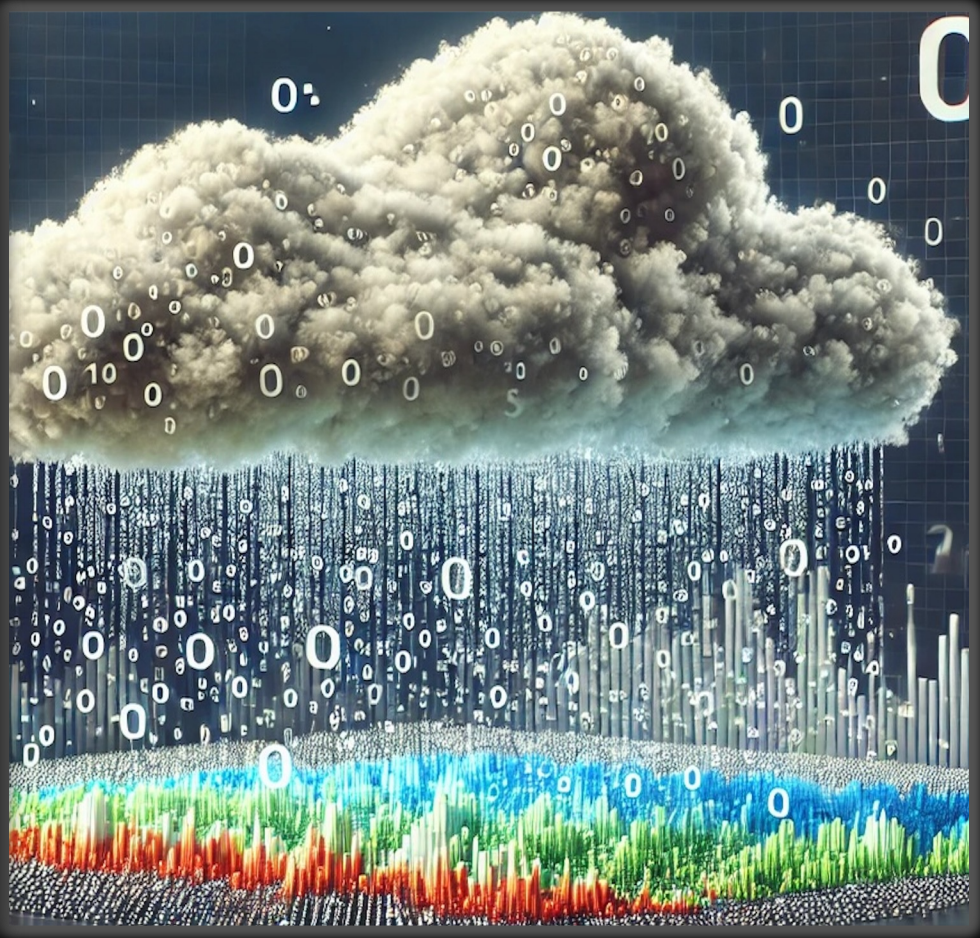
Carnegie Mellon University

Advanced Database Systems (15-721)

Lecture #02

The System R optimizer



Fall 2024 ❯❯ Prof. Jignesh Patel

BACKDROP

The relational data model has recently been proposed.

Key idea: Data independence to insulate applications from changes in internal data formats.

→ Physical DI: Insulate against changes in internal structure, e.g., sorted file, index

→ Logical DI: Insulate from changes in the schema (by using views)

Information Retrieval P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on n-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity
CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

1. Relational Model and Normal Form

1.1. INTRODUCTION

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for noninferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. DATA DEPENDENCIES IN PRESENT SYSTEMS

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

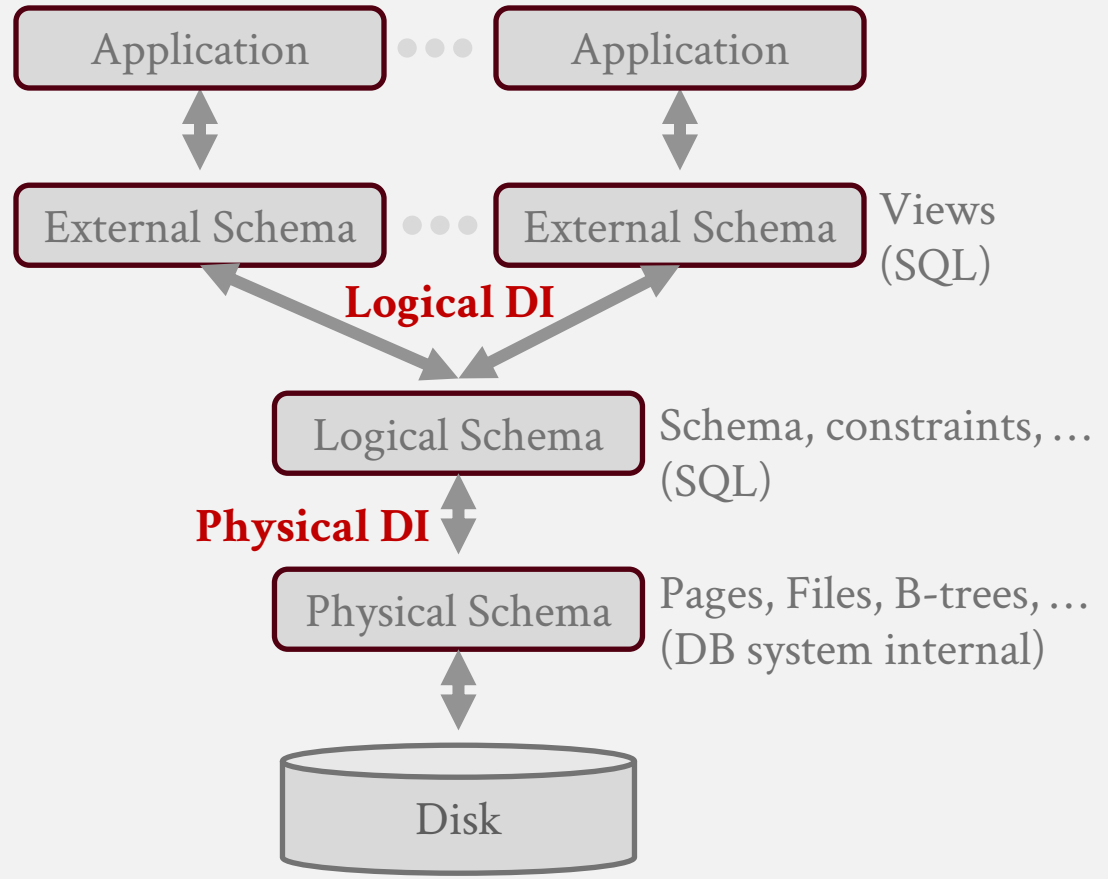
1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

Key Concept: Data Independence (DI)

Isolate the user/application from low level data representation.

→ The user only worries about application logic.

→ Database can optimize the layout (and re-optimize as the workload changes).



MULTI-RELATION QUERY PLANNING

The System R approach

Choice #1: Bottom-up Optimization

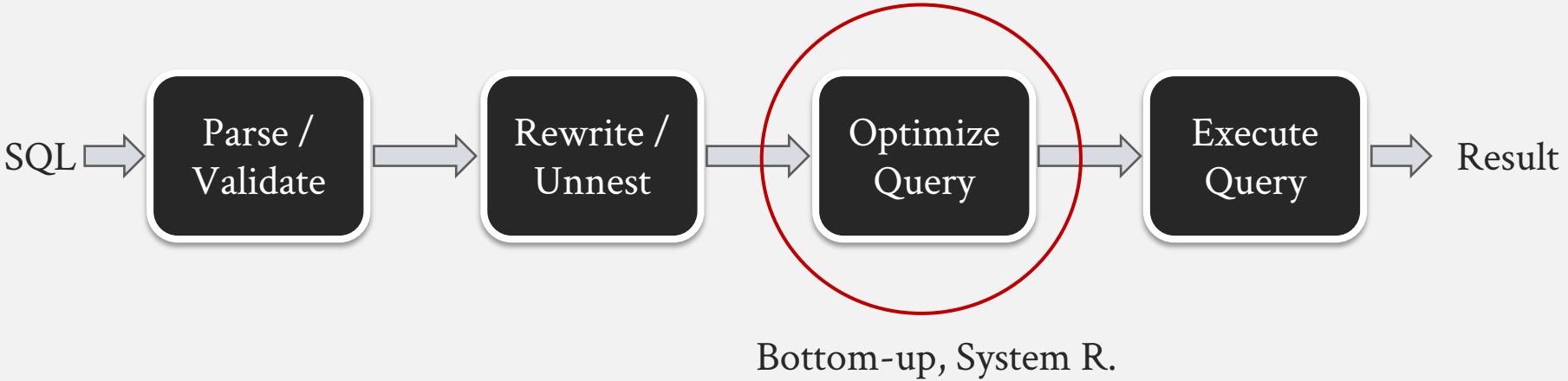
→ Start with nothing and then build up the plan to get to the outcome that you want.

The Volcano/ Cascades style

Choice #2: Top-down Optimization

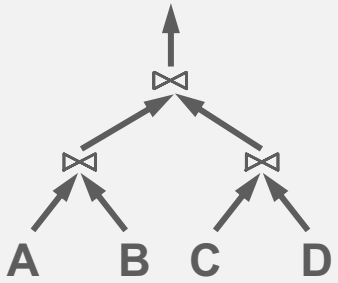
→ Start with the outcome that you want, and then work down the tree to find the optimal plan that gets you to that goal.

SQL Query Optimization Pipeline

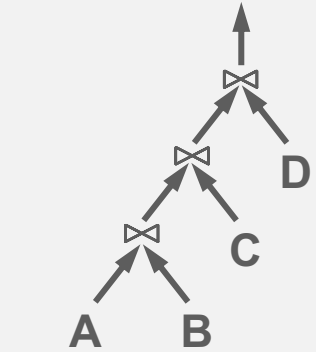


QUERY PLANS

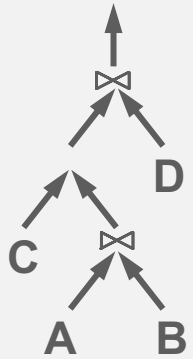
Bushy



Left-deep



Linear



Left-deep plans can be fully pipelined plans.

→ Not all left-deep trees are fully pipelined (e.g., SM join).

→ With Hash join, build all the hash tables first, then stream the outer through the pipeline. An efficient pipeline.

Linear Tree: at least one child in every join node is a base relation

Optimizer Overview

1. Search Space

→ Search only for "low cost" plans.

2. Enumeration Method

→ Need an algorithm to walk through the search space.

3. Cost Estimation

→ Need to estimate the cost of each plan that is enumerated.

→ Want the estimation to be

a) **accurate**: so the estimates are accurate,

b) **fast**: cheap to compute the cost of a plan/operator, and

c) **space-efficient**: It does not take a lot of space to represent any summary structure (e.g., histograms) that is used + want fast construction and update methods for the summary structure.

System R optimizer: Enumerate Left-Deep Plans

Goal

→ Pick the "optimal" join order.

→ Pick the join method for each join operation.

Enumerate using n passes (n = # joins)

→ Find the best 1-relation plan for each relation.

→ Find the best way to join the result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)

→ Pass N: Find the best way to join the result of a (n-1)-relation plan (as outer) to the n^{th} relation. (All n-relation plans.)

For each subset of relations, retain only:

→ The cheapest plan overall, and

→ The cheapest plan for each interesting order of the tuples.

Query

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

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

Conceptual evaluation for any single-block SQL Query:

1. Cross product.
2. Discard tuples (apply the join and selection predicates).
3. Partition into groups using the grouping-list.
4. Evaluate the group expressions (aggregates).
5. Eliminate groups that don't satisfy the group-qualification.
6. Apply the projection.

When the GROUP BY and the HAVING clauses are present.

Query

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

Total: 2M I/Os

Catalog

clustered unclustered unclustered

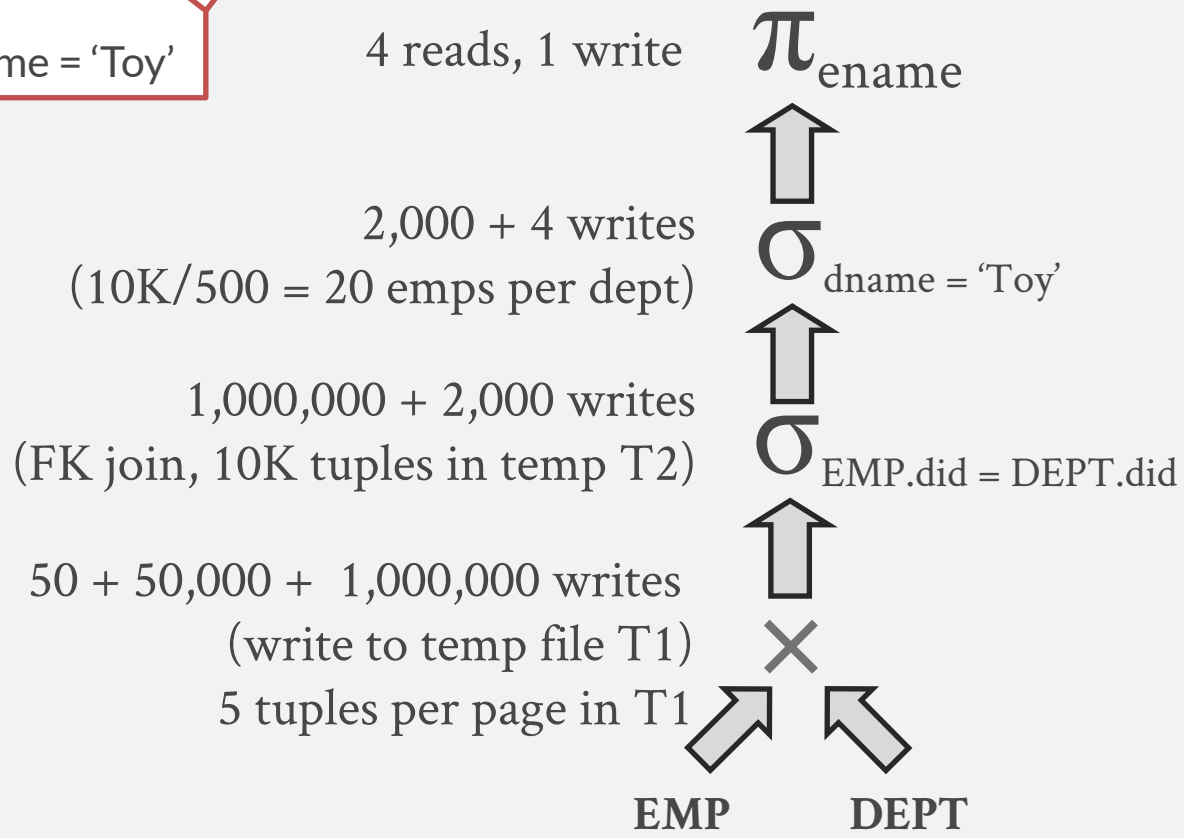
EMP (ssn, ename, addr, sal, did)

10,000 records
1,000 pages

clustered unclustered

DEPT (did, dname, floor, mgr)

500 records
50 pages



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

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

DEPT (**did**, dname, floor, mgr)

Pass 1: EMP: E1: S(EMP), E2: I (EMP.did)

Cost: 1000 1000+100 **KEEP E1 and E2!**

DEPT: D1: S(DEPT), D2: I.(DEPT.did), D3: I(DEPT.dname)

Cost: 50 50+5 3+1 **KEEP D2 and D3**

Pass 2: Consider EMP ⋈ DEPT and DEPT ⋈ EMP

EMP ⋈ DEPT, Alternatives:

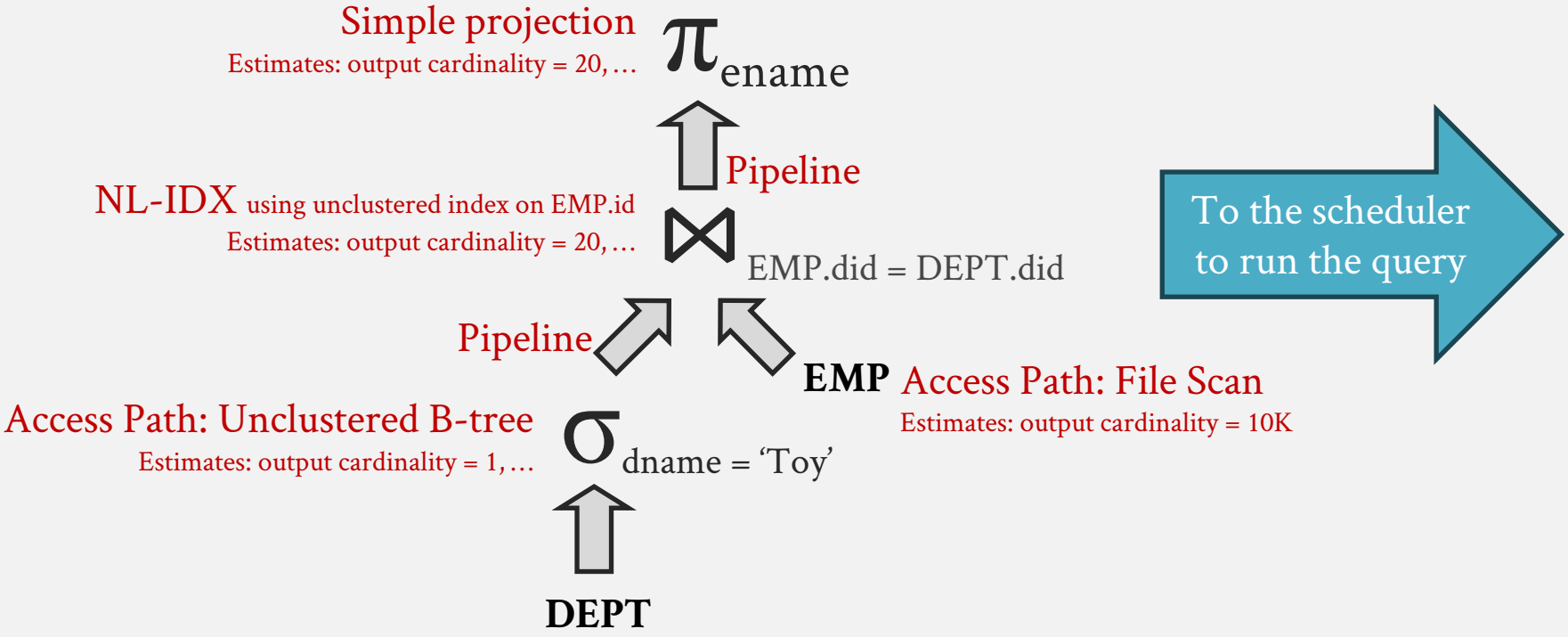
1. E1 ⋈ D2: Algorithms …
2. E1 ⋈ D3: Algorithms …
3. E2 ⋈ D2: Algorithms SM, NL, BNL, NL-IDX, Hash
4. E2 ⋈ D3: Algorithms

Similarly consider DEPT ⋈ EMP

Pick the cheapest 2-relation plan. Done (with join optimization)

Next consider GROUP BY (if present) …

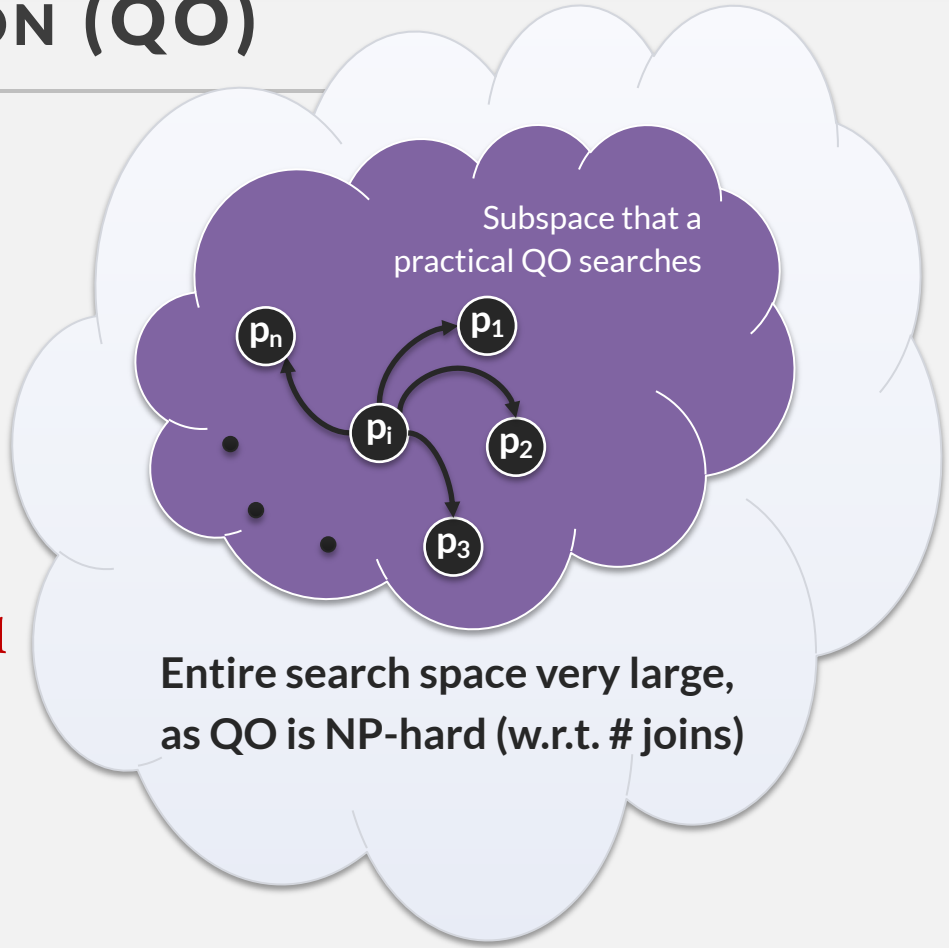
ANNOTATED RA TREE A.K.A. THE PHYSICAL PLAN



Query Optimization (QO)

1. Identify candidate equivalent trees (logical).
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.



EQUIVALENCE

$\sigma_{P1}(\sigma_{P2}(R)) \equiv \sigma_{P2}(\sigma_{P1}(R))$ (σ commutativity)

$\sigma_{P1 \wedge P2 \ldots \wedge Pn}(R) \equiv \sigma_{P1}(\sigma_{P2}(\ldots \sigma_{Pn}(R)))$ (cascading σ)

$\prod_{a1}(R) \equiv \prod_{a1}(\prod_{a2}(\ldots \prod_{ak}(R)\ldots))$, $a_i \subseteq a_{i+1}$ (cascading $\prod$)

$R \bowtie S \equiv S \bowtie R$ (join commutativity)

$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ (join associativity)

$\sigma_P(R \times S) \equiv (R \bowtie_P S)$, if P is a join predicate

$\sigma_P(R \times S) \equiv \sigma_{P1}(\sigma_{P2}(R) \bowtie_{P4} \sigma_{P3}(S))$, where $P = p1 \wedge p2 \wedge p3 \wedge p4$

$\prod_{A1,A2,\ldots An}(\sigma_P(R)) \equiv \prod_{A1,A2,\ldots An}(\sigma_P(\prod_{A1,\ldots An,\ B1,\ldots\ BM}R))$, where B1 … BM are columns in P

…

System R: Cost Estimation

Cost = W*CPU cost + IO Cost

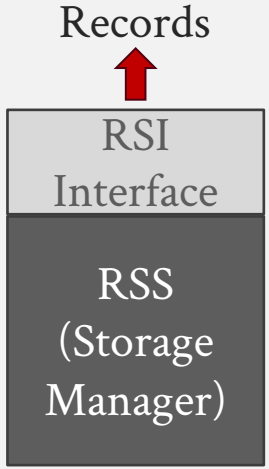
CPU Cost: based on tuples accessed from the storage layer.

→ Weighted by a magic constant W.

IO Cost: more complicated.

→ Keep statistics in the database for each table and index.

- → Table: #tuples, #pages, #non-empty pages in a segment, …
- → Index: #distinct keys, #pages, high-key, low-key, …



System R: Cost Estimation

Estimating the selectivity of each predicate

- colA = value:
  - Index exists: 1/#distinct keys in the index
  - Else 1/10

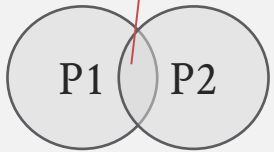
Many such magic numbers in QOs in practice, which causes a lot of pain.

- colA > value:
  - Index exists: (high key – value) / (high key – low key)

Complex predicates

- Conjunct: p1 and p2, e.g., salary > 100K and age < 30
  - Estimate(p1) * Estimate (p2)
  - Note assumes p1 and p2 are independent.
- Disjunct: p1 or p2
  - Estimate(p1) + Estimate (p2) – Estimate(p1)*Estimate(p2)
- Negation: not p1
  - 1 - Estimate(p1)

Last term: Don't double count when both p1 and p2 are true.



Modern Cardinality Estimation Methods

Histograms:

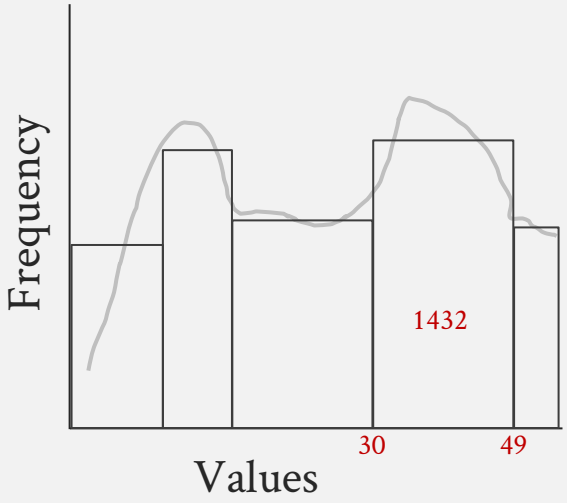
Most commonly used.

Sketches:

Neat theory, and big applications in applied in streaming settings.

HISTOGRAMS

Estimate the distribution of values using discrete "bin."



For each bin store: bin boundaries, and # values in the bins

Histograms

Generally, an equi-depth histogram is better than an equi-width histogram.

An even better approach is to use an equi-width with the most common value (MCV).

→ Takes most of the "pain" away from the "heavy-hitters" that tend to mess up the uniform data assumption.

Proportionately adjust the count for predicates that span only a portion of the bucket boundary.

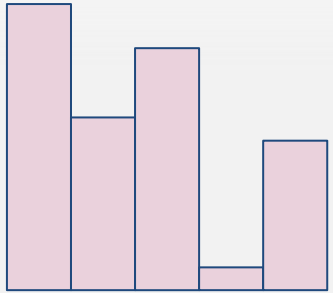
→ Row estimation in PG: https://www.postgresql.org/docs/current/row-estimation-examples.html

Join cardinality estimation is a harder problem.

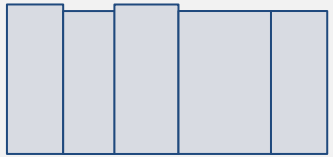
→ Can "join" the two histograms (multiply the counts within bucket pairs).

→ Multi-dimensional histograms can be constructed, but these are complex.

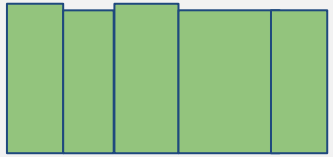
→ System R estimate: |R|x|S|/max(uniq_r, uniq_s).
It tends to underestimate the actual join cardinality.



Equi-width



Equi-depth



Value	Selectivity
3	0.01%
13	0.1%
46	0.04%

Equi-depth with most common values

Histogram Construction (Single Column)

If scanning the full dataset/table is expensive, sample data from the table and build the histogram.

→ A good rule-of-thumb is to sample 300 X # histogram bins.

An algorithm to build a histogram:

1. Sample the column.
2. Sort the values.
3. Create a list of the most common values.
4. Walk through the sorted list, marking bin boundaries to create the equi-depth histogram.

SKETCH

A probabilistic data structure to capture statistical properties over an event data stream.

h_0	0	0	0	0	0	0	0
h_1	0	0	0	0	0	0	0
h_2	0	0	0	0	0	0	0

https://hkorte.github.io/slides/cmsketch/#/7

Pessimistic Cardinality Estimation

Under-estimation can be worse than over-estimation.

→ These methods provide guaranteed upper bound/pessimistic cardinality estimation

→ Weighted by a magic constant W.

Need supporting stats, like degree and count stats/sketches, for each column involved.

Then, plug this information into an entropy formula one can derive for the query at hand.

```
SELECT
    *
FROM
    pseudonym,
    cast,
    movie_companies,
    company_name
WHERE
    pseudonym.person_id = cast.person_id AND
    cast.movie_id_id = movie_companies.movie_id AND
    movie_companies.company_id = company_name.id;
```

$$Q(x,y,z,w) :- pseudo(x,y), cast(y,z), mc(z,w), cn(w)$$

$$\left|Q(x,y,z,w)\right| \le \min \begin{cases} c_{\text{pseudo}} \cdot d^{y}_{\text{cast}} \cdot d^{z}_{\text{mc}} \\ c_{\text{pseudo}} \cdot d^{y}_{\text{cast}} \cdot c_{\text{cn}} \\ c_{\text{pseudo}} \cdot c_{\text{mc}} \\ c_{\text{pseudo}} \cdot d^{w}_{\text{mc}} \cdot c_{\text{cn}} \\ d^{y}_{\text{pseudo}} \cdot c_{\text{cast}} \cdot d^{z}_{\text{mc}} \\ d^{y}_{\text{pseudo}} \cdot c_{\text{cast}} \cdot c_{\text{cn}} \\ d^{y}_{\text{pseudo}} \cdot d^{z}_{\text{cast}} \cdot c_{\text{mc}} \\ d^{y}_{\text{pseudo}} \cdot d^{z}_{\text{cast}} \cdot d^{w}_{\text{mc}} \cdot c_{\text{cn}} \end{cases}$$

Walter Cai, Magdalena Balazinska, Dan Suciu: Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. SIGMOD 2019.

Other Methods

ML-based methods. Two types: data-driven and query-driven.

Data-driven.

→ Denormalize the data, add extra columns, and then learn the distributions.

→ Slow training times and large model size.

Query-driven.

→ Take queries from past workloads, encode the queries (for ML), and feed the encoded data/features to a supervised learning model.

→ Need lots of queries.

COMPARING DIFFERENT ESTIMATION METHODS

Category	Method	Data / Workload					
		IMDB / JOB-LIGHT			STATS / STATS-CEB		
		End-to-End Time	Exec. + Plan Time	Improvement	End-to-End Time	Exec. + Plan Time	Improvement
Baseline	PostgreSQL	3.67h	3.67h + 3s	0.0%	11.34h	11.34h + 25s	0.0%
	TrueCard	**3.15h**	**3.15h + 3s**	**14.2%**	**5.69h**	**5.69h + 25s**	**49.8%**
Traditional	MultiHist	3.92h	3.92h + 30s	−6.8%	14.55h	14.53h + 79s	−28.3%
	UniSample	4.87h	4.84h + 96s	−32.6%	> 25h	−−	−−
	WJSample	4.15h	4.15h + 23s	−13.1%	19.86h	19.85h + 45s	−75.0%
	PessEst	3.47h	3.38h + 324s	5.4%	6.42h	6.10h + 1,135s	43.4%
Query-driven	MSCN	3.50h	3.50h + 12s	4.6%	8.13h	8.11h + 46s	28.3%
	LW-XGB	4.31h	4.31h + 8s	−17.4%	> 25h	−−	−−
	LW-NN	3.63h	3.63h + 9s	1.1%	11.33h	11.33h + 34s	0.0%
	UAE-Q	3.65h	3.55h+356s	−1.9%	11.21h	11.03h+645s	1.1%
Data-driven	NeuroCard^E	3.41h	3.29h + 423s	6.8%	12.05h	11.85h + 709s	−6.2%
	BayesCard	3.18h	3.18h + 10s	13.3%	7.16h	7.15h + 35s	36.9%
	DeepDB	3.29h	3.28h + 33s	10.3%	6.51h	6.46h + 168s	42.6%
	FLAT	3.21h	3.21h + 15s	12.9%	5.92h	5.80h + 437s	47.8%
Query + Data	UAE	3.71h	3.60h + 412s	−2.7%	11.65h	11.46h + 710s	−0.02%

Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, Bin Cui: Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. CoRR abs/2109.05877 (2021)

Comparing Different Estimation Methods

Shows that data-driven ML methods are generally better than query-driven ML methods.

The ML-based methods, in my opinion, overfit the training set.

All methods degrade as the number of joins increases (no surprise) but point to the need for runtime adaptiveness.

Summary of Estimation Methods

Start with at least equi-depth histogram with MCV

No magic bullet for join cardinality estimation.

Newer methods that look good on paper seem to overfit to the benchmark training set, and are complex.

Think holistically: QO + QP, not QO -> QP
(more on this later.)

Nested Queries

```
CREATE TABLE Employee (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    salary DECIMAL(10, 2),
    mgrid INT, FOREIGN KEY (mgrid) REFERENCES Employee(id));
```

Q1

```
SELECT name
FROM Employee
WHERE salary >
     (SELECT AVG(salary)
      FROM Employee);
```

Find those who make more than the average employee salary.

Q2

```
SELECT E1.name
FROM Employee E1
WHERE E1.salary > (
    SELECT E2.salary
    FROM Employee E2
    WHERE E2.id = E1.mgrid);
```

Employees who earn more than their managers.

Q2'

```
SELECT E1.name
FROM Employee E1
JOIN Employee E2 ON
            E1.mgrid = E2.id
WHERE E1.salary > E2.salary;
```

Q3

```
SELECT E1.name
FROM Employee E1
WHERE E1.salary = (
    SELECT MAX(E2.salary)
    FROM Employee E2
    WHERE E2.mgrid = E1.mgrid);
```

Employees who have the highest salary in their department (under the same manager).

Q3'

```
SELECT E1.name
FROM Employee E1
JOIN (
    SELECT mgrid, MAX(salary) AS max_salary
    FROM Employee
    GROUP BY mgrid
) AS E2
ON E1.mgrid = E2.mgrid
AND E1.salary = E2.max_salary;
```

Nested Queries

SQL is quite liberal with subqueries.

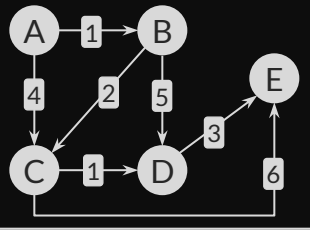
→ Allowed in the SELECT, FROM, and WHERE clauses.

→ Powerful mechanism called Common Table Expressions (CTEs).

```
WITH RECURSIVE shortest_path AS (
    -- Anchor member: Start from the source node
    SELECT
        source AS f,
        destination AS t,
        weight,
        (source || '->' || destination) AS path,
        source || ',' || destination AS visited_nodes
    FROM
        edges
    WHERE
        source = '{source_node}'

    UNION ALL

    -- Recursive member: Find the next node in the path
    SELECT
        e.source AS f,
        e.destination AS t,
        sp.weight + e.weight AS weight,
        (sp.path || '->' || e.destination) AS path,
        sp.visited_nodes || ',' || e.destination AS visited_nodes
    FROM
        edges e
    JOIN
        shortest_path sp
    ON
        e.source = sp.t
    WHERE
        instr(sp.visited_nodes, e.destination) = 0  -- Prevent cycles by checking visited nodes
)

-- Final query: Select the shortest path to the destination
SELECT
    t AS destination,
    weight,
    path AS full_path
FROM
    shortest_path
WHERE
    t = '{destination_node}'
ORDER BY
    weight ASC
LIMIT 1;
```

```
CREATE TABLE edges (
    source TEXT,
    destination TEXT,
    weight INTEGER
)
```

```
INSERT INTO edges (source, destination, weight) VALUES
('A', 'B', 1),
('A', 'C', 4),
('B', 'C', 2),
('B', 'D', 5),
('C', 'D', 1),
('D', 'E', 3),
('C', 'E', 6);
```



Destination	Weight	Path
E	7	A->B->C->D->E

SQL Translation

$T_1 \bowtie_p T_2 \quad := \quad \sigma_p(T_1 \times T_2).$ Inner Join

$T_1 \ltimes_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$

Dependent Join

T. Neumann, V. Leis, A. Kemper: The Complete Story of Joins (in HyPer). BTW 2017.

3.4 Translating SQL Queries

Putting it all together we can now translate arbitrary SQL queries into relational algebra using the following high-level algorithm:

1. translate the *from* clause, from left to right
   a) for each entry produce an operator tree
   b) if there is no correlation combine with the previous tree using $\times$, otherwise use $\ltimes$
   c) the result is a single operator tree
2. translate the *where* clause (if it exists)
   a) for exists/not exists/unique and quantified subqueries add the subquery on top of the current tree using $\bowtie^{M:m}$. Translate the expression itself with *m*.
   b) for scalar subqueries, introduce $\bowtie^{1}$ and translate the expression with the (single) result column/row.
   c) all other expressions are scalars, translate them directly
   d) the result is added to the top of the current tree using σ
3. translate the *group-by* clause (if it exists)
   a) translate the grouped expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Γ (group-by)
4. translate the *having* clause (if it exists)
   a) logic is identical to the *where* clause
5. translate the *select* clause
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Π
6. translate the *order by* clause (if it exists)
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using a *sort* operator

Query Optimization: Summary

Critical to high-performance queries, especially for analytics workloads.

The bottom-up, dynamic programming style invented by System R got the field going.

Need a good enumeration strategy and cost model. Bushy plans, and sometimes cartesian products, can result in better plans.

QO continues to be a hard problem, especially for queries with large # of joins.