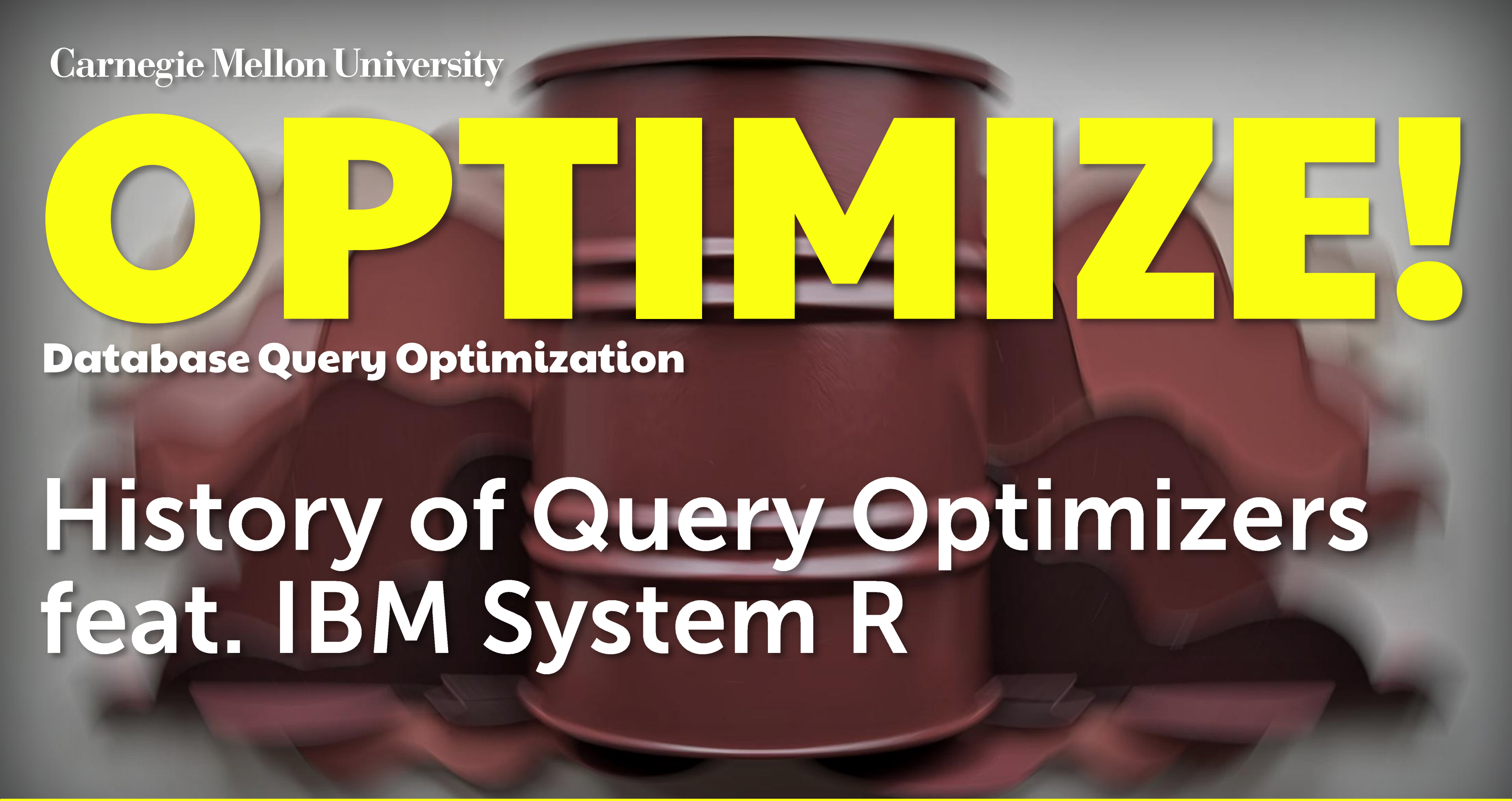


Carnegie Mellon University

OPTIMIZE!

Database Query Optimization

History of Query Optimizers feat. IBM System R

SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

LAST CLASS

Course objectives and expectations.
→ I will assign note taking schedule tonight.

Motivation for why query optimization is important and a hard problem.

TODAY'S AGENDA

Background

Heuristics

Heuristics + Cost-based Search

THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural code.

→ Example: CODASYL

The developer had to choose access paths and execution ordering based on the current database contents.

→ If the database changes, then the developer must rewrite the query code.

1973 ACM Turing Award Lecture

The Turing Award citation read by Richard G. Canning, chairman of the 1973 Turing Award Committee, at the presentation of this lecture on August 28 at the ACM Annual Conference in Atlanta:

A significant change in the computer field in the last five to eight years has been made in the way we treat and handle data. In the early days of our field, data was intimately tied to the application programs that used it. Now we see that we want to break that tie. We want data that is independent of the application programs that use it—that is, data that is organized and structured to serve many applications and many users. What we seek is the *data base*.

This movement toward the data base is in its infancy. Even so, it appears that there are now between 1,000 and 2,000 true data base management systems installed worldwide. In ten years very likely, there will be tens of thousands of such systems. Just from the quantities of installed systems, the impact of data bases promises to be huge.

This year's recipient of the A.M. Turing Award is one of the real pioneers of data base technology. No other individual has had the influence that he has had upon this aspect of our field. I single out three prime examples of what he has done. He was the creator and principal architect of the first commercially available data base management system—the Integrated Data Store—originally developed from 1961 to 1964.[1,2,3,4] I-D-S is today one of the three most widely used data base management systems. Also, he was one of the founding members of the CODASYL Data Base Task Group, and served on that task group from 1966 to 1968. The specifications of that task group are being implemented by many suppliers in various parts of the world.[5,6] Indeed, currently these specifications represent the only proposal of stature for a common architecture for data base management systems. It is to his credit that these specifications, after extended debate and discussion, embody much of the original thinking of the Integrated Data Store. Thirdly, he was the creator of a powerful method for displaying data relationships—a tool for data base designers as well as application system designers.[7,8]

His contributions have thus represented the union of imagination and practicality. The richness of his work has already had, and will continue to have, a substantial influence upon our field.

I am very pleased to present the 1973 A.M. Turing Award to Charles W. Bachman.

The Programmer as Navigator

by Charles W. Bachman

This year the whole world celebrates the five-hundredth birthday of Nicolaus Copernicus, the famous Polish astronomer and mathematician. In 1543, Copernicus published his book, *Concerning the Revolutions of Celestial Spheres*, which described a new theory about the relative physical movements of the earth, the planets, and the sun. It was in direct contradiction with the earth-centered theories which had been established by Ptolemy 1400 years earlier.

Copernicus proposed the heliocentric theory, that planets revolve in a circular orbit around the sun. This theory was subjected to tremendous and persistent criticism. Nearly 100 years later, Galileo was ordered to appear before the Inquisition in Rome and forced to state that he had given up his belief in the Copernican theory. Even this did not placate his inquisitors, and he was sentenced to an indefinite prison term, while Copernicus's book was placed upon the Index of Prohibited Books, where it remained for another 200 years.

I raise the example of Copernicus today to illustrate a parallel that I believe exists in the computing or, more properly, the information systems world. We have spent the last 50 years with almost Ptolemaic information systems. These systems, and most of the thinking about systems, were based on a "computer centered" concept. (I choose to speak of 50 years of history rather than 25, for I see today's information systems as dating from the beginning of effective punched card equipment rather than from the beginning of the stored program computer.)

Just as the ancients viewed the earth with the sun revolving around it, so have the ancients of our information systems viewed a tab machine or computer with a sequential file flowing through it. Each was an

Author's address: Honeywell Information Systems, Inc., 200 Smith Street, Waltham, MA 02154.

The abstract, key words, etc., are on page 654.

[1–8] Footnotes are on page 658.

653

Communications
of
the ACM

November 1973
Volume 16
Number 11

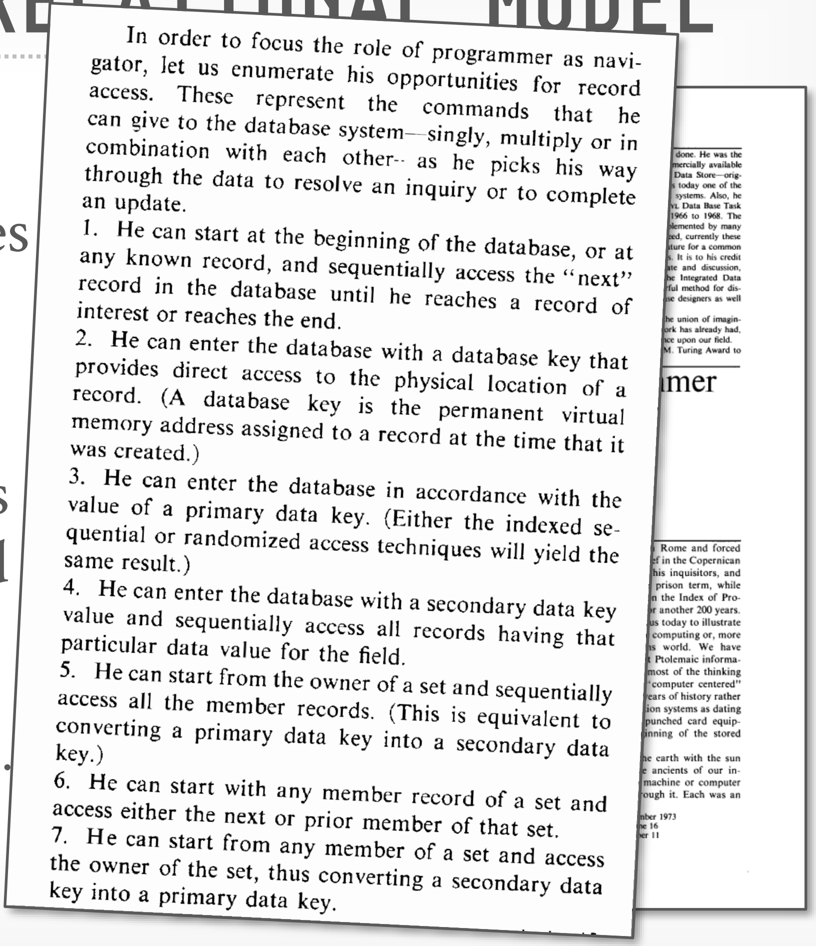
THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural code.

→ Example: CODASYL

The developer had to choose access paths and execution ordering based on the current database contents.

→ If the database changes, then the developer must rewrite the query code.



In order to focus the role of programmer as navigator, let us enumerate his opportunities for record access. These represent the commands that he can give to the database system—singly, multiply or in combination with each other-- as he picks his way through the data to resolve an inquiry or to complete an update.

1. He can start at the beginning of the database, or at any known record, and sequentially access the "next" record in the database until he reaches a record of interest or reaches the end.
2. He can enter the database with a database key that provides direct access to the physical location of a record. (A database key is the permanent virtual memory address assigned to a record at the time that it was created.)
3. He can enter the database in accordance with the value of a primary data key. (Either the indexed sequential or randomized access techniques will yield the same result.)
4. He can enter the database with a secondary data key value and sequentially access all records having that particular data value for the field.
5. He can start from the owner of a set and sequentially access all the member records. (This is equivalent to converting a primary data key into a secondary data key.)
6. He can start with any member record of a set and access either the next or prior member of that set.
7. He can start from any member of a set and access the owner of the set, thus converting a secondary data key into a primary data key.

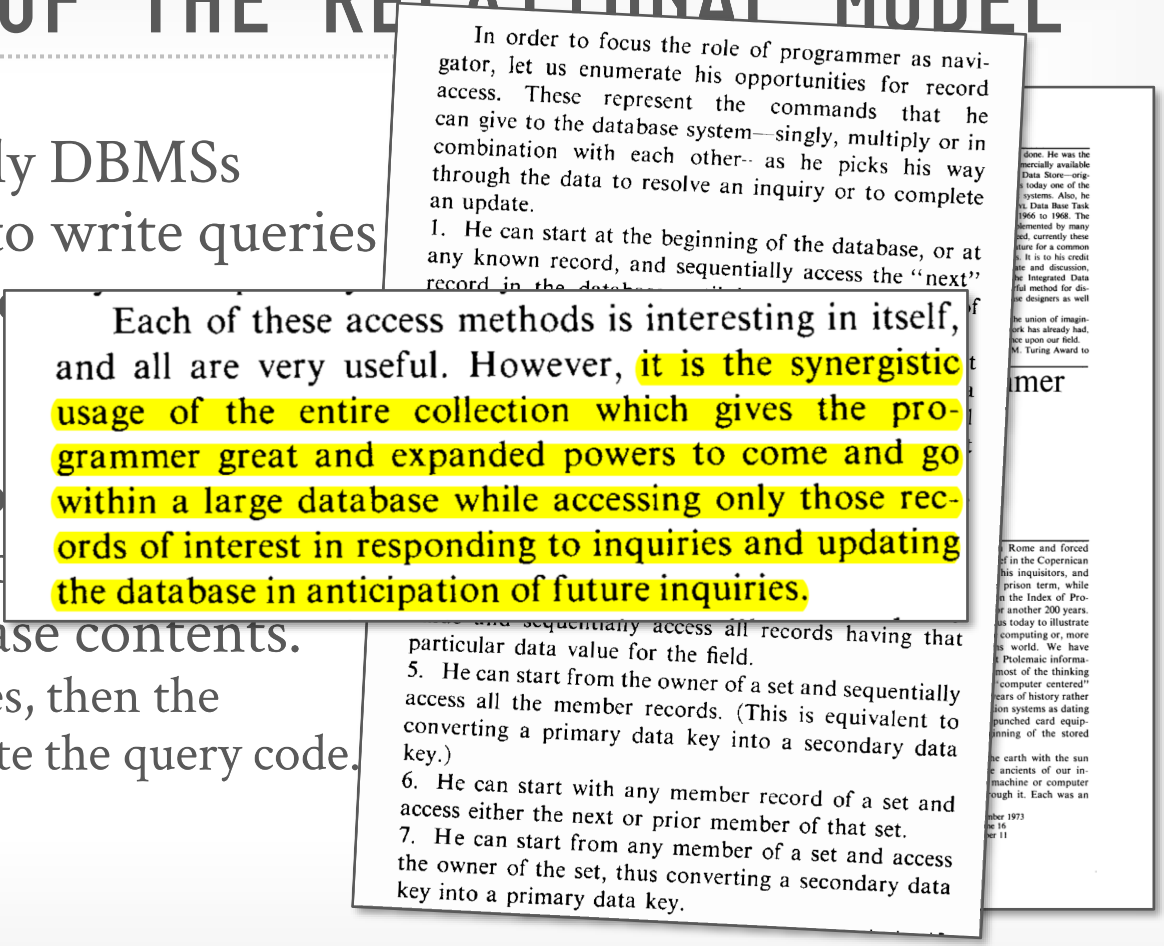
THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural cod

→ Example: CODASYL

The developer had to
paths and execution
on the current database contents.

→ If the database changes, then the developer must rewrite the query code.



In order to focus the role of programmer as navigator, let us enumerate his opportunities for record access. These represent the commands that he can give to the database system—singly, multiply or in combination with each other-- as he picks his way through the data to resolve an inquiry or to complete an update.

1. He can start at the beginning of the database, or at any known record, and sequentially access the "next"

Each of these access methods is interesting in itself, and all are very useful. However, it is the synergistic usage of the entire collection which gives the programmer great and expanded powers to come and go within a large database while accessing only those records of interest in responding to inquiries and updating the database in anticipation of future inquiries.

particular data value for the field.

5. He can start from the owner of a set and sequentially access all the member records. (This is equivalent to converting a primary data key into a secondary data key.)

6. He can start with any member record of a set and access either the next or prior member of that set.

7. He can start from any member of a set and access the owner of the set, thus converting a secondary data key into a primary data key.

THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural code.

→ Example: CODASYL

The developer had to choose access paths and execution ordering based on the current database contents.

→ If the database changes, then the developer must rewrite the query code.

Retrieve the names of artists that appear on the DJ Mooshoo Tribute mixtape.

```
PROCEDURE GET_ARTISTS_FOR_ALBUM;
BEGIN
  /* Declare variables */
  DECLARE ARTIST_RECORD ARTIST;
  DECLARE APPEARS_RECORD APPEARS;
  DECLARE ALBUM_RECORD ALBUM;

  /* Start navigation */
  FIND ALBUM USING ALBUM.NAME = "Mooshoo Tribute"
      ON ERROR DISPLAY "Album not found" AND EXIT;

  /* For each appearance on the album */
  FIND FIRST APPEARS WITHIN APPEARS_ALBUM OF ALBUM_RECORD
      ON ERROR DISPLAY "No artists found for this album" AND EXIT;

  /* Loop through the set of APPEARS */
  REPEAT
      /* Navigate to the corresponding artist */
      FIND OWNER WITHIN ARTIST_APPEARS OF APPEARS_RECORD
          ON ERROR DISPLAY "Error finding artist";
      /* Display artist name */
      DISPLAY ARTIST_RECORD.NAME;
      /* Move to the next APPEARS record in the set */
      FIND NEXT APPEARS WITHIN APPEARS_ALBUM OF ALBUM_RECORD
          ON ERROR EXIT;
  END REPEAT;
END PROCEDURE;
```

THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural code.

→ Example: CODASYL

The developer had to choose access paths and execution ordering based on the current database contents.

→ If the database changes, then the developer must rewrite the query code.

Retrieve the names of artists that appear on the DJ Mooshoo Tribute mixtape.

```
PROCEDURE GET_ARTISTS_FOR_ALBUM;
BEGIN
  /* Declare variables */
  DECLARE      ST_RECORD ARTIST;
  DECLA         S_RECORD APPEARS;
  DECLA           CORD ALBUM;

  /* Start    gation
  FIND ALBUM USING                               "
     ON ERROR DI

  /* For each a
  FIND FIRST A       WITHIN A      BUM OF AL     ECORD
     ON ERROR     LAY "No arti     ound for thi     um" AND EXIT;

  /* Loop thro     he set of A       */
  REPEAT
     /* Navigat               ond
     FIND OWNER              _APPEAR              _RECORD
        ON ERROR                                   t";
     /* Di     artis
     DIS         ST_RECO
     /         e next              in the
     F        APPEARS WI    N A    ARS    BUM OF AL        D
              RROR EXIT;

  END REPEAT;
END PROCEDURE;
```

THE DAWN OF THE RELATIONAL MODEL

In the late 1960s, early DBMSs required developers to write queries using procedural code.

→ Example: CODASYL

The developer had to choose access paths and execution ordering based on the current database contents.

→ If the database changes, then the developer must rewrite the query code.

Retrieve the names of artists that appear on the DJ Mooshoo Tribute mixtape.

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

RELATIONAL MODEL

Structure: The definition of the database's relations and their contents independent of their physical representation.

Integrity: Ensure the database's contents satisfy constraints.

Manipulation: Declarative API for accessing and modifying a database's contents via sets.

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

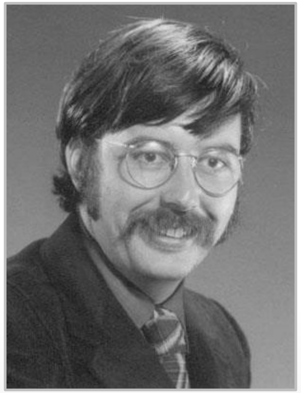
Volume 13 / Number 6 / June, 1970 Communications of the ACM 377

RELATIONAL MODEL

Early relational DBMS implementations:

- → **Peterlee Relational Test Vehicle** – IBM Research (UK)
- → **System R** – IBM Research (San Jose)
- → **INGRES** – U.C. Berkeley
- → **Oracle** – Larry Ellison
- → **Mimer** – Uppsala University

Gray



Stonebraker

Ellison

HISTORY OF QUERY OPTIMIZERS

Choice #1: Heuristics
→ INGRES (1970s), Oracle (until mid 1990s)

Choice #2: Heuristics + Cost-based Join Search
→ System R (1970s), early IBM DB2

Choice #3: Stratified Search
→ IBM STARBURST (late 1980s), now IBM DB2 + Oracle

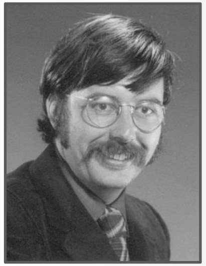
Choice #4: Unified Search
→ Volcano/Cascades in 1990s, now MSSQL + Greenplum

Choice #5: Randomized Search
→ Academics in the 1980s, current Postgres

HEURISTIC-BASED OPTIMIZATION

Define static rules that transform logical operators to a physical plan <u>without</u> a cost model.

- → Perform most restrictive selection early
- → Perform all selections before joins
- → Predicate/Limit/Projection pushdowns
- → Join ordering based on simple rules or cardinality estimates



Stonebraker

Examples: INGRES (until mid-1980s) and Oracle (until early-1990s), MongoDB, most new DBMSs.

QUERY PROCESSING IN A RELATIONAL DATABASE MANAGEMENT SYSTEM
VLDB 1979

RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are equivalent if they generate the same set of tuples.

These equivalences allow the DBMS to manipulate and transform a query plan into different forms without effecting the correctness of its output.

→ This is how a heuristic-based optimizer identifies better query plans without a cost model.

RELATIONAL ALGEBRA EQUIVALENCES

Selections:

→ Perform filters as early as possible.

→ Breakup a complex predicate into conjunctive clauses and push down to lowest part of plan as possible.

$$\sigma_{p1 \wedge p2 \wedge \ldots pn}(R) = \sigma_{p1}(\sigma_{p2}(\ldots\sigma_{pn}(R)))$$

Simplify complex predicates:

→ `(X=Y AND Y=3) → X=3 AND Y=3`

→ `(X=1+1) → X=2`

→ `(X=YEAR('1/15/2025') → X=2025`

RELATIONAL ALGEBRA EQUIVALENCES

Joins:

- → Commutative:
  $R \bowtie S = S \bowtie R$
- → Associative:
  $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

The number of different join orderings for an n-way join is a **Catalan Number** (≈4^n)

- → Exhaustive enumeration will be too slow.

LOGICAL QUERY OPTIMIZATION

Split Conjunctive Predicates

Predicate Pushdown

Replace Cartesian Products with Joins

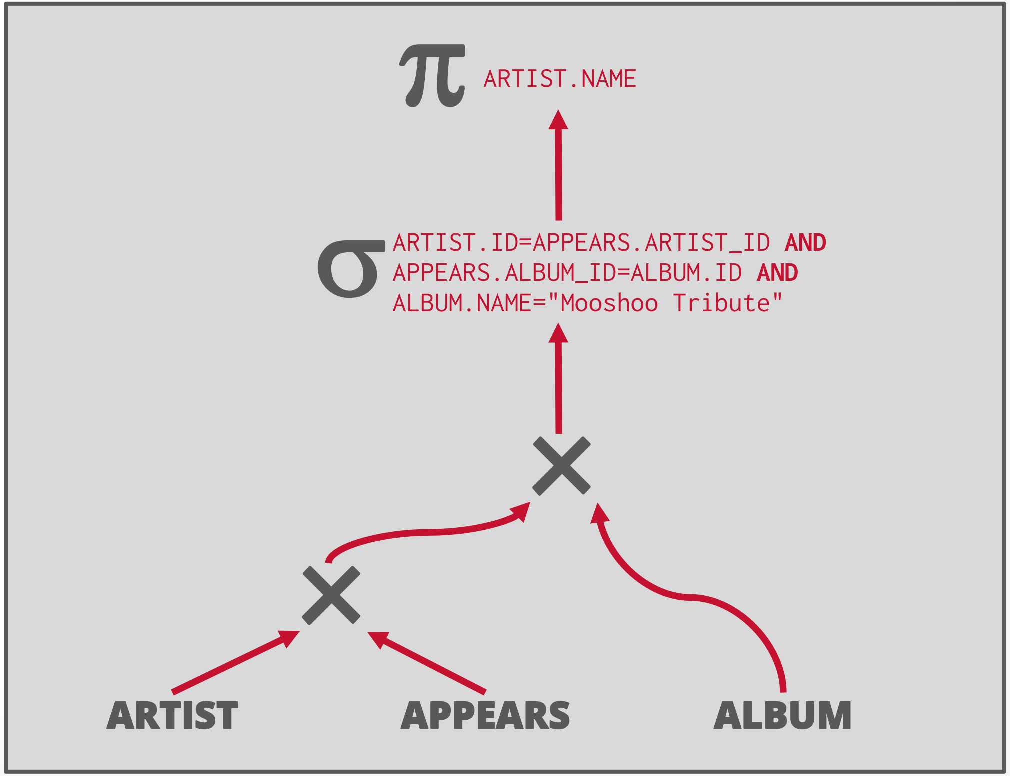
Projection Pushdown

Source: Thomas Neumann

SPLIT CONJUNCTIVE PREDICATES

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

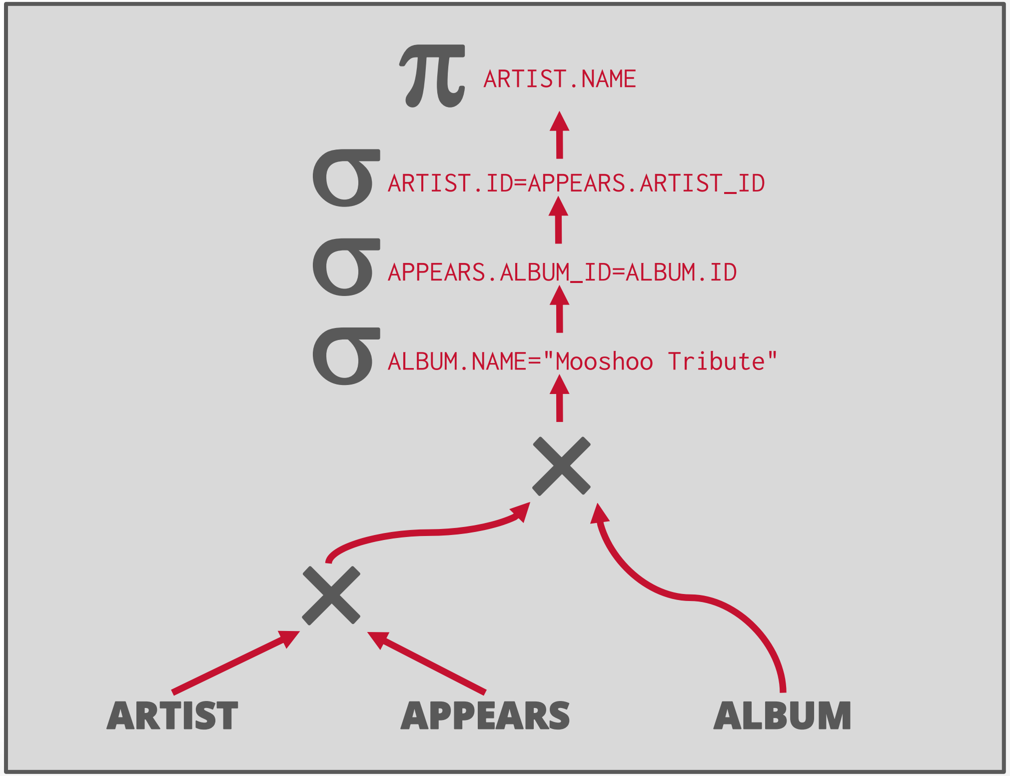
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



SPLIT CONJUNCTIVE PREDICATES

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

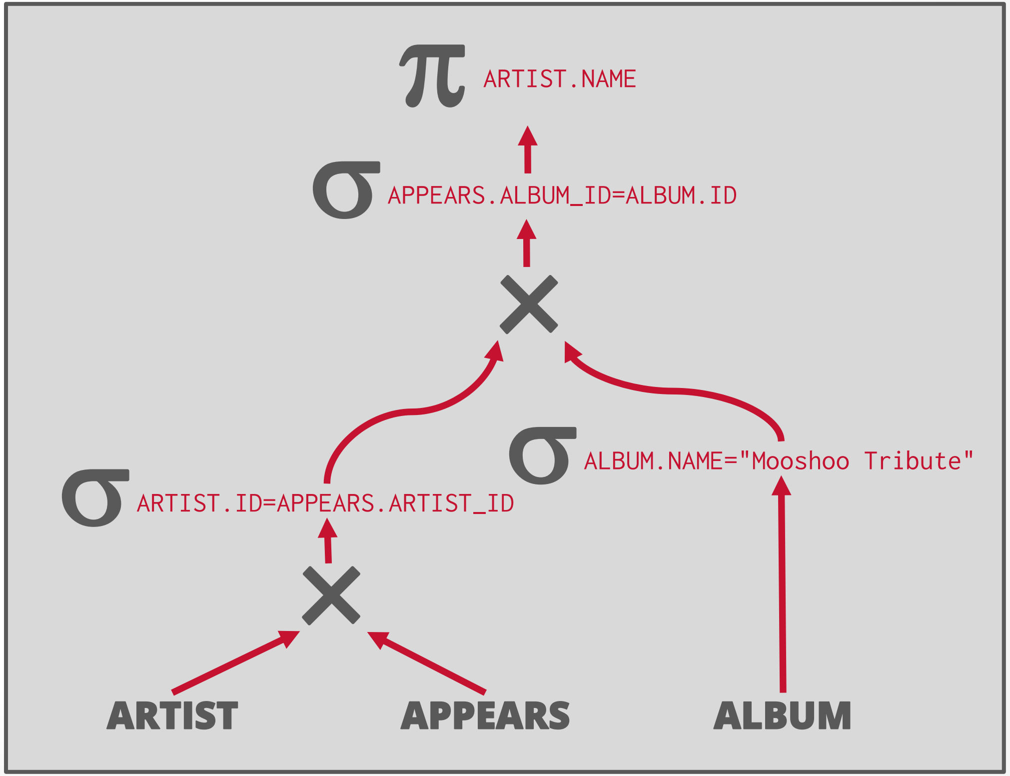
Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



PREDICATE PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

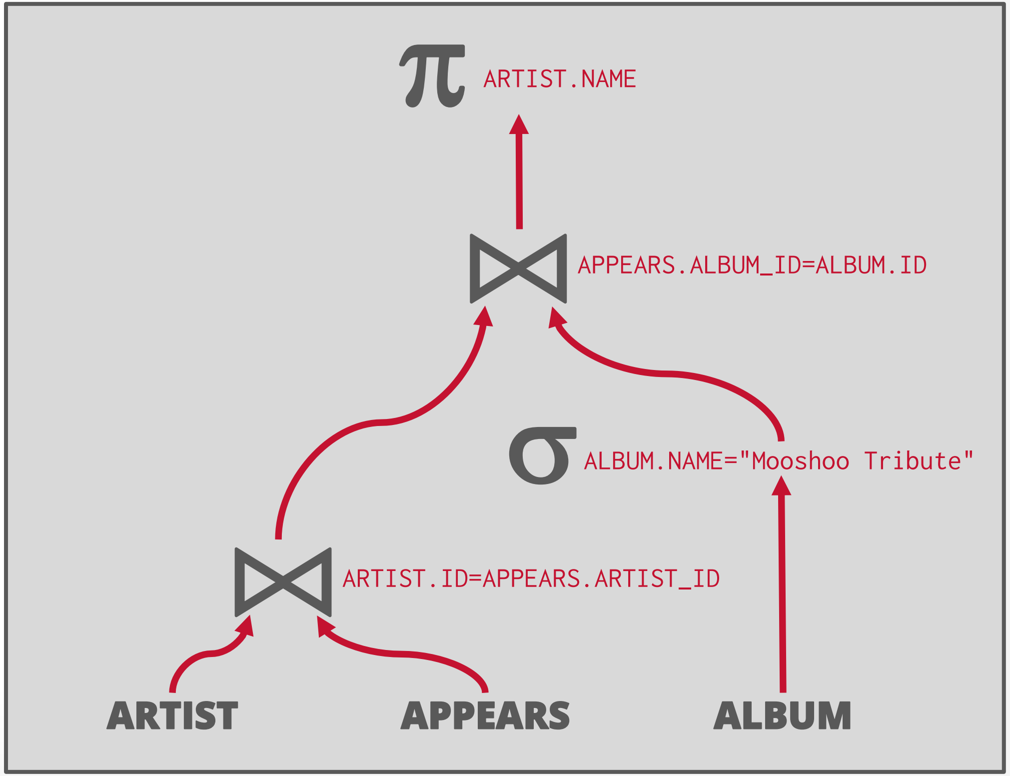
Move the predicate to the lowest point in the plan after Cartesian products.



REPLACE CARTESIAN PRODUCTS

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

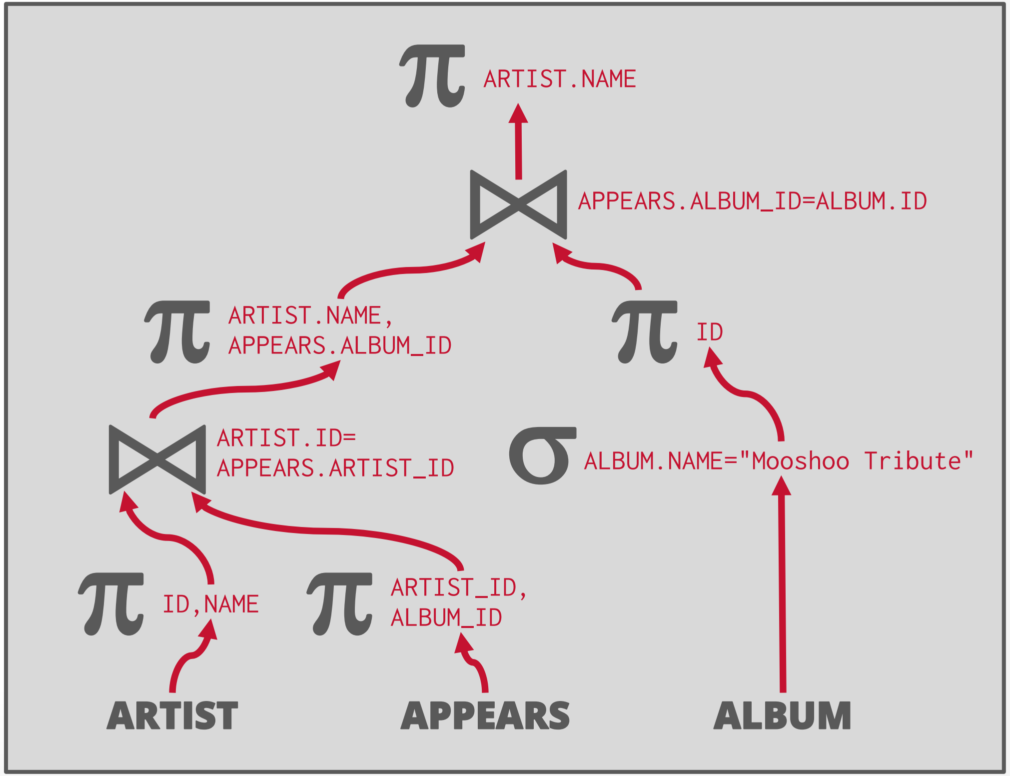
Replace all Cartesian Products with inner joins using the join predicates.



PROJECTION PUSHDOWN

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Mooshoo Tribute"
```

Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Query #1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
  FROM ALBUM
 WHERE ALBUM.NAME="Mooshoo Tribute"
```

Query #2

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, TEMP1
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ID
```

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Query #1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
  FROM ALBUM
 WHERE ALBUM.NAME="Mooshoo Tribute"
```

Query #2

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, TEMP1
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ID
```

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Query #1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
  FROM ALBUM
 WHERE ALBUM.NAME="Mooshoo Tribute"
```

Query #3

```
SELECT APPEARS.ARTIST_ID INTO TEMP2
  FROM APPEARS, TEMP1
 WHERE APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ARTIST_ID
```

Query #4

```
SELECT ARTIST.NAME
  FROM ARTIST, TEMP2
 WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Step #2: Substitute the values from
Query#1 → Query #3 → Query #4

Query #1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
  FROM ALBUM
 WHERE ALBUM.NAME="Mooshoo Tribute"
```

Query #3

```
SELECT APPEARS.ARTIST_ID INTO TEMP2
  FROM APPEARS, TEMP1
 WHERE APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ARTIST_ID
```

Query #4

```
SELECT ARTIST.NAME
  FROM ARTIST, TEMP2
 WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

ALBUM_ID
9999

```
SELECT APPEARS.ARTIST_ID
  FROM APPEARS
 WHERE APPEARS.ALBUM_ID=9999
 ORDER BY APPEARS.ARTIST_ID
```

Step #1: Decompose into single-value queries

Step #2: Substitute the values from
Query#1 → Query #3 → Query #4

Query #4

```
SELECT ARTIST.NAME
  FROM ARTIST, TEMP2
 WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

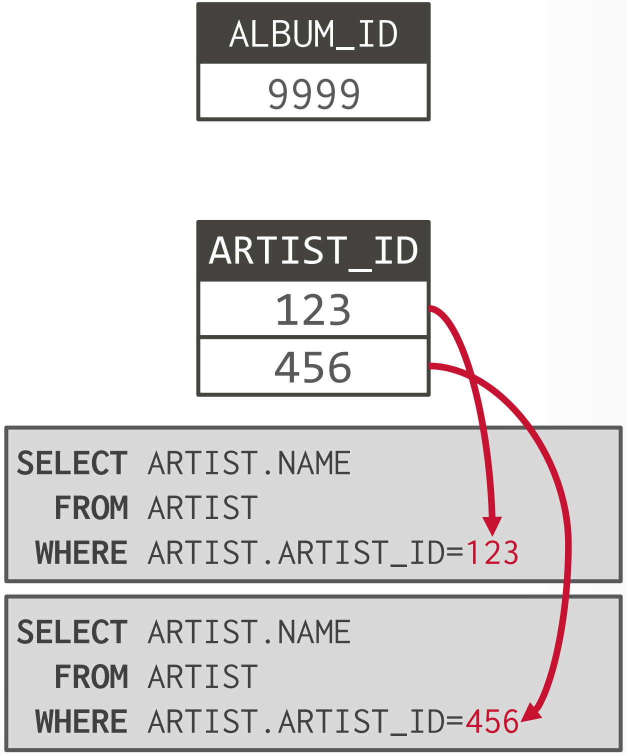
INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Step #2: Substitute the values from Query#1 → Query #3 → Query #4



ALBUM_ID
9999

ARTIST_ID
123
456

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=123
```

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=456
```

INGRES OPTIMIZER

Retrieve the names of people that appear on the DJ Mooshoo Tribute mixtape ordered by their artist id.

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

Step #1: Decompose into single-value queries

Step #2: Substitute the values from
Query#1 → Query #3 → Query #4

ALBUM_ID
9999

ARTIST_ID
123
456

NAME
O.D.B.

NAME
DJ Premier

HEURISTIC-BASED OPTIMIZATION

Advantages:

- → Easy to implement and debug.
- → Works reasonably well and is fast for simple queries.

Disadvantages:

- → Relies on magic constants that predict the efficacy of a planning decision.
- → Nearly impossible to generate good plans when operators have complex inter-dependencies.

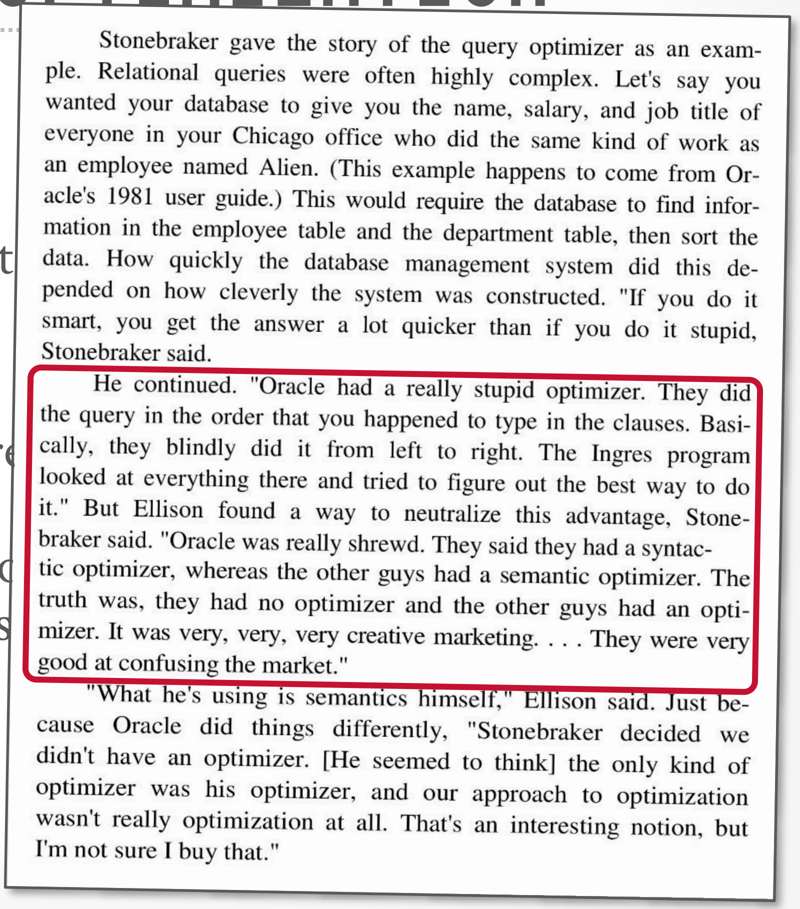
HEURISTIC-BASED OPTIMIZATION

Advantages:

- → Easy to implement and debug.
- → Works reasonably well and is fast

Disadvantages:

- → Relies on magic constants that pre
  planning decision.
- → Nearly impossible to generate go
  have complex inter-dependencies



Stonebraker gave the story of the query optimizer as an example. Relational queries were often highly complex. Let's say you wanted your database to give you the name, salary, and job title of everyone in your Chicago office who did the same kind of work as an employee named Alien. (This example happens to come from Oracle's 1981 user guide.) This would require the database to find information in the employee table and the department table, then sort the data. How quickly the database management system did this depended on how cleverly the system was constructed. "If you do it smart, you get the answer a lot quicker than if you do it stupid, Stonebraker said.

He continued. "Oracle had a really stupid optimizer. They did the query in the order that you happened to type in the clauses. Basically, they blindly did it from left to right. The Ingres program looked at everything there and tried to figure out the best way to do it." But Ellison found a way to neutralize this advantage, Stonebraker said. "Oracle was really shrewd. They said they had a syntactic optimizer, whereas the other guys had a semantic optimizer. The truth was, they had no optimizer and the other guys had an optimizer. It was very, very, very creative marketing. . . . They were very good at confusing the market."

"What he's using is semantics himself," Ellison said. Just because Oracle did things differently, "Stonebraker decided we didn't have an optimizer. [He seemed to think] the only kind of optimizer was his optimizer, and our approach to optimization wasn't really optimization at all. That's an interesting notion, but I'm not sure I buy that."

HISTORY OF QUERY OPTIMIZERS

Choice #1: Heuristics
→ INGRES (1970s), Oracle (until mid 1990s)

Choice #2: Heuristics + Cost-based Join Search
→ System R (1970s), early IBM DB2

Choice #3: Stratified Search
→ IBM STARBURST (late 1980s), now IBM DB2 + Oracle

Choice #4: Unified Search
→ Volcano/Cascades in 1990s, now MSSQL + Greenplum

Choice #5: Randomized Search
→ Academics in the 1980s, current Postgres

HEURISTICS + COST-BASED SEARCH

Selinger

First evaluate static rules to perform initial logical→logical optimizations.

Then enumerate plans using logical→physical transformations to find best plan according to a cost model.

Examples: System R, early IBM DB2, most open-source DBMSs today.

ACCESS PATH SELECTION IN A RELATIONAL DATABASE MANAGEMENT SYSTEM
SIGMOD 1979

PHYSICAL QUERY OPTIMIZATION

Transform a query plan's logical operators into physical operators.

- → Add more execution information
- → Select indexes / access paths
- → Choose operator implementations
- → Choose when to materialize (i.e., temp tables).

This stage must support cost model estimates.

SYSTEM R OPTIMIZER

Break query up 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

If a block accesses multiple relations, iteratively construct a join tree that minimizes the estimated amount of work to execute the plan.

ACCESS PATH SELECTION IN A RELATIONAL DATABASE MANAGEMENT SYSTEM
SIGMOD 1979

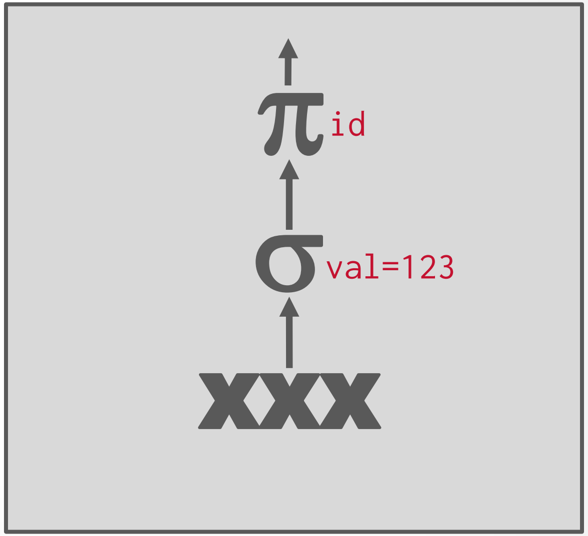
SYSTEM R – SINGLE RELATION QUERIES

Access path selection for a single relation query block is (relatively) easy because they are **sargable**.

Search
Argument
Able

Pick the best access method (sequential scan vs. index) using a simple cost model.

```
SELECT id
  FROM xxx
 WHERE val >= 123
   AND val <= 456;
```



SYSTEM R – SINGLE RELATION QUERIES

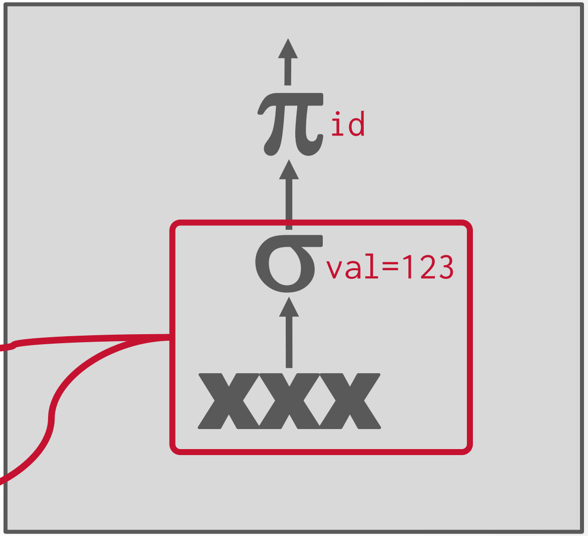
Access path selection for a single relation query block is (relatively) easy because they are **sargable**.

Search Argument Able

Pick the best access method (sequential scan vs. index) using a simple cost model.

```
SELECT id
  FROM xxx
 WHERE val >= 123
   AND val <= 456;
```

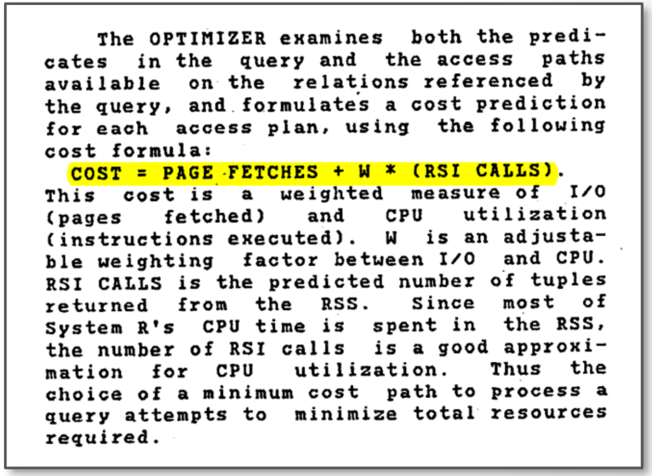
```
CREATE TABLE xxx (
  id INT PRIMARY KEY,
  val INT,
  ⋮
);
CREATE INDEX ON xxx (val);
```



SYSTEM R – COST MODEL

The cost of an access method is the summation of the expected number of I/Os ("page fetches") and weighted computational cost ("RSI calls").

→ Weight determines relative cost of I/O versus CPU.



The OPTIMIZER examines both the predicates in the query and the access paths available on the relations referenced by the query, and formulates a cost prediction for each access plan, using the following cost formula:

COST = PAGE FETCHES + W * (RSI CALLS).

This cost is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). W is an adjustable weighting factor between I/O and CPU. RSI CALLS is the predicted number of tuples returned from the RSS. Since most of System R's CPU time is spent in the RSS, the number of RSI calls is a good approximation for CPU utilization. Thus the choice of a minimum cost path to process a query attempts to minimize total resources required.

The DBMS estimates these values based on the **selectivity factor** of predicates derived from statistics for each relation and its indexes.

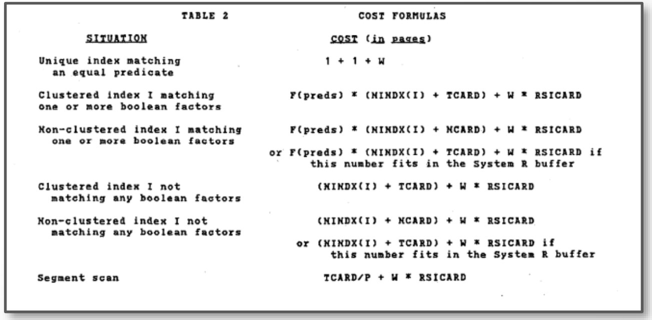


TABLE 2 COST FORMULAS

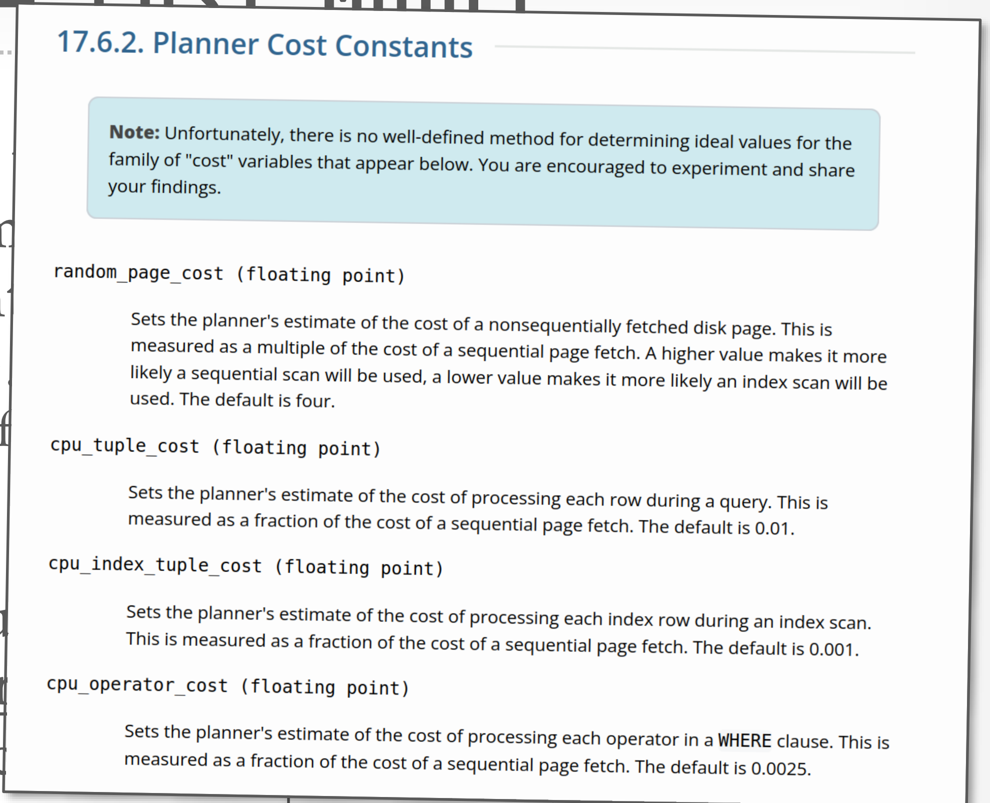
SITUATION	COST (in pages)
Unique index matching an equal predicate	1 + 1 + W
Clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I matching one or more boolean factors	F(preds) * (NINDX(I) + NCARD) + W * RSICARD or F(preds) * (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Clustered index I not matching any boolean factors	(NINDX(I) + TCARD) + W * RSICARD
Non-clustered index I not matching any boolean factors	(NINDX(I) + NCARD) + W * RSICARD or (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Segment scan	TCARD/P + W * RSICARD

SYSTEM R – COST MODEL

The cost of an access method is summation of the expected num I/Os ("page fetches") and weigh computational cost ("RSI calls").

→ Weight determines relative cost of versus CPU.

The DBMS estimates these valu based on the **selectivity factor** predicates derived from statistic each relation and its indexes.



17.6.2. Planner Cost Constants

Note: Unfortunately, there is no well-defined method for determining ideal values for the family of "cost" variables that appear below. You are encouraged to experiment and share your findings.

`random_page_cost (floating point)`

Sets the planner's estimate of the cost of a nonsequentially fetched disk page. This is measured as a multiple of the cost of a sequential page fetch. A higher value makes it more likely a sequential scan will be used, a lower value makes it more likely an index scan will be used. The default is four.

`cpu_tuple_cost (floating point)`

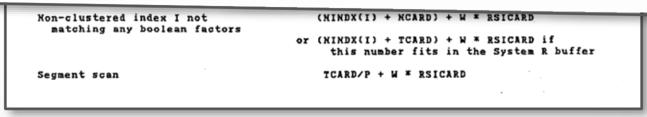
Sets the planner's estimate of the cost of processing each row during a query. This is measured as a fraction of the cost of a sequential page fetch. The default is 0.01.

`cpu_index_tuple_cost (floating point)`

Sets the planner's estimate of the cost of processing each index row during an index scan. This is measured as a fraction of the cost of a sequential page fetch. The default is 0.001.

`cpu_operator_cost (floating point)`

Sets the planner's estimate of the cost of processing each operator in a `WHERE` clause. This is measured as a fraction of the cost of a sequential page fetch. The default is 0.0025.



Non-clustered index I not matching any boolean factors	(NINDX(I) + NCARD) + W * RSICARD or (NINDX(I) + TCARD) + W * RSICARD if this number fits in the System R buffer
Segment scan	TCARD/P + W * RSICARD

SYSTEM R – SELECTIVITY FACTOR

A selectivity factor of a predicate is the expected faction of tuples that will satisfy that predicate.

The optimizer uses formulas to approximate each predicate's selectivity factor.

→ Make several assumptions about distribution of values in columns to simplify the problem.

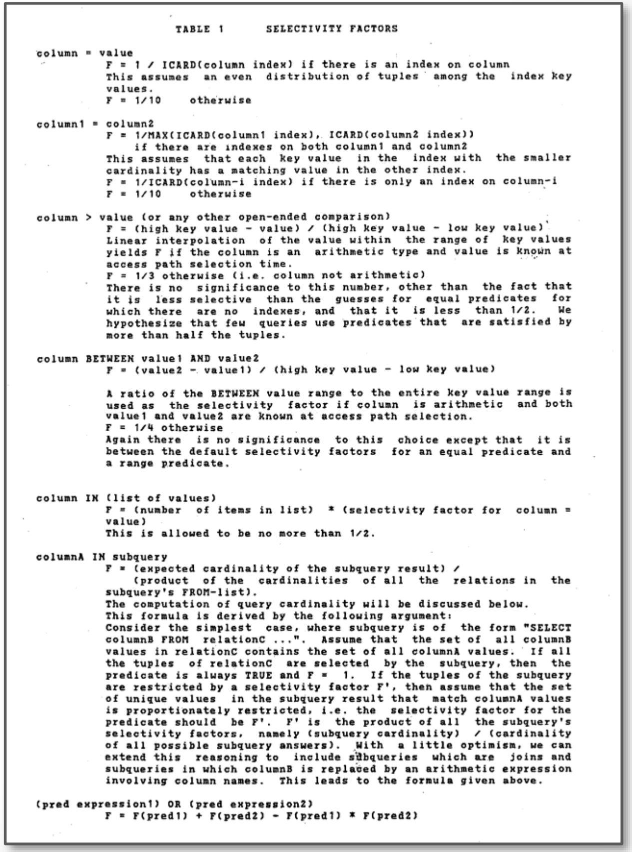


TABLE 1 SELECTIVITY FACTORS

column = value

F = 1 / ICARD(column index) if there is an index on column

This assumes an even distribution of tuples among the index key values.

F = 1/10 otherwise

column1 = column2

F = 1/MAX(ICARD(column1 index), ICARD(column2 index)) if there are indexes on both column1 and column2

This assumes that each key value in the index with the smaller cardinality has a matching value in the other index.

F = 1/ICARD(column-i index) if there is only an index on column-i

F = 1/10 otherwise

column > value (or any other open-ended comparison)

F = (high key value - value) / (high key value - low key value)

Linear interpolation of the value within the range of key values yields F if the column is an arithmetic type and value is known at access path selection time.

F = 1/3 otherwise (i.e. column not arithmetic)

There is no significance to this number, other than the fact that it is less selective than the guesses for equal predicates for which there are no indexes, and that it is less than 1/2. We hypothesize that few queries use predicates that are satisfied by more than half the tuples.

column BETWEEN value1 AND value2

F = (value2 - value1) / (high key value - low key value)

A ratio of the BETWEEN value range to the entire key value range is used as the selectivity factor if column is arithmetic and both value1 and value2 are known at access path selection.

F = 1/4 otherwise

Again there is no significance to this choice except that it is between the default selectivity factors for an equal predicate and a range predicate.

column IN (list of values)

F = (number of items in list) * (selectivity factor for column = value)

This is allowed to be no more than 1/2.

columnA IN subquery

F = (expected cardinality of the subquery result) / (product of the cardinalities of all the relations in the subquery's FROM-list).

The computation of query cardinality will be discussed below.

This formula is derived by the following argument:

Consider the simplest case, where subquery is of the form "SELECT columnB FROM relationC ...". Assume that the set of all columnB values in relationC contains the set of all columnA values. If all the tuples of relationC are selected by the subquery, then the predicate is always TRUE and F = 1. If the tuples of the subquery are restricted by a selectivity factor F', then assume that the set of unique values in the subquery result that match columnA values is proportionately restricted, i.e. the selectivity factor for the predicate should be F'. F' is the product of all the subquery's selectivity factors, namely (subquery cardinality) / (cardinality of all possible subquery answers). With a little optimism, we can extend this reasoning to include subqueries which are joins and subqueries in which columnB is replaced by an arithmetic expression involving column names. This leads to the formula given above.

(pred expression1) OR (pred expression2)

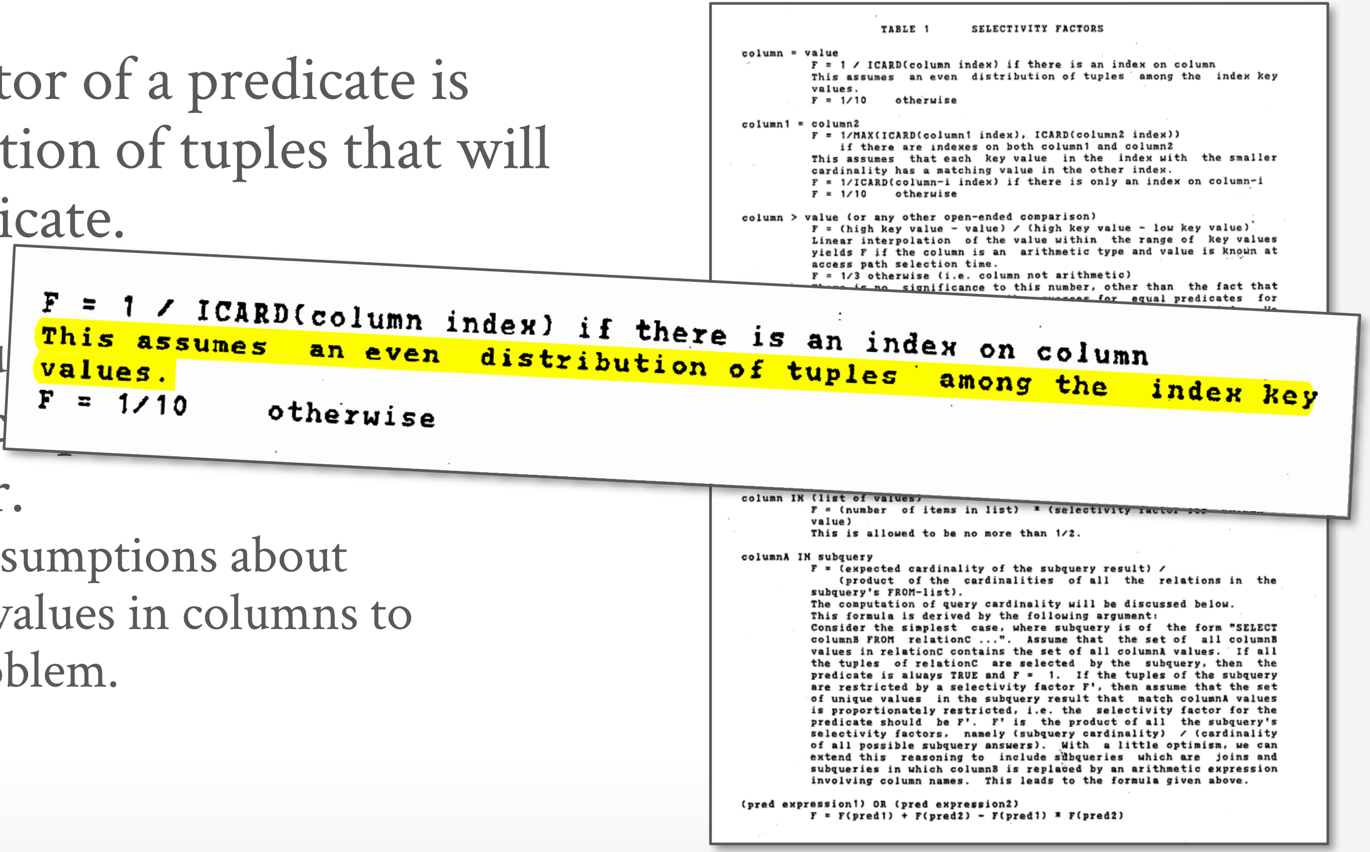
F = F(pred1) + F(pred2) - F(pred1) * F(pred2)

SYSTEM R – SELECTIVITY FACTOR

A selectivity factor of a predicate is the expected faction of tuples that will satisfy that predicate.

The optimizer u
approximate eac
selectivity factor.

→ Make several assumptions about distribution of values in columns to simplify the problem.



```
F = 1 / ICARD(column index) if there is an index on column
This assumes  an even  distribution of tuples  among the  index key
values.
F = 1/10    otherwise
```

```
                         TABLE 1      SELECTIVITY FACTORS

column = value
          F = 1 / ICARD(column index) if there is an index on column
          This assumes  an even  distribution of tuples  among the  index key
          values.
          F = 1/10    otherwise

column1 = column2
          F = 1/MAX(ICARD(column1 index), ICARD(column2 index))
              if there are indexes on both column1 and column2
          This assumes  that each  key value  in the  index with  the smaller
          cardinality has a matching value in the other index.
          F = 1/ICARD(column-i index) if there is only an index on column-i
          F = 1/10    otherwise

column > value (or any other open-ended comparison)
          F = (high key value - value) / (high key value - low key value)
          Linear interpolation  of the value within  the range of  key values
          yields F if the column is an  arithmetic type and value is known at
          access path selection time.
          F = 1/3 otherwise (i.e. column not arithmetic)
          There is no  significance to this number, other than  the fact that
                                               for  equal predicates  for

column IN (list of values)
          F = (number  of items in list)  * (selectivity
          value)
          This is allowed to be no more than 1/2.

columnA IN subquery
          F = (expected cardinality of the subquery result) /
              (product  of the  cardinalities  of all  the  relations in  the
          subquery's FROM-list).
          The computation of query cardinality will be discussed below.
          This formula is derived by the following argument:
          Consider the simplest  case, where subquery is of  the form "SELECT
          columnB FROM  relationC ...".  Assume that  the set of  all columnB
          values in relationC contains the set of all columnA values.  If all
          the tuples  of relationC  are selected  by the  subquery, then  the
          predicate is always TRUE and F =  1.  If the tuples of the subquery
          are restricted by a selectivity factor F', then assume that the set
          of unique values  in the subquery result that  match columnA values
          is proportionately restricted, i.e. the  selectivity factor for the
          predicate should  be F'.  F' is  the product of all  the subquery's
          selectivity factors,  namely (subquery cardinality)  / (cardinality
          of all possible subquery answers).  With  a little optimism, we can
          extend this  reasoning to  include subqueries  which are  joins and
          subqueries in which columnB is replaced by an arithmetic expression
          involving column names.  This leads to the formula given above.

(pred expression1) OR (pred expression2)
          F = F(pred1) + F(pred2) - F(pred1) * F(pred2)
```

SYSTEM R – INTERESTING ORDERS

For each query block, the DBMS extracts the required ("interesting") ordering of its output.

→ Examples: **ORDER BY**, **GROUP BY**

It then compares the best access method that orders the data versus the best unordered access method + sort operator.

If there is no required ordering, then the DBMS selects the access method with the lowest cost.

SYSTEM R – MULTIPLE RELATIONS

If a query block accesses multiple relations, then the DBMS must determine the best ordering to join those relations.

→ Also identify interesting orders based on join predicates.

Leverage domain knowledge to reduce the search complexity by delaying or discarding plan choices.

→ Example: Only consider left-deep trees.

Join costs are estimated based on the number of tuples processed in outer/inner relations.

SYSTEM R – MULTIPLE RELATIONS

Step #1: Choose the best access paths to each relation.

Step #2: Enumerate all join orderings for 1-relation plans using best access path found in Step #1.

Step #3: For each subsequent pass, the algorithm determines the best way to join the result of an n – 1 relation plan as the outer relation to the nth relation.

Algorithm does <u>not</u> need to remember anything at a previous level explicitly as it's being remembered implicitly by the nature of a bottom-up approach.

SYSTEM R – MULTIPLE RELATIONS

Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

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

ARTIST: Sequential Scan
APPEARS: Sequential Scan
ALBUM: Index Look-up on **NAME**

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

SYSTEM R – MULTIPLE RELATIONS

Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

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

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

Step #2: Enumerate all possible join orderings for tables

ARTIST: Sequential Scan
APPEARS: Sequential Scan
ALBUM: Index Look-up on **NAME**

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

SYSTEM R – MULTIPLE RELATIONS

Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

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

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

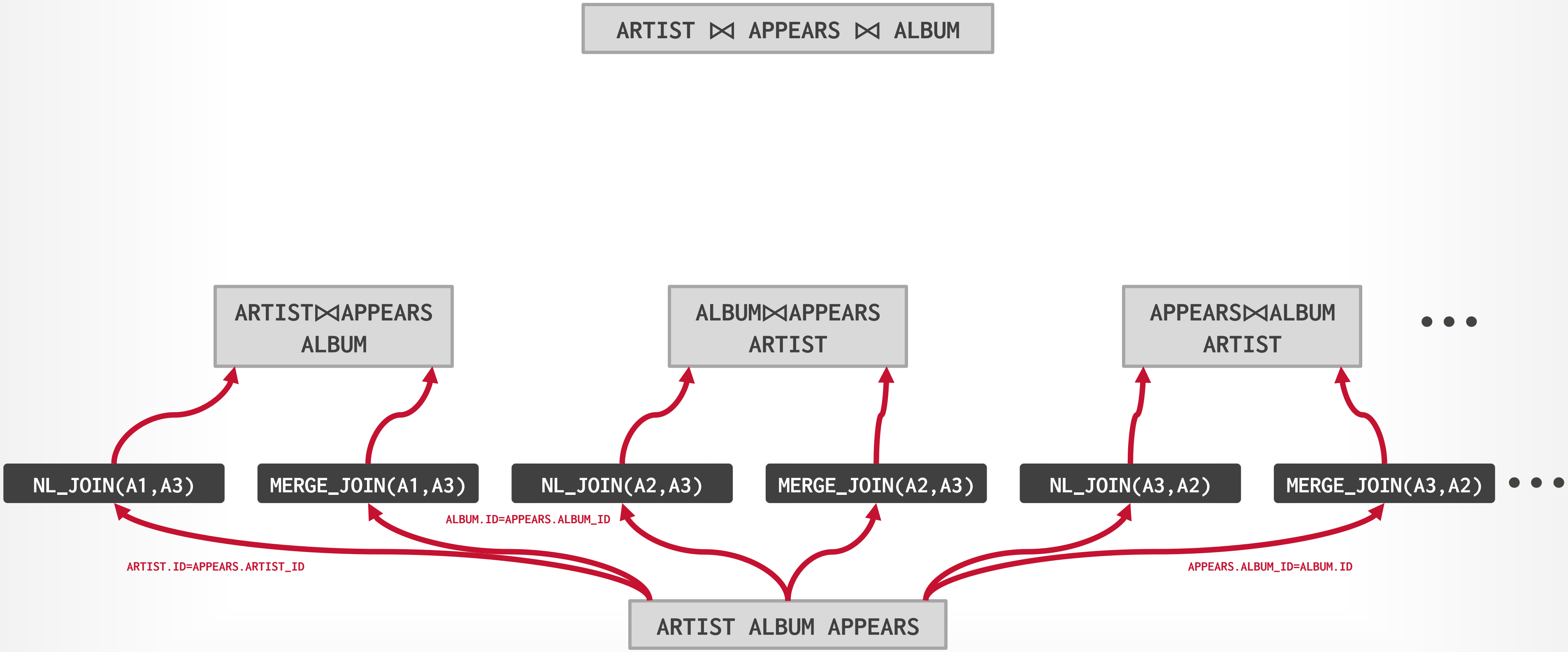
ARTIST: Sequential Scan
APPEARS: Sequential Scan
ALBUM: Index Look-up on **NAME**

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

☐ *Logical Op*

■ *Physical Op*

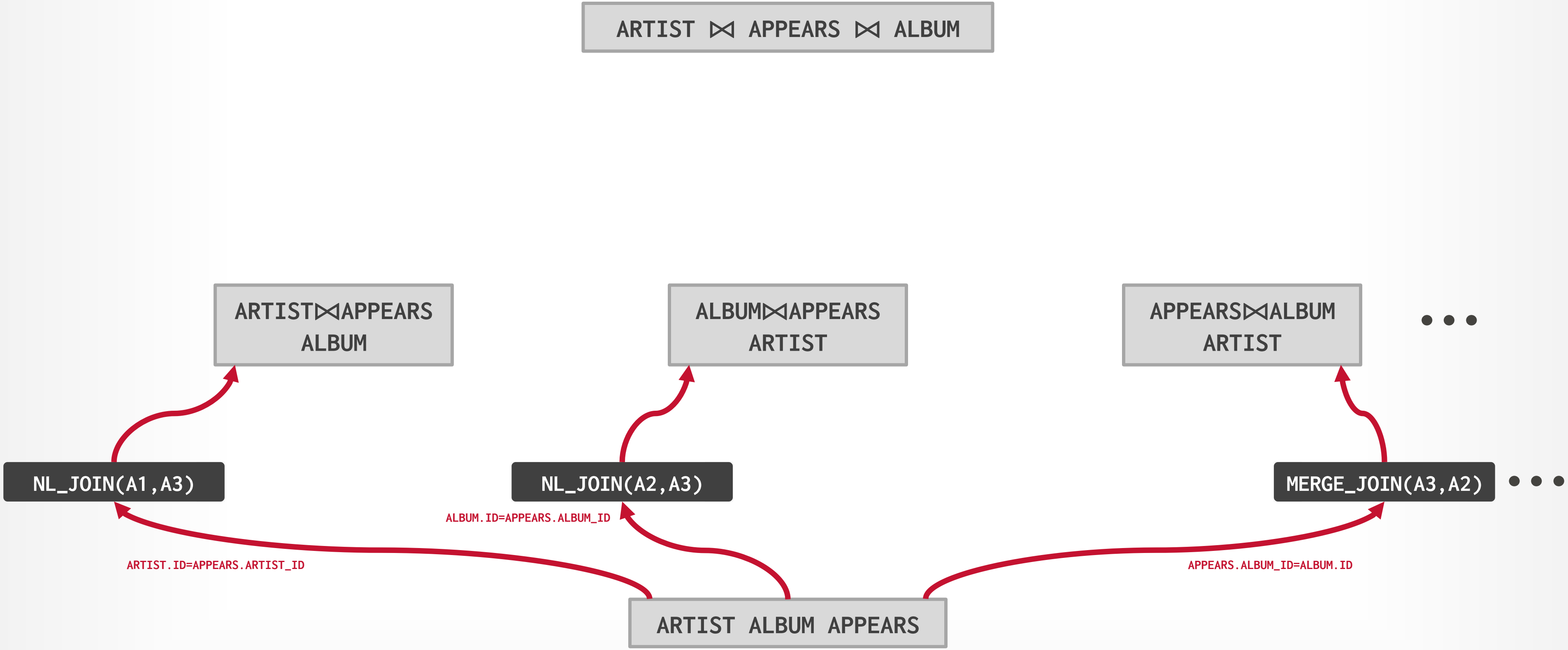
SYSTEM R – BOTTOM-UP SEARCH



☐ *Logical Op*

■ *Physical Op*

SYSTEM R – BOTTOM-UP SEARCH



ARTIST ⋈ APPEARS ⋈ ALBUM

ARTIST⋈APPEARS
ALBUM

ALBUM⋈APPEARS
ARTIST

APPEARS⋈ALBUM
ARTIST

• • •

NL_JOIN(A1,A3)

NL_JOIN(A2,A3)

MERGE_JOIN(A3,A2)

• • •

ALBUM.ID=APPEARS.ALBUM_ID

ARTIST.ID=APPEARS.ARTIST_ID

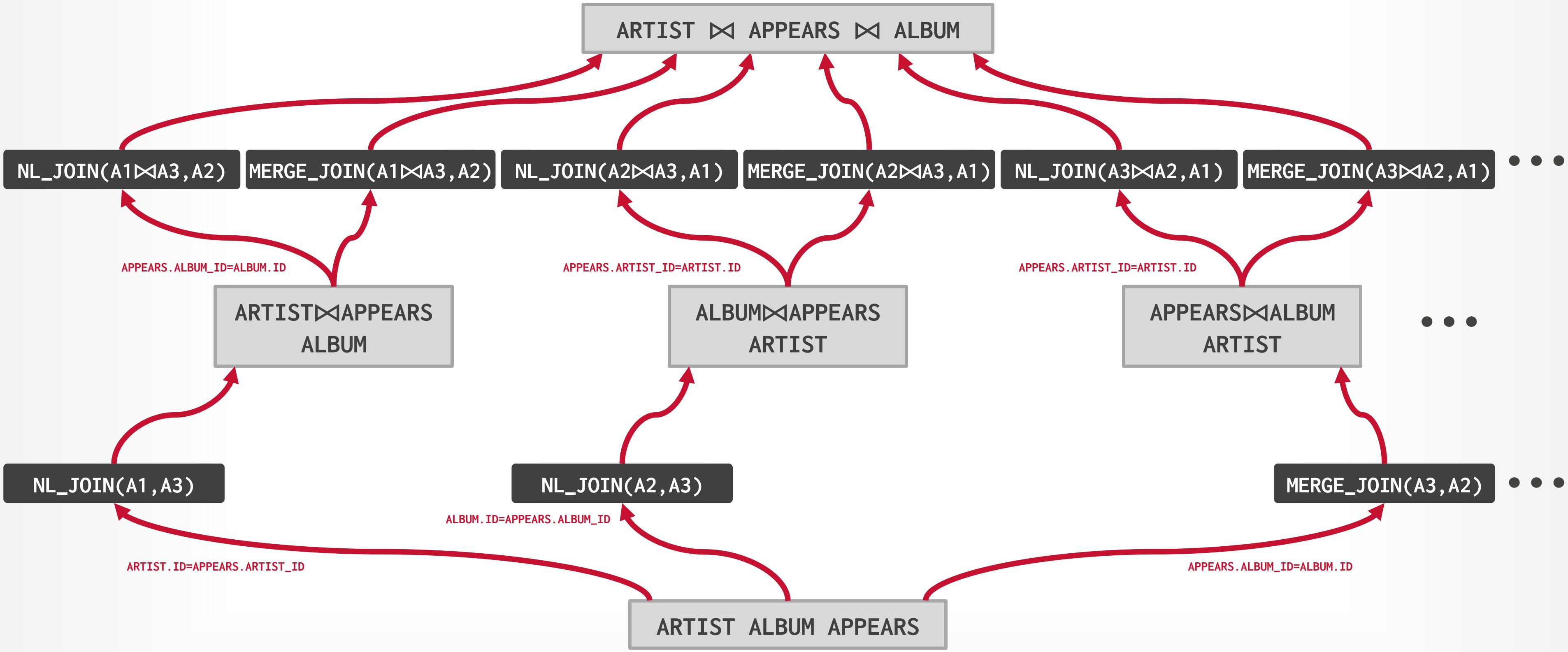
APPEARS.ALBUM_ID=ALBUM.ID

ARTIST ALBUM APPEARS

☐ *Logical Op*

■ *Physical Op*

SYSTEM R – BOTTOM-UP SEARCH



ARTIST ⋈ APPEARS ⋈ ALBUM

NL_JOIN(A1⋈A3,A2)
MERGE_JOIN(A1⋈A3,A2)
NL_JOIN(A2⋈A3,A1)
MERGE_JOIN(A2⋈A3,A1)
NL_JOIN(A3⋈A2,A1)
MERGE_JOIN(A3⋈A2,A1)
• • •

APPEARS.ALBUM_ID=ALBUM.ID
APPEARS.ARTIST_ID=ARTIST.ID
APPEARS.ARTIST_ID=ARTIST.ID

ARTIST⋈APPEARS
ALBUM

ALBUM⋈APPEARS
ARTIST

APPEARS⋈ALBUM
ARTIST

• • •

NL_JOIN(A1,A3)
NL_JOIN(A2,A3)
MERGE_JOIN(A3,A2)
• • •

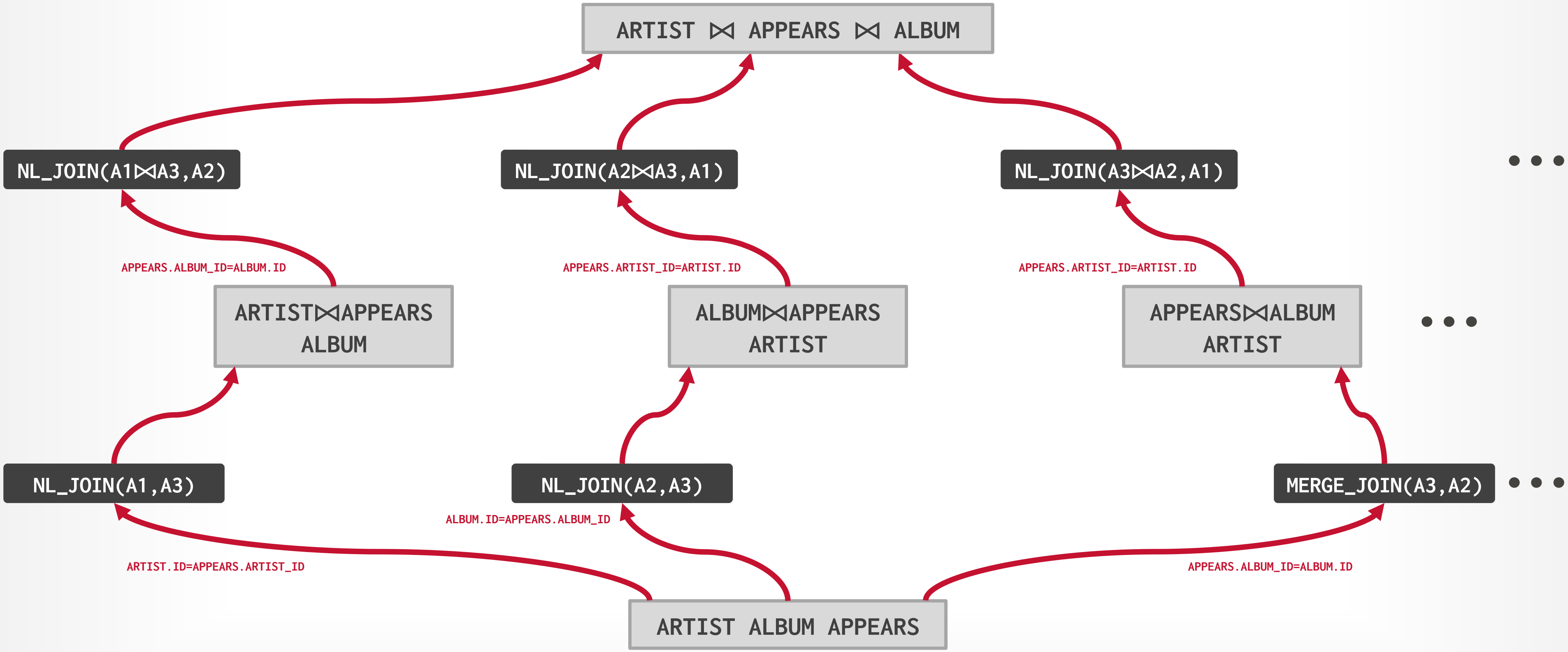
ALBUM.ID=APPEARS.ALBUM_ID
ARTIST.ID=APPEARS.ARTIST_ID
APPEARS.ALBUM_ID=ALBUM.ID

ARTIST ALBUM APPEARS

☐ *Logical Op*

■ *Physical Op*

SYSTEM R – BOTTOM-UP SEARCH



ARTIST ⋈ APPEARS ⋈ ALBUM

NL_JOIN(A1⋈A3,A2)

NL_JOIN(A2⋈A3,A1)

NL_JOIN(A3⋈A2,A1)

• • •

APPEARS.ALBUM_ID=ALBUM.ID

APPEARS.ARTIST_ID=ARTIST.ID

APPEARS.ARTIST_ID=ARTIST.ID

ARTIST⋈APPEARS
ALBUM

ALBUM⋈APPEARS
ARTIST

APPEARS⋈ALBUM
ARTIST

• • •

NL_JOIN(A1,A3)

NL_JOIN(A2,A3)

MERGE_JOIN(A3,A2)

• • •

ALBUM.ID=APPEARS.ALBUM_ID

ARTIST.ID=APPEARS.ARTIST_ID

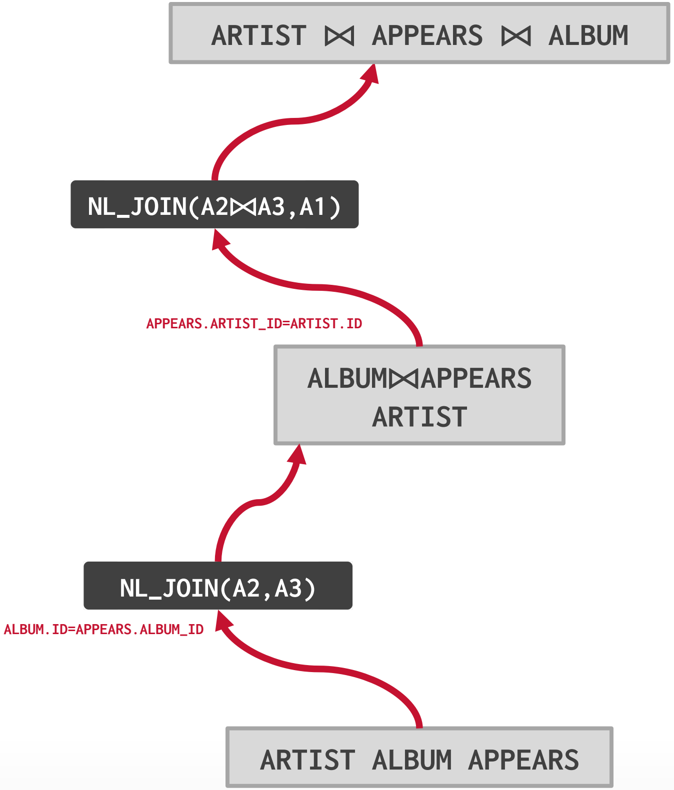
APPEARS.ALBUM_ID=ALBUM.ID

ARTIST ALBUM APPEARS

☐ *Logical Op*

■ *Physical Op*

SYSTEM R – BOTTOM-UP SEARCH



ARTIST ⋈ APPEARS ⋈ ALBUM

NL_JOIN(A2⋈A3,A1)

APPEARS.ARTIST_ID=ARTIST.ID

ALBUM⋈APPEARS
ARTIST

NL_JOIN(A2,A3)

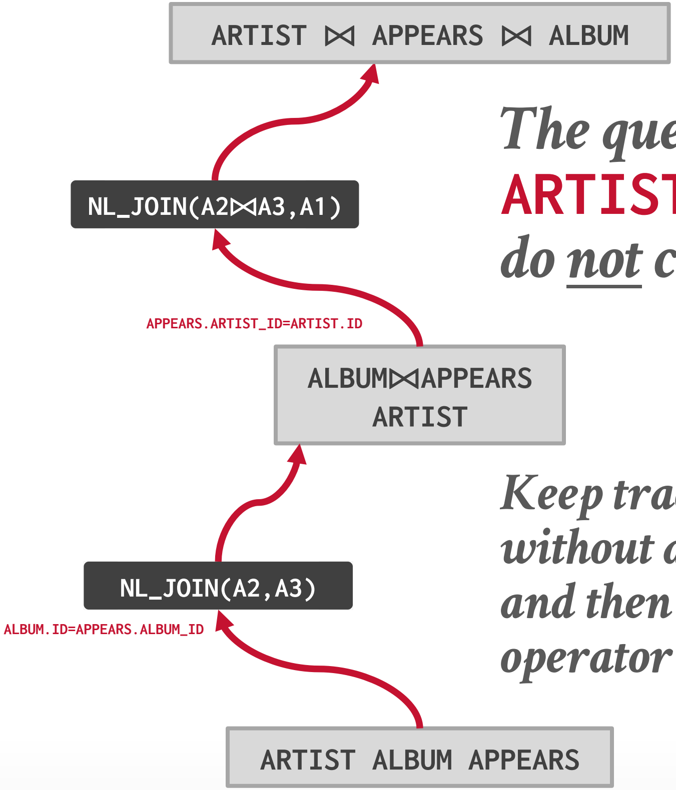
ALBUM.ID=APPEARS.ALBUM_ID

ARTIST ALBUM APPEARS

☐ *Logical Op*

■ *Physical Op*

SYSTEM R – BOTTOM-UP SEARCH



The query has **ORDER BY** *on* **ARTIST.ID** *but the logical plans do not contain sorting properties.*

Keep track of best plans with and without data in proper physical form, and then check whether tacking on a sort operator at the end is better.

PLAN ENUMERATION

Approach #1: Generative / Bottom-Up

→ Start with nothing and then iteratively assemble and add building blocks to generate a query plan.

→ **Examples:** System R, Starburst

Approach #2: Transformation / Top-Down

→ Start with the outcome that the query wants and then transform it to equivalent alternative sub-plans to find the optimal plan that gets to that goal.

→ **Examples**: Volcano, Cascades

ON THE CORRECT AND COMPLETE ENUMERATION OF THE CORE SEARCH SPACE
SIGMOD 2013

SYSTEM R – NESTED QUERIES

The DBMS treats nested queries as separate queries.

The optimizer executes an inner query before it begins planning an outer query so that it can substitute values into it or materialize its results to a temporary table.

We will have an entire lecture on rewriting nested queries into joins…

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

SYSTEM R – NESTED QUERIES

The DBMS treats nested queries as separate queries.

The optimizer executes an inner query before it begins planning an outer query so that it can substitute values into it or materialize its results to a temporary table.

We will have an entire lecture on rewriting nested queries into joins…

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

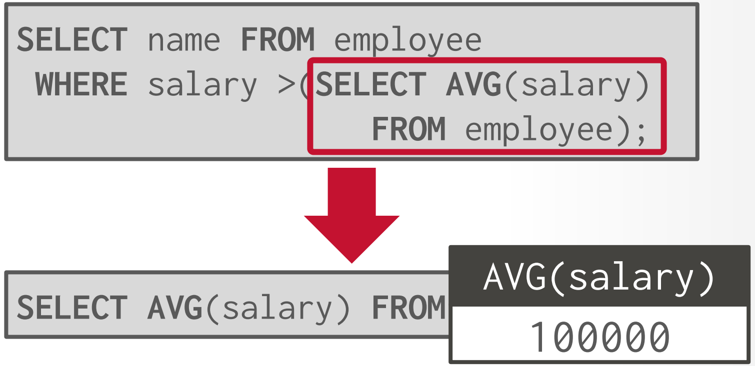
```
SELECT AVG(salary) FROM employee;
```

SYSTEM R – NESTED QUERIES

The DBMS treats nested queries as separate queries.

The optimizer executes an inner query before it begins planning an outer query so that it can substitute values into it or materialize its results to a temporary table.

We will have an entire lecture on rewriting nested queries into joins…



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

```
SELECT AVG(salary) FROM
```

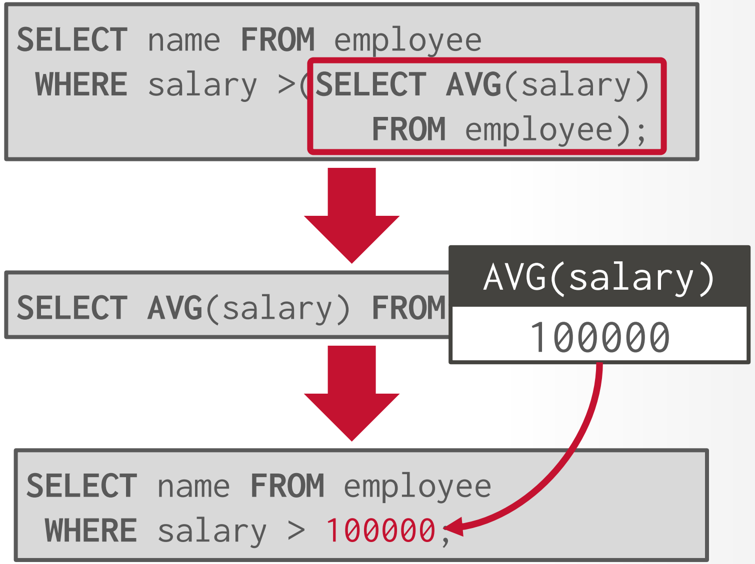
AVG(salary)
100000

SYSTEM R – NESTED QUERIES

The DBMS treats nested queries as separate queries.

The optimizer executes an inner query before it begins planning an outer query so that it can substitute values into it or materialize its results to a temporary table.

We will have an entire lecture on rewriting nested queries into joins…



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

```
SELECT AVG(salary) FROM
```

AVG(salary)
100000

```
SELECT name FROM employee
 WHERE salary > 100000;
```

HEURISTICS + COST-BASED SEARCH

Advantages:

→ Usually finds a reasonable plan without having to perform an exhaustive search.

Disadvantages:

→ All the same problems as the heuristic-only approach.
→ Left-deep join trees are not always optimal.
→ Must take in consideration the physical properties of data in the cost model (e.g., sort order).

PARTING THOUGHTS

Although the System R paper is over 40 years old, it still provides a reasonable foundation for building a modern query optimizer.

- → For two relation queries, it will find the optimal join ordering quickly.

But many of its simplifying assumptions in its cost estimates and selectivity factor cause problems in the real-world.

NEXT CLASS

IBM Starburst Optimizer

HISTORY OF QUERY OPTIMIZERS

Choice #1: Heuristics
→ INGRES (1970s), Oracle (until mid 1990s)

Choice #2: Heuristics + Cost-based Join Search
→ System R (1970s), early IBM DB2

Choice #3: Stratified Search
→ IBM STARBURST (late 1980s), now IBM DB2 + Oracle

Choice #4: Unified Search
→ Volcano/Cascades in 1990s, now MSSQL + Greenplum

Choice #5: Randomized Search
→ Academics in the 1980s, current Postgres