses, RAM consumption, network messages...
- Depends heavily on hardware.

- **Choice #2: Logical Costs**

- Estimate output size per operator.
- Independent of the operator algorithm.
- Need estimations for operator result sizes.



# Postgres Cost Model

- Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.
- Default settings are obviously for a disk-resident database without a lot of memory:
  - Processing a tuple in memory is **400x** faster than reading a tuple from disk.
  - Sequential I/O is **4x** faster than random I/O.

## 19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to 1.0 and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

**Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (floating point)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see [ALTER TABLESPACE](#)).

`random_page_cost` (floating point)



# Statistics

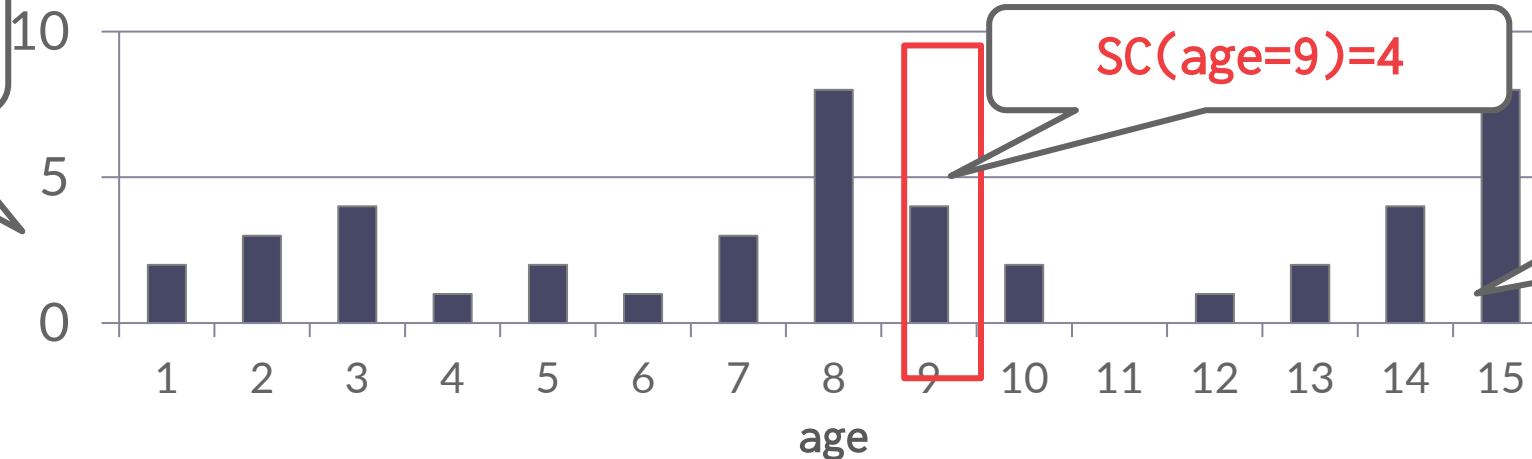
- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update them at different times.
- Manual invocations:
  - Postgres/SQLite: **ANALYZE**
  - Oracle/MySQL: **ANALYZE TABLE**
  - SQL Server: **UPDATE STATISTICS**
  - DB2: **RUNSTATS**

# Selection Cardinality

- The selectivity (**sel**) of a predicate **P** is the fraction of tuples that qualify.
- Equality Predicate:  $A = \text{constant}$** 
  - $\text{sel}(A = \text{constant}) = \frac{\text{\#occurrences}}{|R|}$
  - Example:  $\text{sel}(\text{age} = 9) = \frac{4}{45}$

```
SELECT * FROM people
WHERE age = 9
```

**# of  
occurrences**



$SC(\text{age} = 9) = 4$

**Distinct values  
of attribute**



# Selection Cardinality

- **Assumption #1: Uniform Data**
  - The distribution of values (except for the heavy hitters) is the same.
- **Assumption #2: Independent Predicates**
  - The predicates on attributes are independent
- **Assumption #3: Inclusion Principle**
  - The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.



# Correlated Attributes

- Consider a database of automobiles:
  - # of Makes = 10, # of Models = 100
- And the following query:
  - `(make="Honda" AND model="Accord")`
- With the independence and uniformity assumptions, the selectivity is:
  - $1/10 \times 1/100 = 0.001$
- But since only Honda makes Accords the real selectivity is  $1/100 = 0.01$

Source: [Guy Lohman](#)

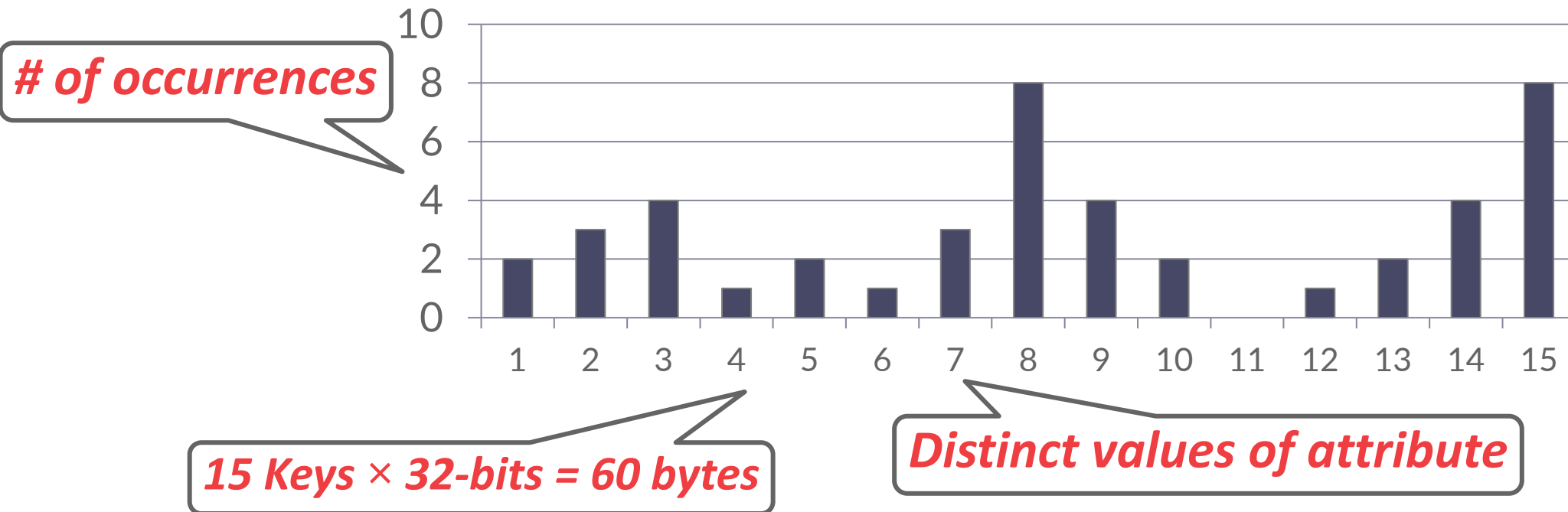
# Statistics

- **Choice #1: Histograms**
  - Maintain an occurrence count per value (or range of values) in a column.
- **Choice #2: Sketches**
  - Probabilistic data structure that gives an approximate count for a given value.
- **Choice #3: Sampling**
  - DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.

# Histograms

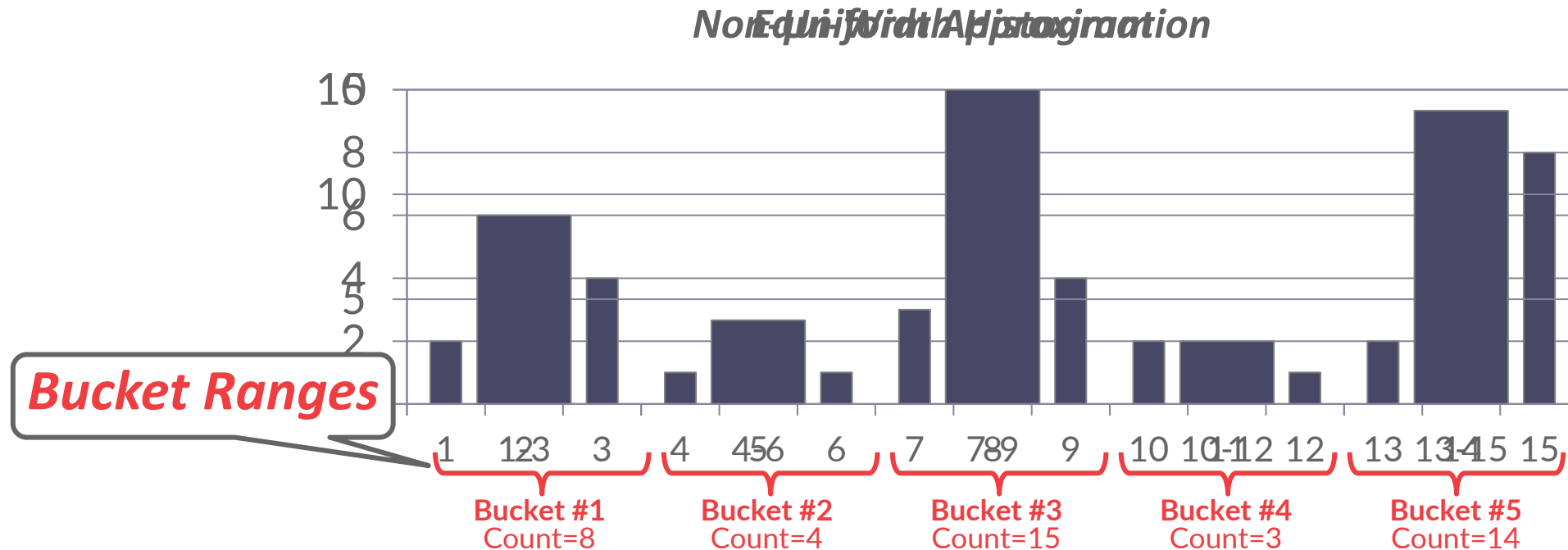
- Our formulas are nice, but we assume that data values are uniformly distributed.

*Histogram*



# Equi-width Histogram

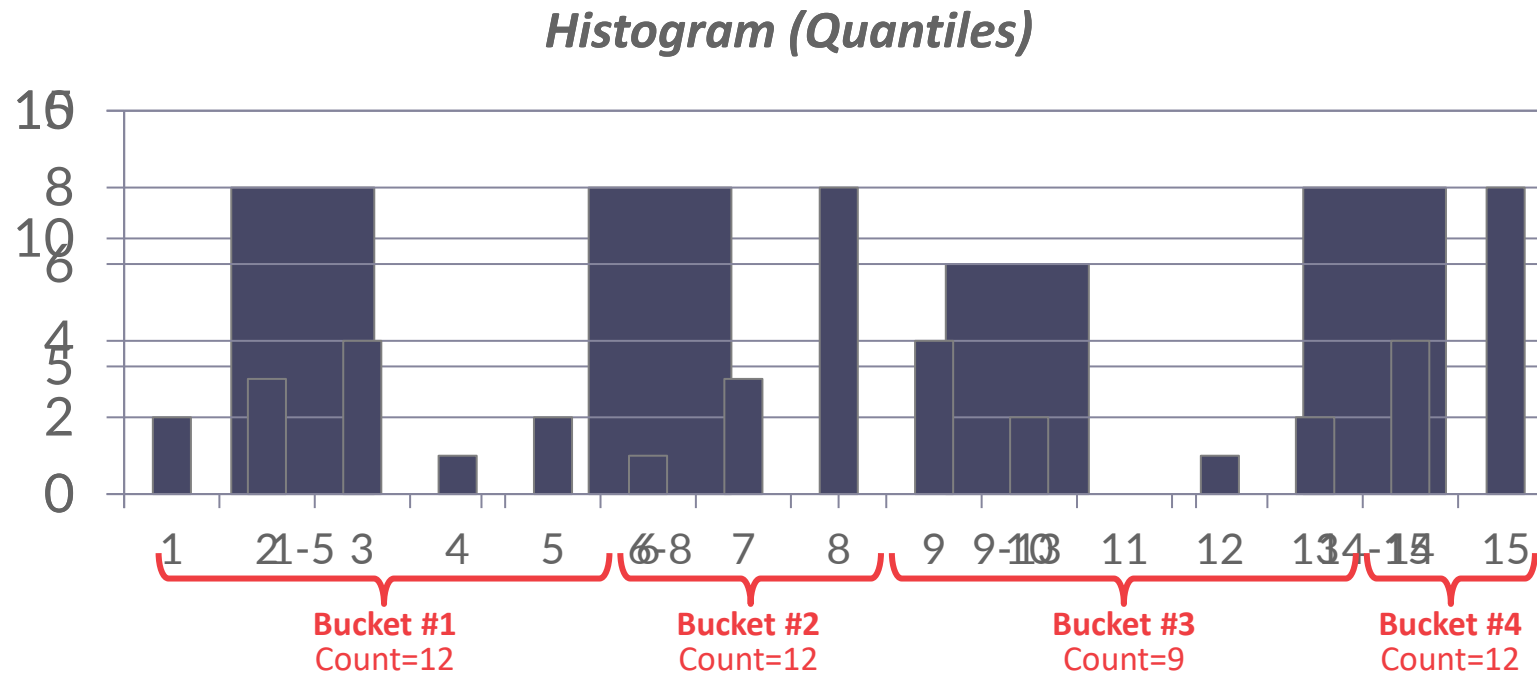
- Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).





# Equi-depth Histogram

- Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



# Sketches

- Probabilistic data structures that generate approximate statistics about a data set.
- Cost-model can replace histograms with sketches to improve its selectivity estimate accuracy.
- Most common examples:
  - [Count-Min Sketch](#) (1988): Approximate frequency count of elements in a set.
  - [HyperLogLog](#) (2007): Approximate the number of distinct elements in a set.

# Sampling

- Modern DBMSs also collect samples from tables to estimate selectivities.
- Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
FROM people
WHERE age > 50
```

**Table Sample**

1001	Obama	61	Rested
1003	Tupac	25	Dead
1005	Andy	41	Illin

$\text{sel}(\text{age} > 50) = 1/3$



id	name	age	status
1001	Obama	61	Rested
1002	Kanye	45	Weird
1003	Tupac	25	Dead
1004	Bieber	28	Crunk
1005	Andy	41	Illin
1006	TigerKing	59	Jailed

*1 billion tuples*

# Conclusion

- Query optimization is critical for a database system.
- SQL -> logical plan -> physical plan.
- Flatten queries before going to the optimization part. Expression handling is also important.
- QO enumeration can be bottom-up or top-down.
- Need to cost each plan, so need cost-estimation methods.

# Essential Query Optimization papers

## An Overview of Query Optimization in Relational Systems

Surajit Chaudhuri  
Microsoft Research  
One Microsoft Way  
Redmond, WA 98052  
+1-(425)-703-1938  
surajitc@microsoft.com

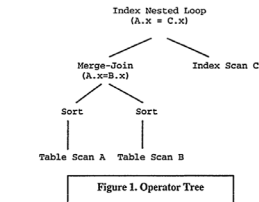
### 1. OBJECTIVE

There has been extensive work in query optimization since the early 70s. It is hard to capture the breadth and depth of this large body of work in a short article. Therefore, I have decided to focus primarily on the optimization of SQL queries in relational database systems and present my biased and incomplete view of this field. The goal of this article is not to be comprehensive, but rather to explain the foundations and present samplings of significant work in this area. I would like to apologize to the many contributors in this area whose work I have failed to explicitly acknowledge due to oversight or lack of space. I take the liberty of trading technical precision for ease of presentation.

### 2. INTRODUCTION

Relational query languages provide a high-level "declarative" interface to access data stored in relational databases. Over time, SQL [41] has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the *query optimizer* and the *query execution engine*.

The query execution engine implements a set of *physical operators*. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index scan, nested-loop join, and sort-merge join. I refer to such operators as physical operators since they are not necessarily tied one-to-one with relational operators. The simplest way to think of physical operators is as pieces of code that are used as building blocks to make possible the execution of SQL queries. An abstract representation of such an execution is a *physical operator tree*, as illustrated in Figure 1. The edges in an operator tree represent the data flow among the physical operators. We use the terms *physical operator tree* and *execution plan* (or, simply *plan*) interchangeably. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible. We defer the reader to [20] for an overview of query evaluation techniques.



The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an *efficient* execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is nontrivial since for a given SQL query, there can be a large number of possible operator trees:

- The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations: e.g.,  
 $Join(Join(A, B), C) = Join(Join(B, C), A)$
- For a given algebraic representation, there may be many operator trees that implement the algebraic expression, e.g., typically there are several join algorithms supported in a database system.

Furthermore, the throughput or the response times for the execution of these plans may be widely different. Therefore, a judicious choice of an execution by the optimizer is of critical importance. Thus, query optimization can be viewed as a difficult search problem. In order to solve this problem, we need to provide:

- A space of plans (*search space*).
- A *cost estimation* technique so that a cost may be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan.
- An *enumeration algorithm* that can search through the execution space.

## The Volcano Optimizer Generator: Extensibility and Efficient Search

Goetz Graefe  
Portland State University  
graefe@cs.pdx.edu

William J. McKenna  
University of Colorado at Boulder  
bill@cs.colorado.edu

### Abstract

*Emerging database application domains demand not only new functionality but also high performance. To satisfy these two requirements, the Volcano project provides efficient, extensible tools for query and request processing, particularly for object-oriented and scientific database systems. One of these tools is a new optimizer generator. Data model, logical algebra, physical algebra, and optimization rules are translated by the optimizer generator into optimizer source code. Compared with our earlier EX-ODUS optimizer generator prototype, the search engine is more extensible and powerful; it provides effective support for non-trivial cost models and for physical properties such as sort order. At the same time, it is much more efficient as it combines dynamic programming, which until now had been used only for relational select-project-join optimization, with goal-directed search and branch-and-bound pruning. Compared with other rule-based optimizer systems, it provides complete data model independence and more natural extensibility.*

### 1. Introduction

While extensibility is an important goal and requirement for many current database research projects and system prototypes, performance must not be sacrificed for two reasons. First, data volumes stored in database systems continue to grow, in many application domains far beyond the capabilities of most existing database systems. Second, in order to overcome acceptance problems in emerging database application areas such as scientific computation, database systems must achieve at least the same performance as the file systems currently in use. Additional software layers for database management must be counterbalanced by database performance advantages normally not used in these application areas. Optimization and parallelization are prime candidates to provide these performance advantages, and tools and techniques for optimization and parallelization are crucial for the wider use of extensible database technology.

For a number of research projects, namely the Volcano extensible, parallel query processor [4], the REVELATION OODBMS project [11] and optimization and parallelization in scientific databases [20] as well as to assist research efforts by other researchers, we have built a new extensible query optimization system. Our earlier experience with the EXODUS optimizer generator had been inconclusive; while it had proven the feasibility and validity of the optimizer generator paradigm, it was difficult to construct efficient, production-quality optimizers. Therefore, we designed a new optimizer generator, requiring several important improvements over the EXODUS prototype.

First, this new optimizer generator had to be usable both in the Volcano project with the existing query execution software as well as in other projects as a stand-alone tool. Second, the new system had to be more efficient, both in optimization time and in memory consumption for the search. Third, it had to provide effective, efficient, and extensible support for physical properties such as sort order and compression status. Fourth, it had to permit use of heuristics and data model semantics to guide the search and to prune futile parts of the search space. Finally, it had to support flexible cost models that permit generating dynamic plans for incompletely specified queries.

In this paper, we describe the Volcano Optimizer Generator, which will soon fulfill all the requirements above. Section 2 introduces the main concepts of the Volcano optimizer generator and enumerates facilities for tailoring a new optimizer. Section 3 discusses the optimizer search strategy in detail. Functionality, extensibility, and search efficiency of the EXODUS and Volcano optimizer generators are compared in Section 4. In Section 5, we describe and compare other research into extensible query optimization. We offer our conclusions from this research in Section 6.

### 2. The Outside View of the Volcano Optimizer Generator

In this section, we describe the Volcano optimizer generator as seen by the person who is implementing a database system and its query optimizer. The focus is the wide array of facilities given to the optimizer implementor, i.e., modularity and extensibility of the Volcano optimizer generator design. After considering the design principles of the Volcano optimizer generator, we discuss generator input and operation. Section 3 discusses the search strategy used by optimizers generated with the Volcano optimizer generator.

Figure 1 shows the optimizer generator paradigm. When the DBMS software is being built, a model specification is translated into optimizer source code, which is then compiled and linked with the other DBMS

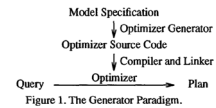


Figure 1. The Generator Paradigm.

## Access Path Selection in a Relational Database Management System

P. Griffiths Selinger  
M. M. Astrahan  
D. D. Chamberlin  
R. A. Lorie  
T. G. Price

IBM Research Division, San Jose, California 95193

**ABSTRACT:** In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

### 1. Introduction

System R is an experimental database management system based on the relational model of data which has been under development at the IBM San Jose Research Laboratory since 1975 <1>. The software was developed as a research vehicle in relational database, and is not generally available outside the IBM Research Division.

This paper assumes familiarity with relational data model terminology as described in Codd <2> and Date <3>. The user interface in System R is the unified query, data definition, and manipulation language SQL <4>. Statements in SQL can be issued both from an on-line casual-user-oriented terminal interface and from programming languages such as PL/I and COBOL.

In System R a user need not know how the tuples are physically stored and what access paths are available (e.g. which columns have indexes). SQL statements do not require the user to specify anything about the access path to be used for tuple retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an

Copyright © 1979 by the ACM, Inc. used by permission. Permission to make digital or hard copies is granted provided that copies are not made or distributed for profit or direct commercial advantage, and that copies show this notice on the first page or initial screen of a display along with the citation. Originally published in the *Proceedings of the 1979 ACM SIGMOD International Conference on the Management of Data*. Digitized by Eric A. Brewer, brewer@cs.berkeley.edu, October 2002.

access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in the processing of a SQL statement, and section 3 will describe the storage component access paths that are available on a single physically stored table. In section 4 the optimizer cost formulas are introduced for single table queries; and section 5 discusses the joining of two or more tables, and their corresponding costs. Nested queries (queries in predicates) are covered in section 6.

### 2. Processing of an SQL statement.

A SQL statement is subjected to four phases of processing. Depending on the origin and contents of the statement, these phases may be separated by arbitrary intervals of time. In System R these arbitrary time intervals are transparent to the system components which process a SQL statement. These mechanisms and a description of the processing of SQL statements from both programs and terminals are further discussed in <2>. Only an overview of those processing steps that are relevant to access path selection will be discussed here.

The four phases of statement processing are parsing, optimization, code generation, and execution. Each SQL statement is sent to the parser, where it is checked for correct syntax. A query block is represented by a SELECT list, a FROM list, and a WHERE clause, respectively the list of items to be retrieved, the table(s) referenced, and the boolean combination of simple predicates specified by the user. A single SQL statement may have many query blocks because a predicate may have one operand which is itself a query.

If the parser returns without any errors detected, the OPTIMIZER component called. The OPTIMIZER accumulates the names

## Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers

Umeshwar Dayal  
Computer Corporation of America  
4 Cambridge Center  
Cambridge, Massachusetts 02142-1489

### Abstract

Existing query optimizers focus on Restrict-Project-Join queries. In practice, however, query languages such as SQL and DAPLEX have many powerful features (e.g., control over duplicates, aggregation functions, grouping and quantifiers) that cannot be mapped to the restrict-project-join subset of the relational algebra. Such languages pose an important challenge for query optimization. The semantics of queries that use these features are often described procedurally, and existing query optimizers are severely limited in their tactics for processing such queries.

Consider, for example, the following relations:

```
EMP (Emp#, Name, Dept#, Sal)
DEPT (Dept#, Name, Loc, Mgr)
```

and the following SQL query, which contains a nested subquery block:

```
Query 1
SELECT E.Name
FROM EMP E,
WHERE E.Dept# IN
SELECT D.Dept#
FROM DEPT D
WHERE D.Loc = 'Denver' AND
E.Mgr# = D.Mgr
```

The semantics of SQL prescribe that the tuples of the EMP relation be substituted in turn into the inner subquery block for each tuple E of EMP; the inner block is evaluated to yield a list of Dept# values; if E.Dept# is in this list, then E.Name is inserted into the result. The System R optimizer follows this prescription quite literally, optimizing only the execution of the inner block (after the substitution, the inner block contains two selections and the optimizer considers strategies for efficiently evaluating them) [SEL79].

In [KIM82], Kim showed that some nested SQL queries could be transformed into equivalent "canonical" queries that did not contain nesting; for example, query 1 could be transformed into query 2 (the queries are not quite equivalent, but more on this later):

```
Query 2
SELECT E.Name
FROM EMP E, DEPT D
WHERE E.Dept# = D.Dept# AND
D.Loc = 'Denver' AND
E.Mgr# = D.Mgr
```

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 13th VLDB Conference, Brighton 1987

Surajit Chaudhuri: An Overview of Query Optimization in Relational Systems. PODS 1998: 34-43

Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993: 209-218

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34

Umeshwar Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. VLDB 1987: 197-208

# Suggestions if you are going to build a QO

- **Rule 1: Read lots of papers, especially from the 80s & 90s.**
  - Expect new combinations, only partially new core inventions.
- **Rule 2: Early on, test various workloads on the QO.**
  - QOs harden over time as they “see” new workloads. Let them see more ASAP.
- **Rule 3: Throw away the initial one (or two) and start anew.**
  - The hard part is going to be nitty-gritty details like data structures and pointers to shared objects; e.g., the list of predicates and the query graph structure, ... You will NOT get this right in the first pass. Don't try to patch; be prepared to rewrite.

# Next Lecture

- DB Design
  - ER-diagrams, FDs