Query Processing and Optimization

Juliana Freire

Some slides adapted from L. Delcambre, R. Ramakrishnan, G. Lindstrom, J. Ullman and Silberschatz, Korth and Sudarshan, H. Garcia-Molina

Numbers Everyone Should Know*

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA → Netherlands → CA	150,000,000 ns

* According to Jeff Dean (LADIS 2009 keynote)

Indexes and Efficient Access to Data

- Data is transferred between disk and main memory in blocks
- Goal: Minimize the number of blocks read/written from disk
- Data are stored in a fixed structure
- A fixed structure is unlikely to be the best for all possible access patterns
  - Good for:
    List all accounts in the Downtown branch
  - What about:
    List all accounts with balance = 350

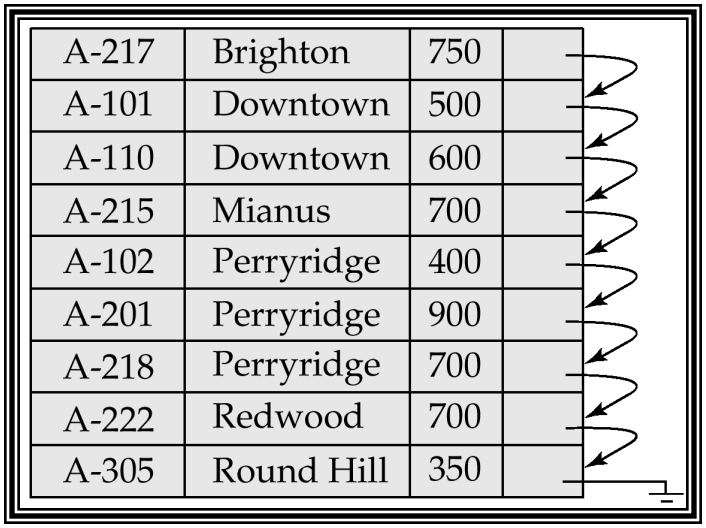
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Indexes and Efficient Access to Data

Q1: List all accounts with balance = 350

Requires all tuples to be examined – very inefficient if table is large

- An index on *balance* makes it *efficient* to find tuples with a specific balance
  - Only accounts with balance = 350 are examined
  - Fewer blocks retrieved from disk!



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Example

R(A,B,C) S(C,D,E)

Select B,D

From R,S

Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C

Select B,D

From R,S

Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

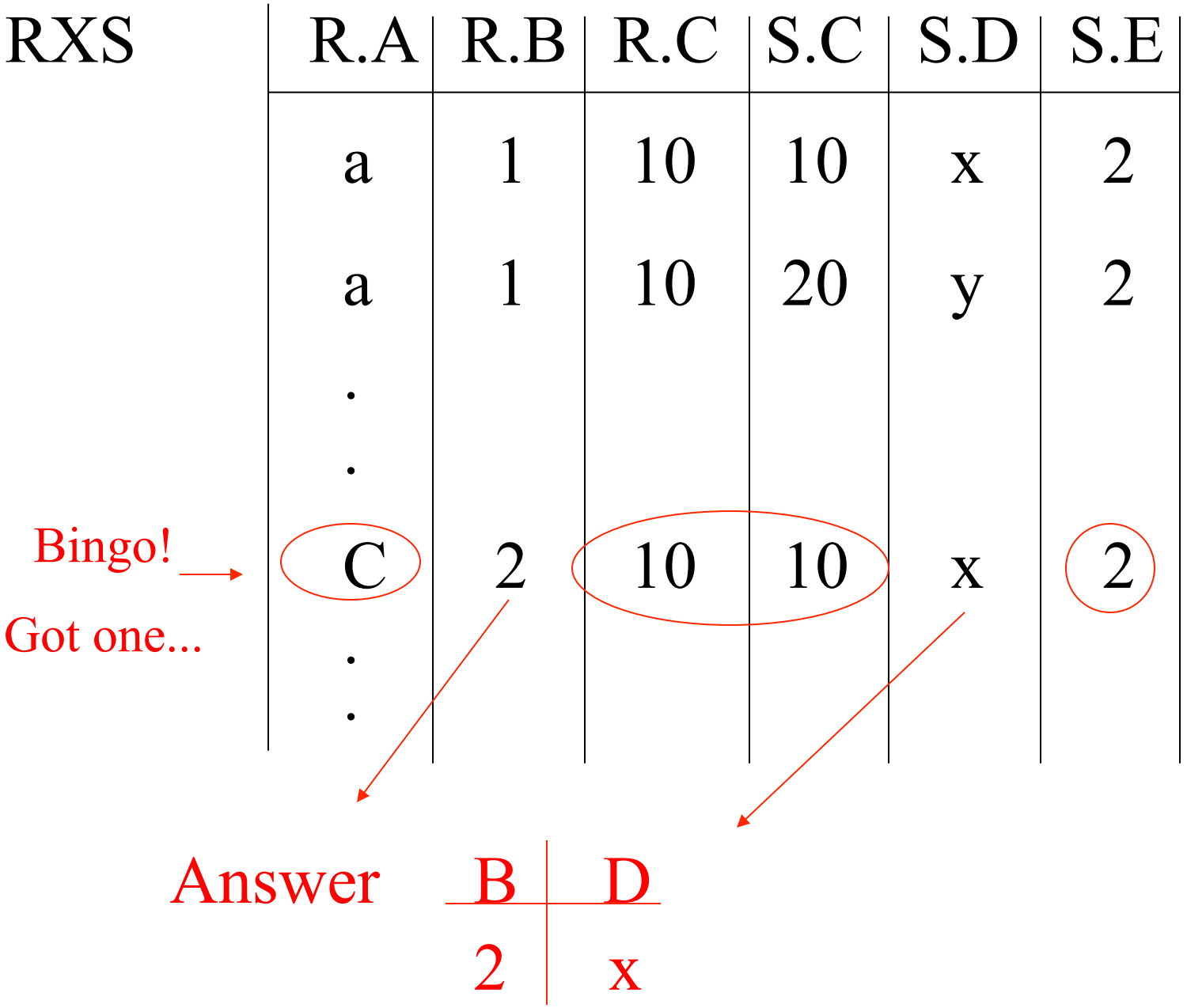
Answer

B	D
2	x

- How do we execute query?

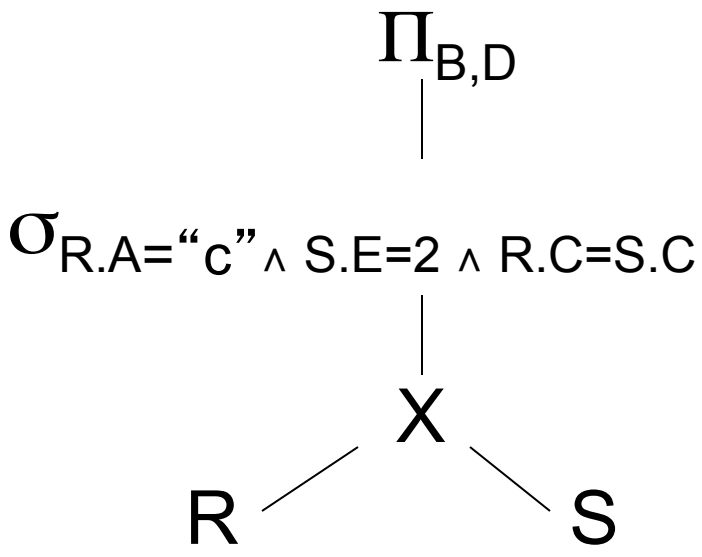
One idea

- Do Cartesian product
- Select tuples
- Do projection


RXS
R.A
R.B
R.C
S.C
S.D
S.E
a 1 10 10 x 2
a 1 10 20 y 2
C 2 10 10 x 2
Bingo!
Got one...
Answer
B
D
2
x

Relational Algebra - can be used to describe plans...

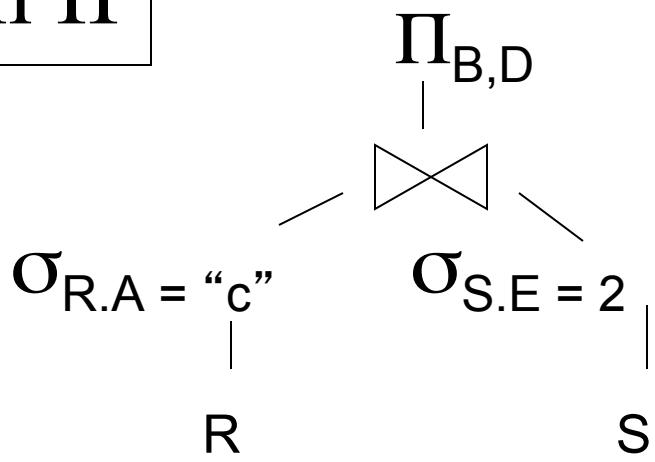
Ex: Plan I

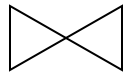


OR: $\Pi_{B,D}\,[\,\sigma_{R.A=\text{“c”} \wedge S.E=2 \wedge R.C = S.C}\,(RXS)]$

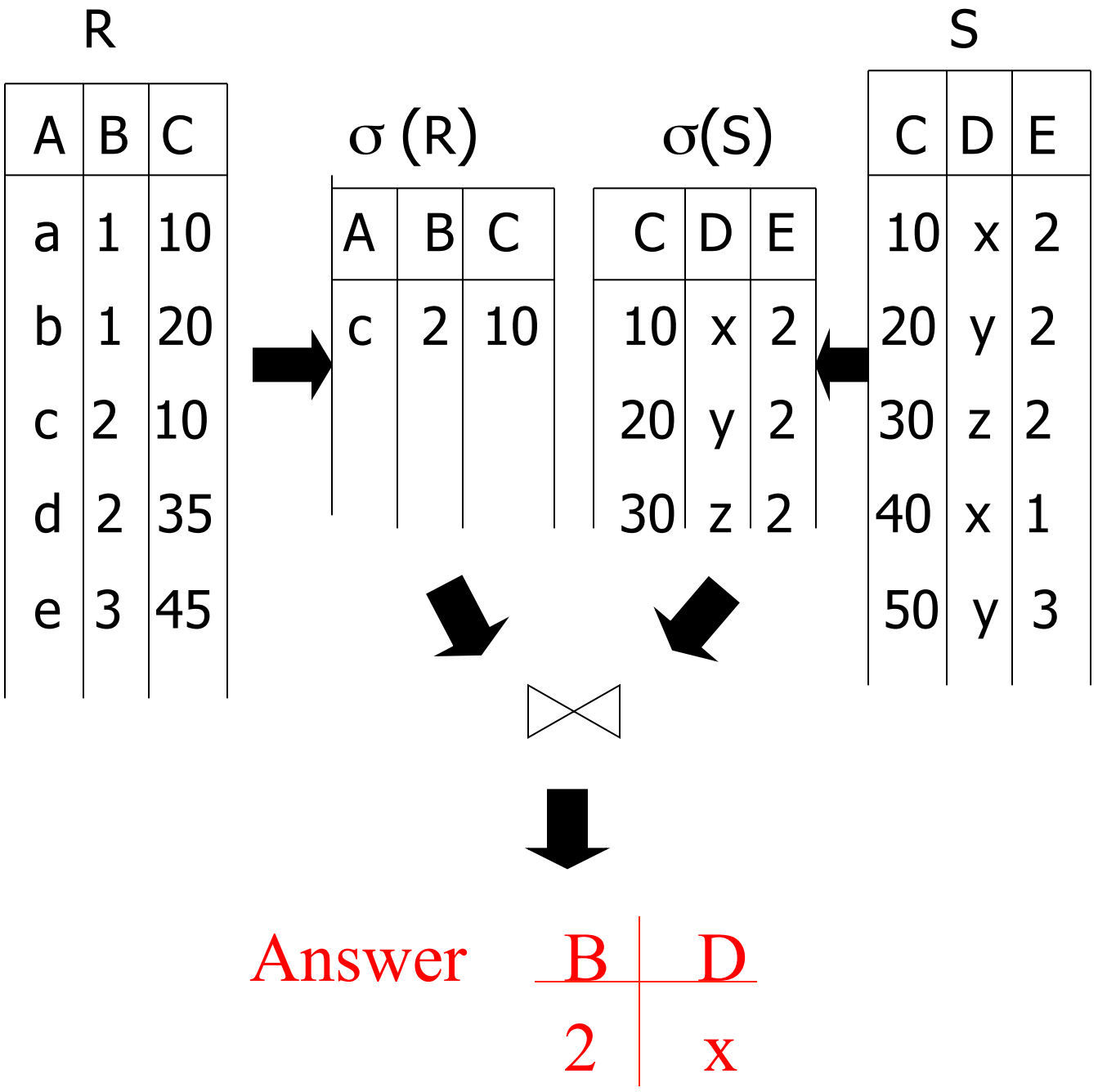
Another idea:

Plan II





natural join


R
A B C
a 1 10
b 1 20
c 2 10
d 2 35
e 3 45
σ (R)
A B C
c 2 10
σ(S)
C D E
10 x 2
20 y 2
30 z 2
S
C D E
10 x 2
20 y 2
30 z 2
40 x 1
50 y 3
⋈
Answer
B D
2 x

Plan III

Use R.A and S.C Indexes

(1) Use R.A index to select R tuples with R.A = “c”

(2) For each R.C value found, use S.C index to find matching tuples

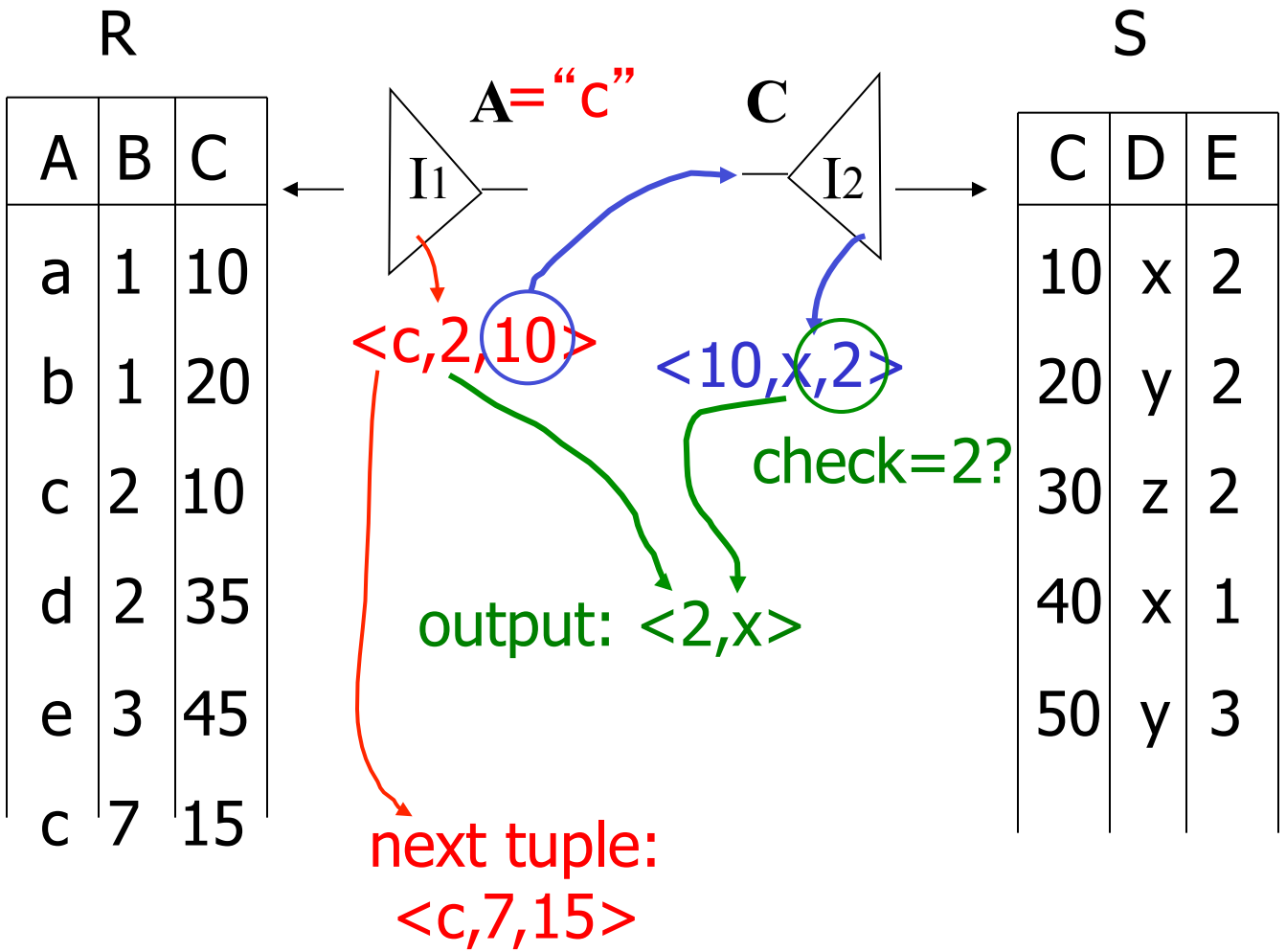
(3) Eliminate S tuples S.E ≠ 2

(4) Join matching R,S tuples, project B,D attributes and place in result

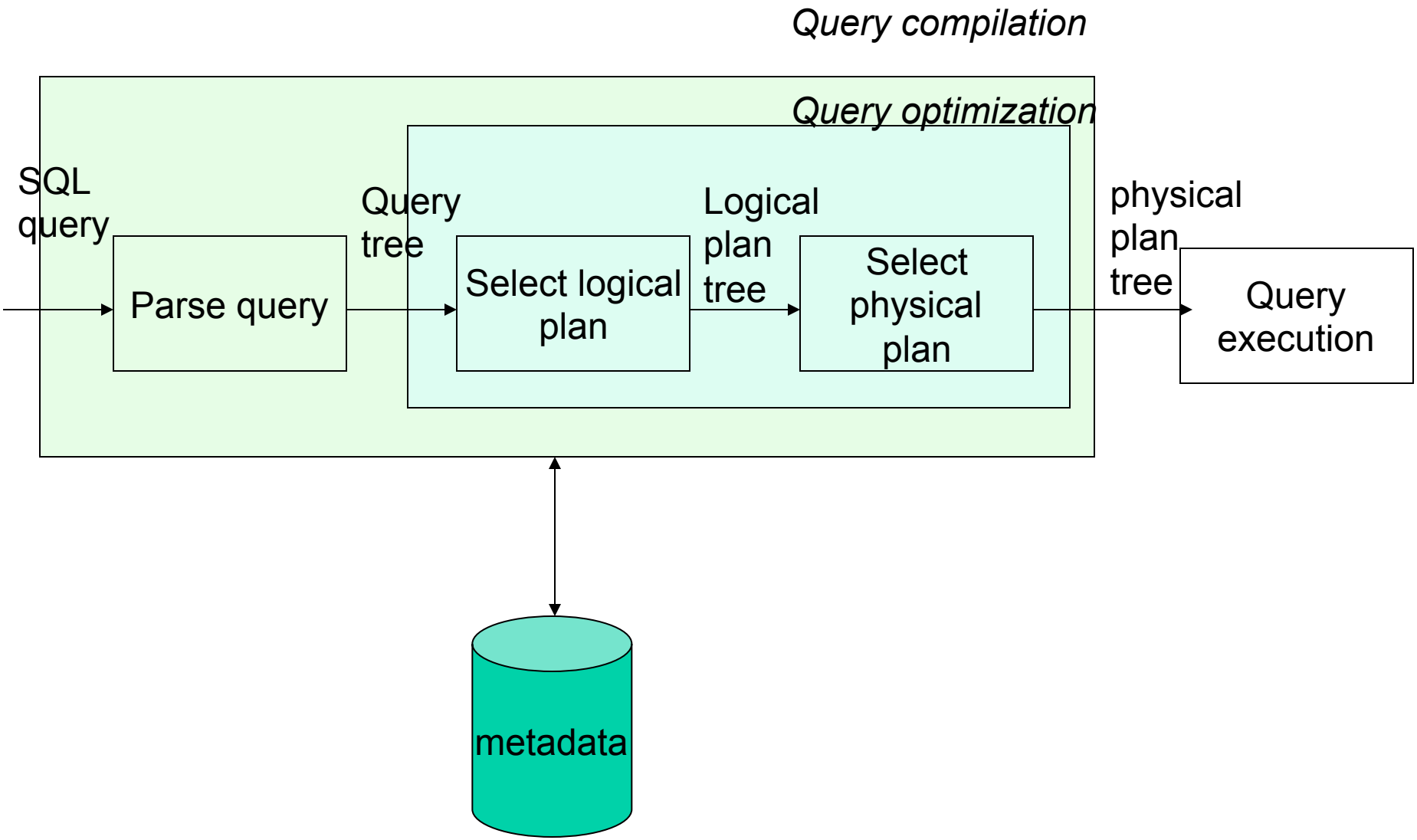
Select B,D

From R,S

Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C



Overview


Query compilation
Query optimization
SQL query
Parse query
Query tree
Select logical plan
Logical plan tree
Select physical plan
physical plan tree
Query execution
metadata

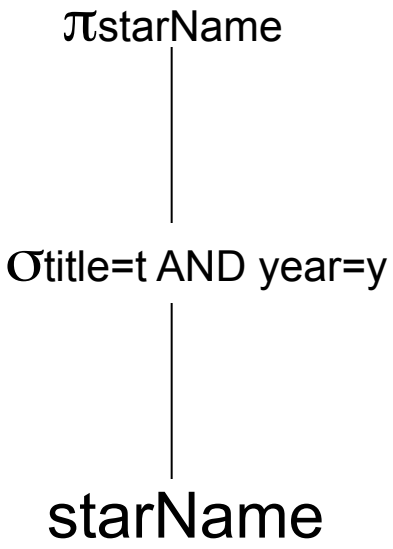
Example: Parsing

StarsIn(title, year, starName)

```
SELECT name
FROM StarsIn
WHERE title = t AND year = y;
```

Parse error!
No name attribute in starsIn relation

```
SELECT starName
FROM StarsIn
WHERE title = t AND year = y;
```

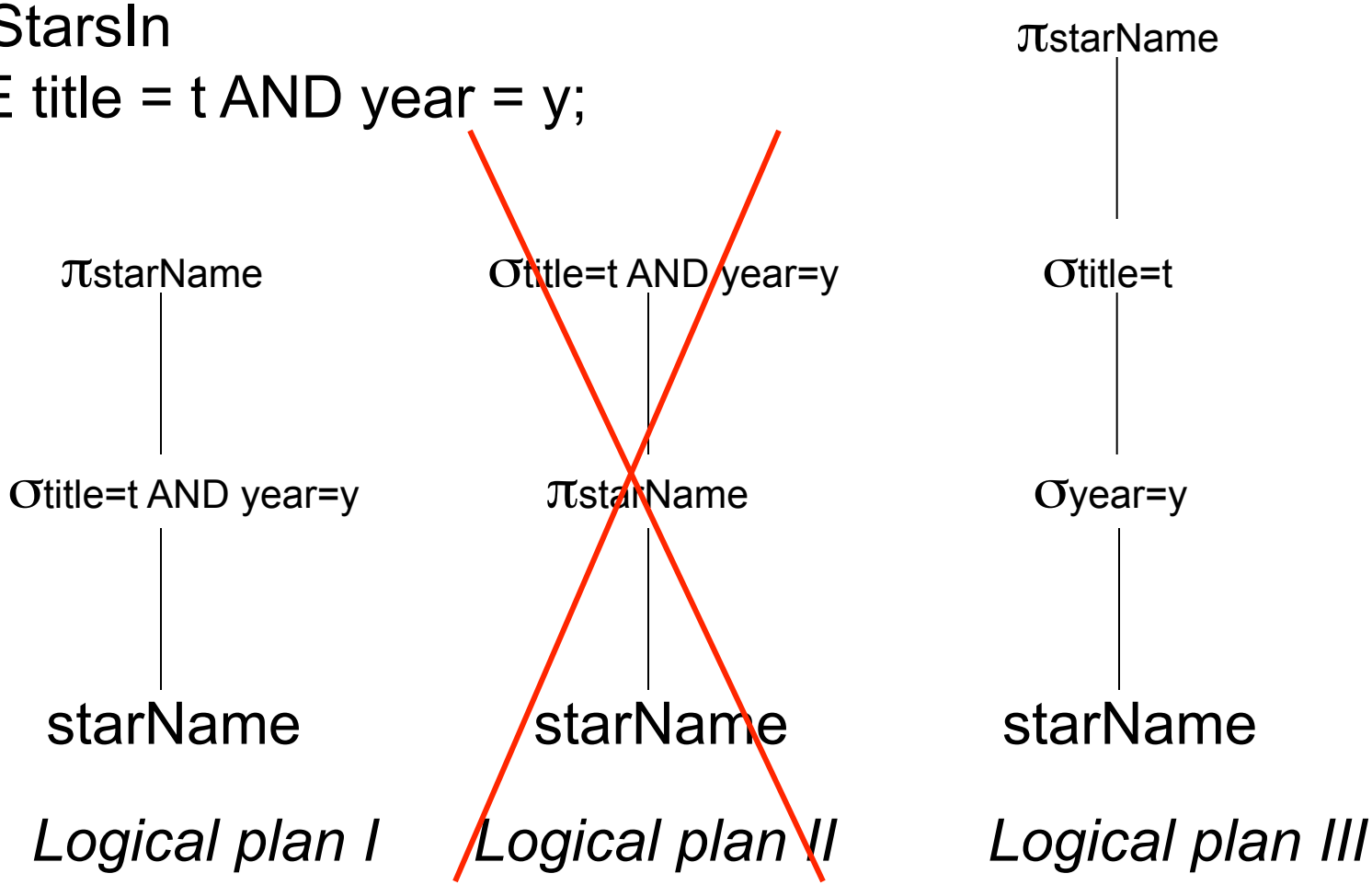


Query tree

Example: Logical Plan

StarsIn(title, year, starName)

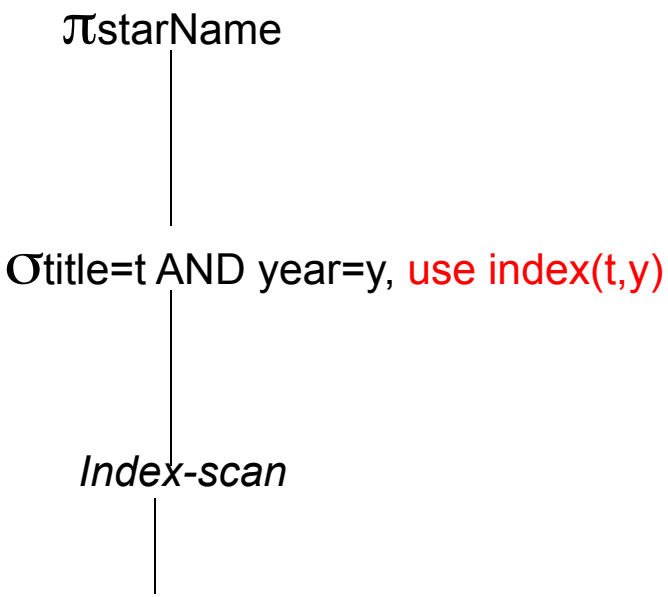
```
SELECT starName
FROM StarsIn
WHERE title = t AND year = y;
```



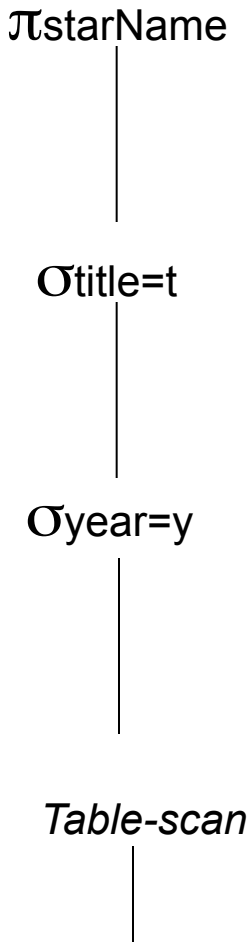
Example: Physical Plan

StarsIn(title, year, starName)

```
SELECT starName
FROM StarsIn
WHERE title = t AND year = y;
```



Physical plan I

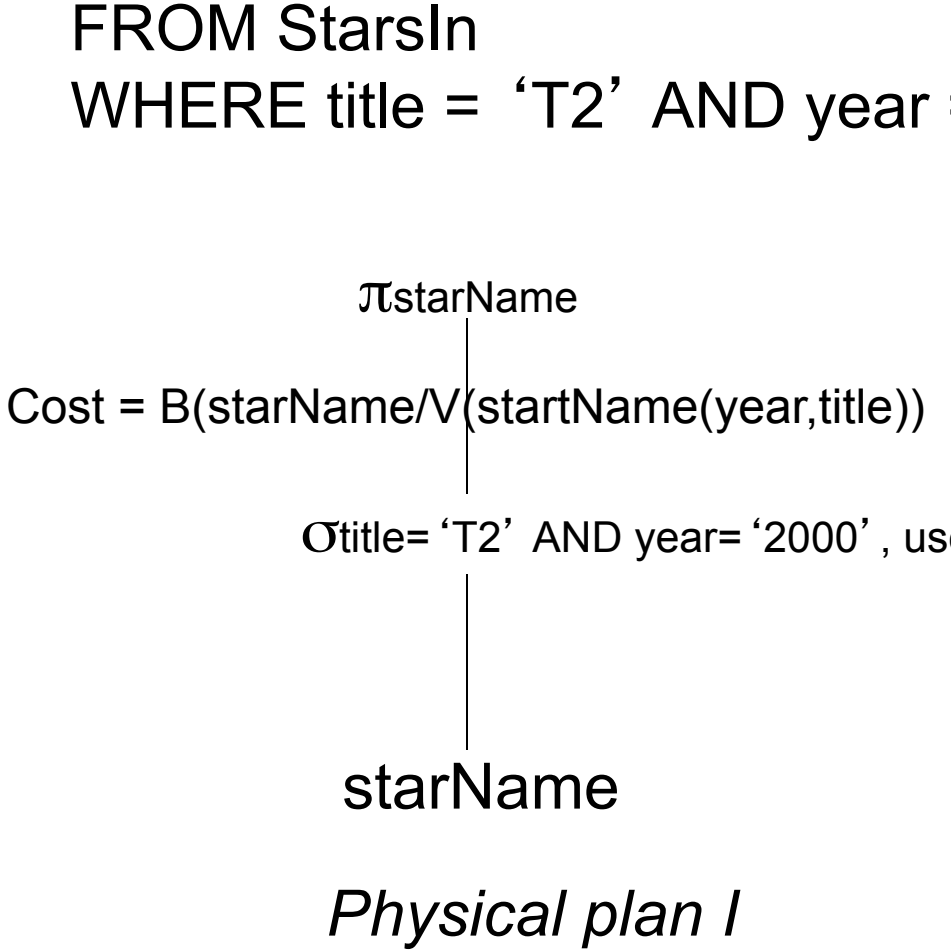


Physical plan II

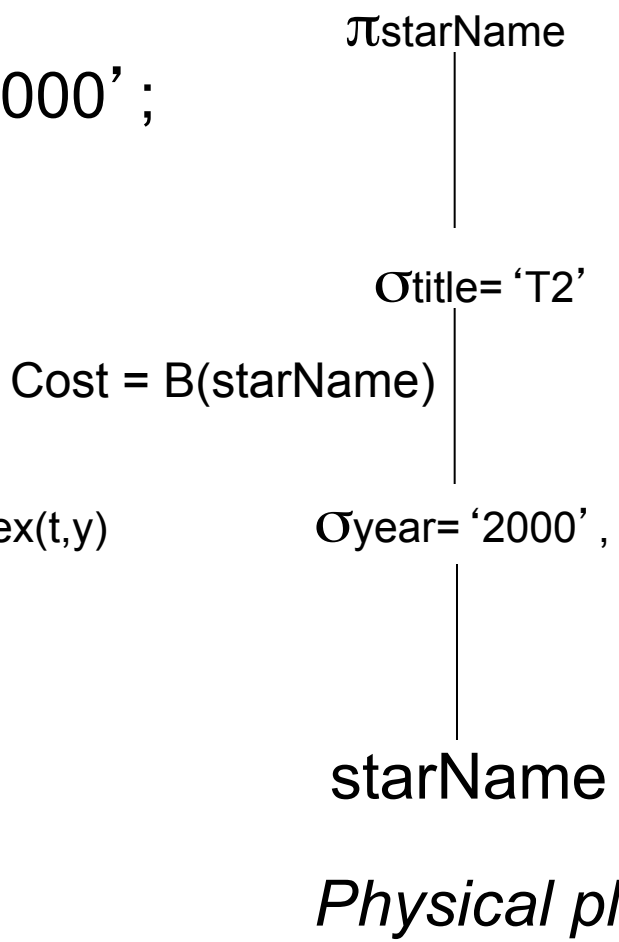
Example: Select Best Plan

StarsIn(title, year, starName)

```
SELECT starName
FROM StarsIn
WHERE title = 'T2' AND year = '2000';
```



Physical plan I



Physical plan II

Overview of Query Optimization

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
  - Example: performing a r X s followed by a selection $r.A = s.B$ is much slower than performing a join on the same condition
- Need to estimate the cost of operations
  - Computing the *precise* cost not possible without *evaluating* the plan
  - Depends critically on statistical information about relations which the database must maintain
    - E.g. number of tuples, number of distinct values for join attributes, etc.
  - Need to estimate statistics for intermediate results to compute cost of complex expressions

Overview of Query Optimization (cont.)

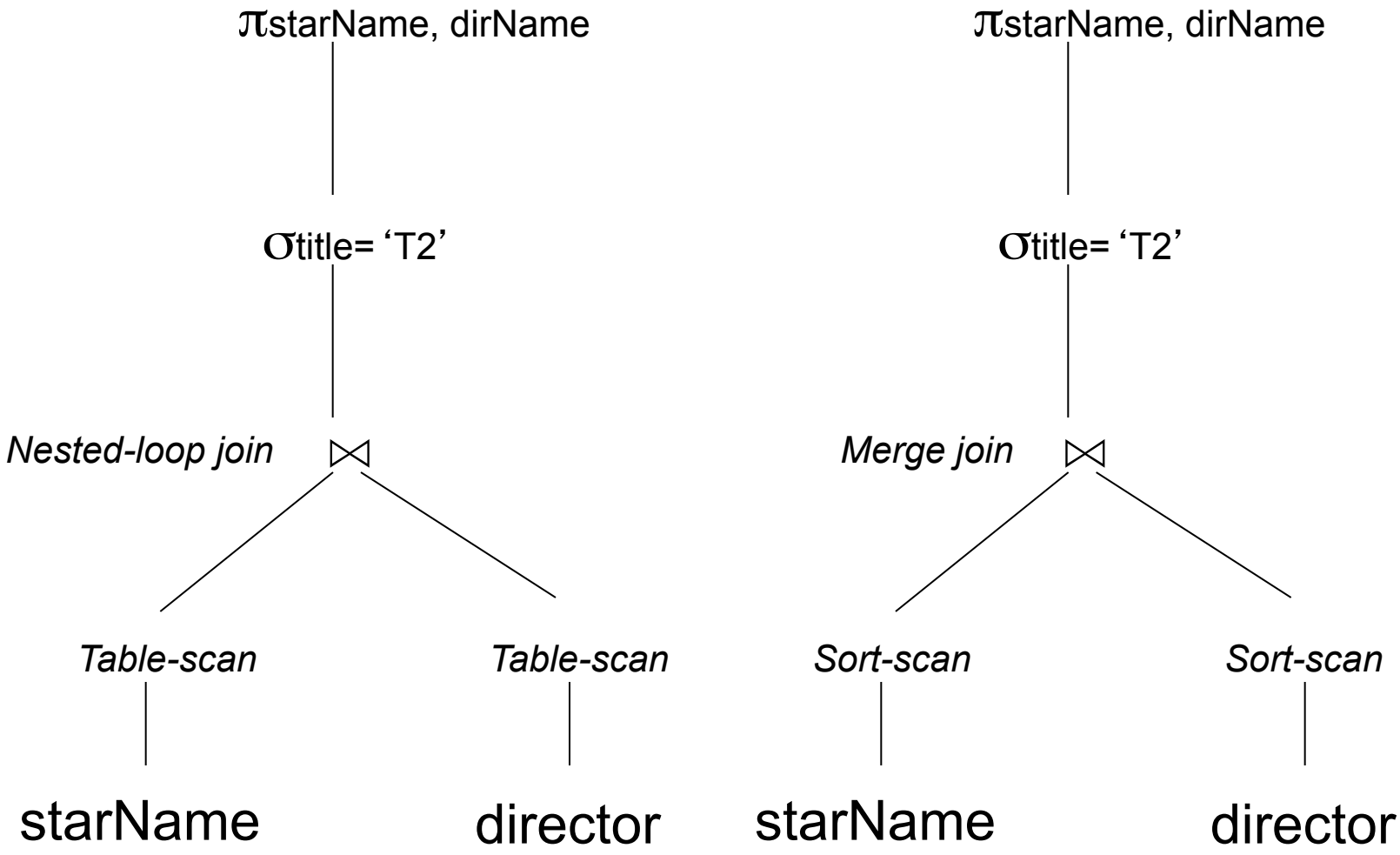
- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions
     - Use **equivalence rules** to transform an expression into an equivalent one.
  2. Annotating resultant expressions to get alternative query plans using different implementations of the relational operators
  3. Choosing the cheapest plan based on **estimated cost**
- The overall process is called **cost-based optimization.**

Query Optimization: Issues

- *Plan*: *Tree of relational algebra. ops, with choice of algorithm for each op.*
- Two main issues:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan. But there are too many!
- Reality: Avoid worst plans!

Query Plans

- Dataflow: control is determined by the flow of data



Query Plans: Model of Computation

- Dataflow: control is determined by the flow of data
- Number of I/Os determines the cost of plan
- Cost parameters
  - Size of a relation: B(R),
  - Number of tuples in a relation: T(R)
  - Number of distinct values in a column: V(R,a)
- For example:
  - What is the cost of a table-scan over starName?
  - What is the cost of an index-scan for year=1994 over starName?
  - What is the cost of a sort-scan over starName?

Query Plans: Model of Computation

- Dataflow: control is determined by the flow of data
- Number of I/Os determines the cost of plan
- Cost parameters
  - Size of a relation: B(R),
  - Number of tuples in a relation: T(R)
  - Number of distinct values in a column: V(R,a)
- Iterator model: a physical operator is an iterator
  - Similar to a cursor
  - Open: starts to obtain tuples
  - GetNext: returns the next tuple
  - Close: ends the iteration

The Table-Scan Iterator

```
Open() {
    B := the 1st block of R
    T := the 1st tuple of B}
 GetNext() {
    IF (t is past the last tuple on B) {
      increment B to the next block;
       IF (there is no next block) return NOT FOUND;
       ELSE T:= first tuple in new block B
    }
    oldT:=T
    increment T to the next tuple of B;
    RETURN oldT;
 }
```

Pipelining vs. Blocking Operators

- Pipelining: Produce tuples incrementally, 1 tuple at a time
  - Table-scan, nested-loop join
- Blocking: Need to go through all input tuples before outputing result
  - MIN, MAX, ACG

Relational Algebra Equivalences

- Allow us to choose different join orders and to `push' selections and projections ahead of joins.
- *Selections*: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$ *(Cascade)*

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ *(Commute)*

❖ *Projections:* $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ *(Cascade)*

❖ *Joins:* $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(Associative)*

$R \bowtie S \equiv S \bowtie R$ *(Commute)*

☞ Show that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection

Let x = subset of R attributes

z = attributes in predicate P (subset of R attributes)

$$\pi_x[\sigma_{p\,(R)}] = \pi_x \{\sigma_p [\, \pi_{xz\,(R)}]\}$$

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join
  $\sigma_{R.a=S.a}(R \times S) \equiv R \bowtie S$
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma (R \bowtie S) \equiv \sigma (R) \bowtie S$)
- Similarly, if a projection follows a join $R \bowtie S$, we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection

More Equivalences (cont.)

- A projection commutes with a selection that only uses attributes retained by the projection

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join $\sigma_{R.a=S.a}(R \times S) \equiv R \bowtie S$
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma (R \bowtie S) \equiv \sigma (R) \bowtie S$)
- Similarly, if a projection follows a join $R \bowtie S$, we can `push' it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection

Question

Let: X = set of attributes

Y = set of attributes

XY = X U Y

$$\pi_{xy}(R) = \pi_x[\pi_y(R)] \quad ?$$

Equivalent Expressions: Example

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch \bowtie (account \bowtie depositor)))$$

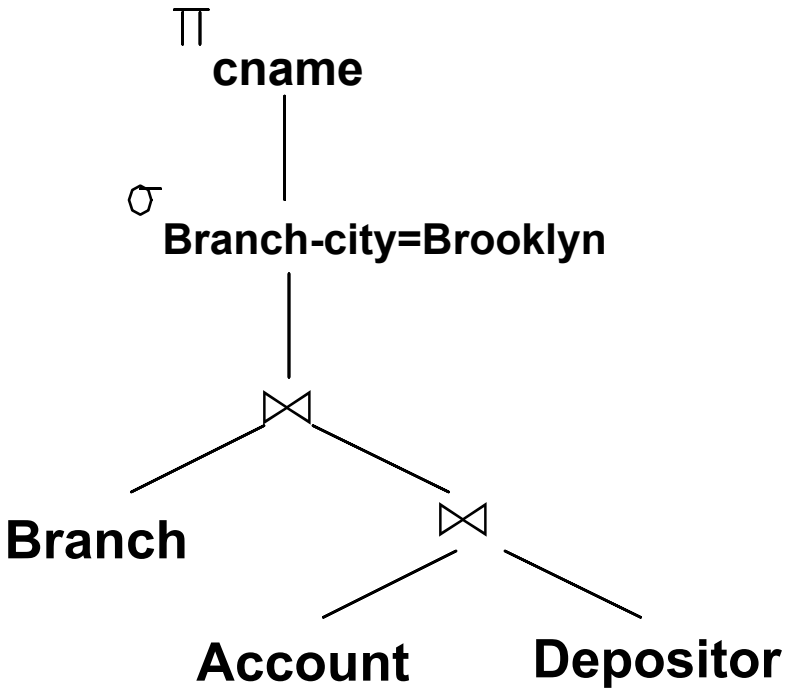
- Push selection:

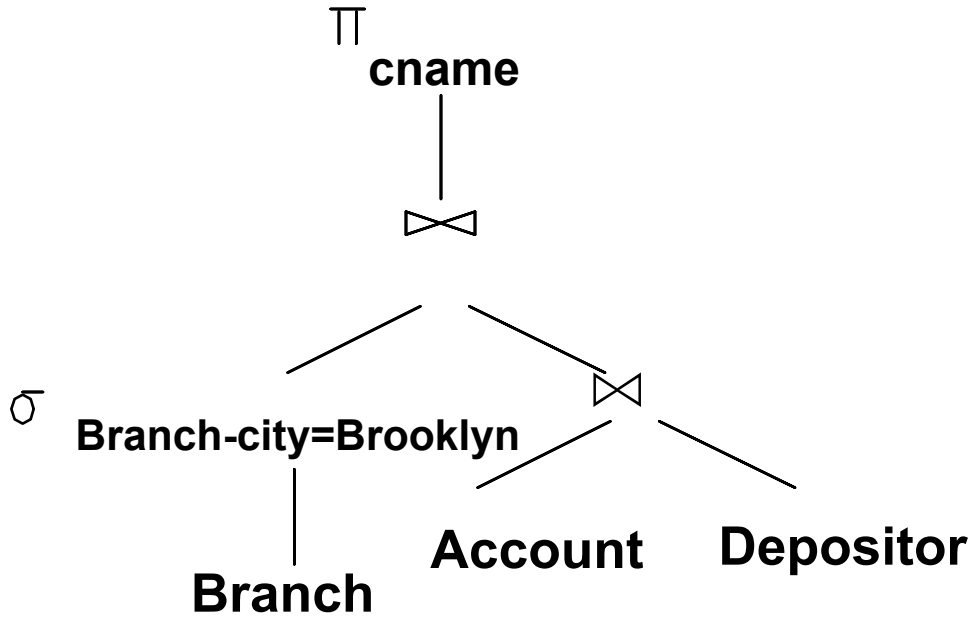
$$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch)) \bowtie (account \bowtie depositor))$$

Equivalent Expressions: Example

Find the names of all customers who have an account at any branch located in Brooklyn

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples.





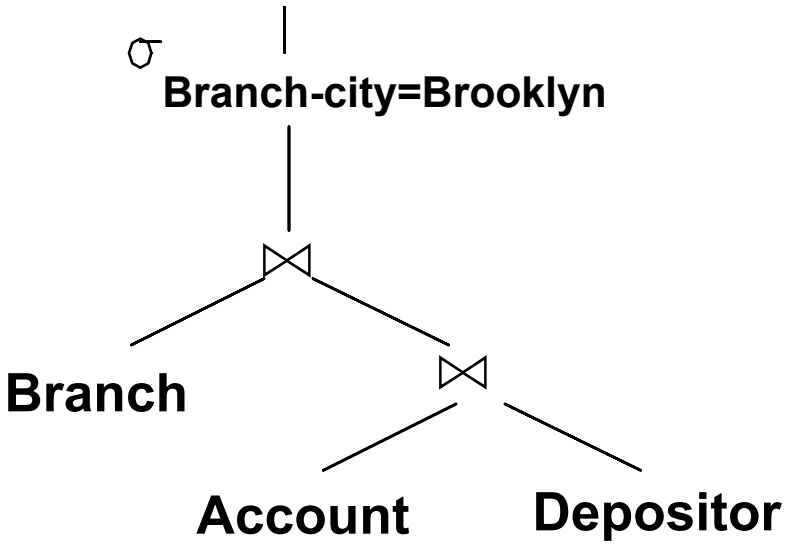
Which plan is the most efficient?

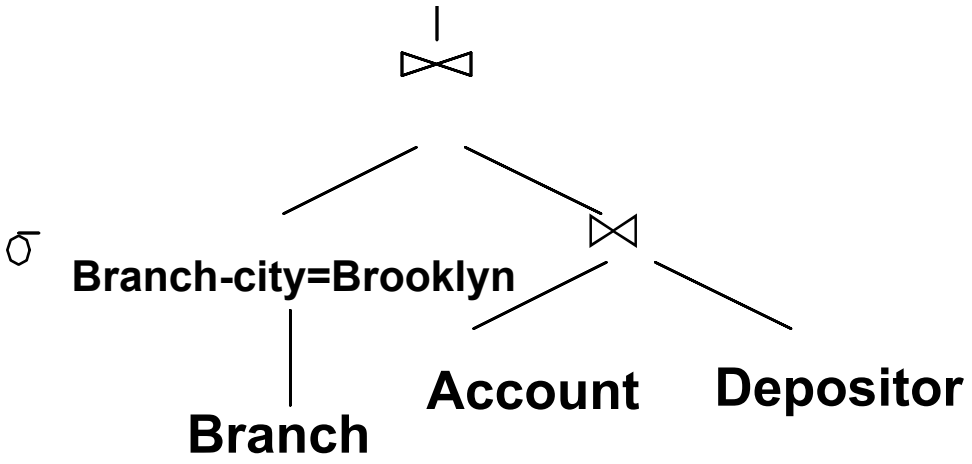
Query Optimization: Equivalent Expressions

Find the names of all customers who have an account at any branch located in Brooklyn

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples.

- *Performing the selection as early as possible reduces the size of the relation to be joined.*





Which plan is the most efficient?

Join Ordering Example

- Consider the expression

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)) \bowtie account \bowtie depositor)$$

- Could compute $account \bowtie depositor$ first, and join result with

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)$$

but $account \bowtie depositor$ is likely to be a large relation.

- Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie account$$

first.

A good join ordering is important to reduce the size of intermediate results!

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to systematically generate expressions equivalent to the given expression
  - for each expression, use all applicable equivalence rules, and add newly generated expressions to the set of expressions
  - Repeat until no new expressions can be found
- Very expensive in space and time
- Space requirements reduced by sharing common sub-expressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared, e.g., when applying join associativity
- Time requirements are reduced by not generating all expressions

Too Many Equivalent Expressions

- What is the best join-order for $r_1 \bowtie r_2 \ldots \bowtie r_n$?
- All possible orderings for n=3:
  - r1 ⋈ (r2 ⋈ r3); r1 ⋈ (r3 ⋈ r2); (r3⋈r2) ⋈ r1; (r2 ⋈ r3) ⋈ r1
  - r2 ⋈ (r1 ⋈ r3); r2 ⋈ (r3 ⋈ r1); (r3⋈r1) ⋈ r2; (r1 ⋈ r3) ⋈ r2
  - r3 ⋈ (r1 ⋈ r2); r3 ⋈ (r2 ⋈ r1); (r2⋈r1) ⋈ r3; (r1 ⋈ r2) ⋈ r3
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
  - Remember the Catalan Numbers?

  http://mathworld.wolfram.com/CatalanNumber.html

Too Many Equivalent Expressions

- What is the best join-order for $r_1 \bowtie r_2 \ldots \bowtie r_n$?
- All possible orderings for n=3:
  - r1 ⋈ (r2 ⋈ r3); r1 ⋈ (r3 ⋈ r2); (r3⋈r2) ⋈ r1; (r2 ⋈ r3) ⋈ r1
  - r2 ⋈ (r1 ⋈ r3); r2 ⋈ (r3 ⋈ r1); (r3⋈r1) ⋈ r2; (r1 ⋈ r3) ⋈ r2
  - r3 ⋈ (r1 ⋈ r2); r3 ⋈ (r2 ⋈ r1); (r2⋈r1) ⋈ r3; (r1 ⋈ r2) ⋈ r3
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
  - Remember the Catalan Numbers?

http://mathworld.wolfram.com/CatalanNumber.html

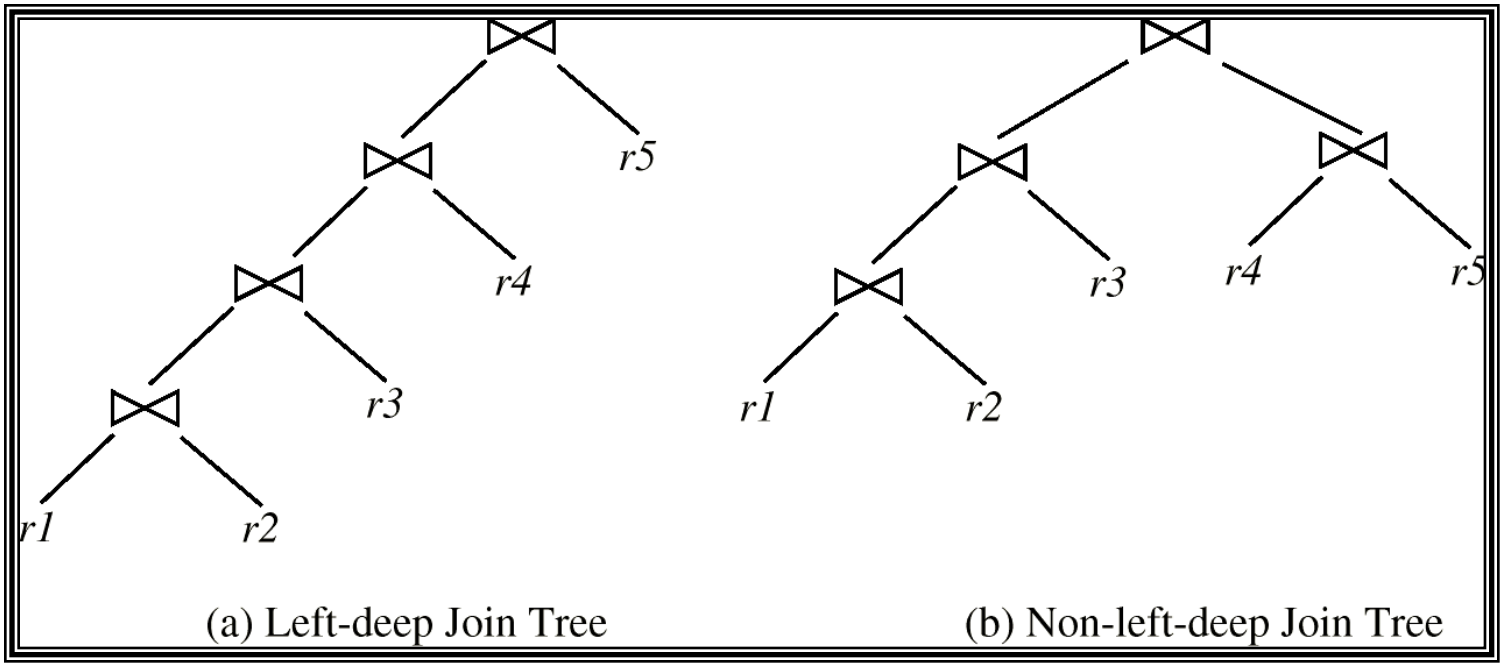
Shared expressions

Dealing with Too Many Equivalent Expressions

- No need to generate all the join orders!
- Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Restricting Logical Plans: Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



Cost of Optimization

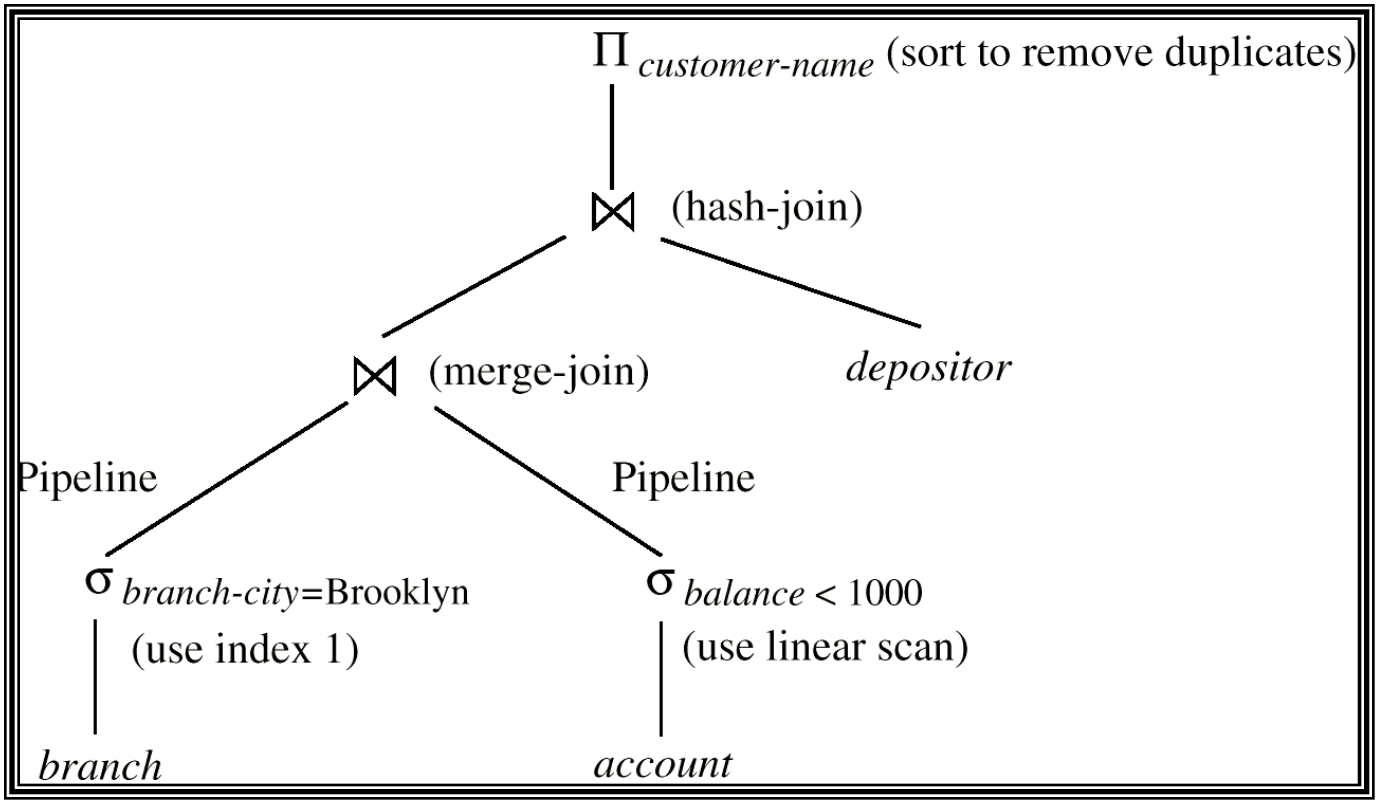
- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the n alternatives.
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Evaluation Plan

- An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when selecting evaluation plans: choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.,
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation
  - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Estimating Cost of Query Plan

Estimating size of results

- Keep statistics for relation R
  - T(R) : # tuples in R
  - S(R) : # of bytes in each R tuple
  - B(R): # of blocks to hold all R tuples
  - V(R, A) : # distinct values in R for attribute A

Estimating # of IOS-- Count # of disk blocks that must be read (or written) to execute query plan

- B(R) = # of blocks containing R tuples
- f(R) = max # of tuples of R per block
- M = # memory blocks available
- HT(i) = # levels in index i
- LB(i) = # of leaf blocks in index i

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string

B: 4 byte integer

C: 8 byte date

D: 5 byte string

T(R) = 5

V(R,A) = 3

V(R,B) = 1

S(R) = 20+4+8+5 = 37

V(R,C) = 5

V(R,D) = 4

What is the size of…

T(R) : # tuples in R
S(R) : # of bytes in each R tuple
B(R): # of blocks to hold all R tuples
V(R, A) : # distinct values in R for attribute A

W = R1 x R2

- Number of tuples: T(W) = $T(R1) \times T(R2)$
- Number of bytes: S(W) = $S(R1) + S(R2)$

W = $\sigma_{A=a}$ (R)

- S(W) = S(R)
- T(W) = ?

Example: Estimating the Size of Selections

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

V(R,A)=3
V(R,B)=1
V(R,C)=5
V(R,D)=4

$W = \sigma_{Z=val}(R) \quad T(W) = \frac{T(R)}{V(R,Z)}$

$W = \sigma_{A=cat}(R) \quad T(W) = \frac{5}{3}$

$W = \sigma_{A=bat}(R) \quad T(W) = \frac{5}{3}$

Assumes values in select expression Z = val are uniformly distributed over possible V(R,Z) values.

What if C is the key for R, what is the size of $\sigma_{C=10}(R)$?

Example: Estimating Size of Joins

W = R1 ⋈ R2

Let x = attributes of R1

y = attributes of R2

- Case 1: $x \cap y = \varnothing$ ---> T(W) = T(R1 X R2)
- Case 2: $x \cap y$ = A R1(A,B,C) R2(A,D)

Assumption:

If A values are disjoint, size = 0

A is key for R2 and fkey in R1, size = T(R2)

Almost all tuples of R1 and R2 have the same values size = T(R1)*T(R2)

$V(R1,A) \subseteq V(R2,A) \Rightarrow$

Every A value in R1 is in R2

Every tuple t of R1 has chance 1/V(R2,A) of joining with R2

Thus,

1 R1 tuple matches with T(R2)/V(R2,A) R2 tuples

Since there are T(R2) tuples in R2

$$T(W) = (T(R2) \times T(R1))/ V(R2, A)$$

What if $V(R2,A) \subseteq V(R1,A)$ I.e., every A value in R2 is in R1?

Optimization in Oracle

- SET AUTOTRACE ON
- Run query
- See query plan, e.g.,

```
 0 SELECT STATEMENT Optimizer=CHOOSE
1 0 NESTED LOOPS
2 1     NESTED LOOPS
3 2             TABLE ACCESS (FULL) OF 'BRANCH'
4 2             TABLE ACCESS (BY INDEX ROWID) OF
   'RENTAL'
5 4                 INDEX (RANGE SCAN) OF
   'IDX_RENTAL_CITY' (NON-UNIQUE)
6 1     TABLE ACCESS (BY INDEX ROWID) OF 'STAFF'
7 6       INDEX (UNIQUE SCAN) OF 'PK_STAFF' (UNIQUE)
```

Optimization in SQLServer

- stmtText

```
select * from customers c inner join orders o on c.customerid =
    o.customerid


  |--Hash Match(Inner Join, HASH:([c].[CustomerID])=([o].
    [CustomerID]),


       |--Clustered Index Scan(OBJECT:([nwind].[dbo].
    [Customers].[aaaaa_PrimaryKey] AS [c]))


       |--Clustered Index Scan(OBJECT:([nwind].[dbo].[Orders].
    [aaaaa_PrimaryKey] AS [o]))
```

Summary

- Query optimization is an important task in a relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Summary (Contd.)

- Don’t forget: statistics must be kept up to date!
- Optimization is the reason for the lasting power of relational systems
- But it is primitive…

Choice of Operations: Selection

- Tr = time to transfer 1 block
- Ts = seek time to find a block
- br = number of blocks in relation
- Linear search: applies to any file, regardless of the ordering
  - Cost = Ts + br*Tr (assumes block are stored contiguously)
- Binary search: file needs to be ordered on attribute A and query contains equality comparison with A
  - A is key: Cost = top($\log_2$(br))*(Ts +Tr)
  - A is not key: may need to retrieve additional blocks – need to *estimate the size of the selection*!

Selection Size Estimation

- Let
  - Nr = number of tuples in R
  - Br = number of blocks of R
  - Fr = blocking factor – num of tuples per block
  - V(A,r) = num of distinct values for attribute A in R

Catalog information

- Equality selection $\sigma_{A=v}(r)$
  - *SC(A, r)* : number of records that will satisfy the selection
    - SC(A,r) = Nr / V(A,r) (assumes uniform distribution of values for A)
  - $\lceil SC(A, r)/F_r \rceil$ — number of blocks that these records will occupy
  - E.g. Binary search cost estimate becomes

$$E_{a2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1$$

  - Equality condition on a key attribute: *SC(A,r)* = 1

Selections Involving Comparisons

- Selections of the form $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
- Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - C = 0 if v < min(A,r)
    - C = $$n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

      (assumes values are uniformly distributed!)
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Choice of Operations: Selection+Index

- Tt = time to transfer 1 block
- Ts = seek time to find a block
- Br = number of blocks in relation
- N = number of tuples fetched
- Primary index, equality on key: use index to retrieve the record
  - Cost = (hr+1)* (Ts+Tr) -- (assumes B+ tree)
- Primary index, equality on non-key: multiple records may need to be retrieved
  - Cost = (hr)* (Ts+Tr) + Ts + b*Tt (assumes B+ tree, b = number of blocks for records that satisfy condition)
- Secondary index: may require one I/O per tuple!
  - If A is key, cost = (hr+1)* (Ts+Tr)
  - If A is not a key, cost = (hi + n) * (Ts + Tr) (n = num of records that satisfy condition)

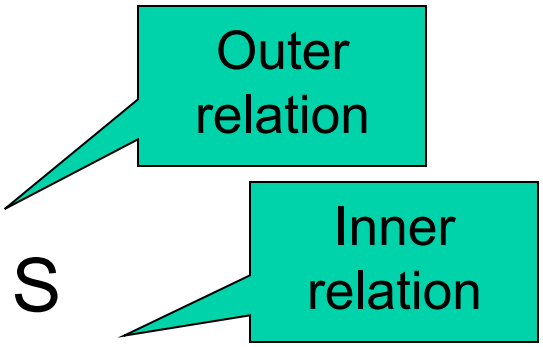
Choice of Operations: Joins

- Nested-loop Join:

  for each tr in R

  for each ts in S

  if (tr,ts) satisfy join condition, add to result



  - Expensive: examines all pairs of tuples Nr*Ns
  - Worst case: **Nr** * bs + br block transfers
  - If one relation fits in memory, make it the inner relation – cost = br + bs *much better!*
- Block nested-loop join: process relations on a per-block basis
  - Worst case: **br** * bs + br block transfers
- Index Nested-Loop Join: use index to access tuples in inner relation

Choice of Operations: More Joins

- Merge Join:

  Sort relations R and S

  Join step similar to sort-merge

  (Detailed algorithm in textbook)

  – Requires sorting of relations

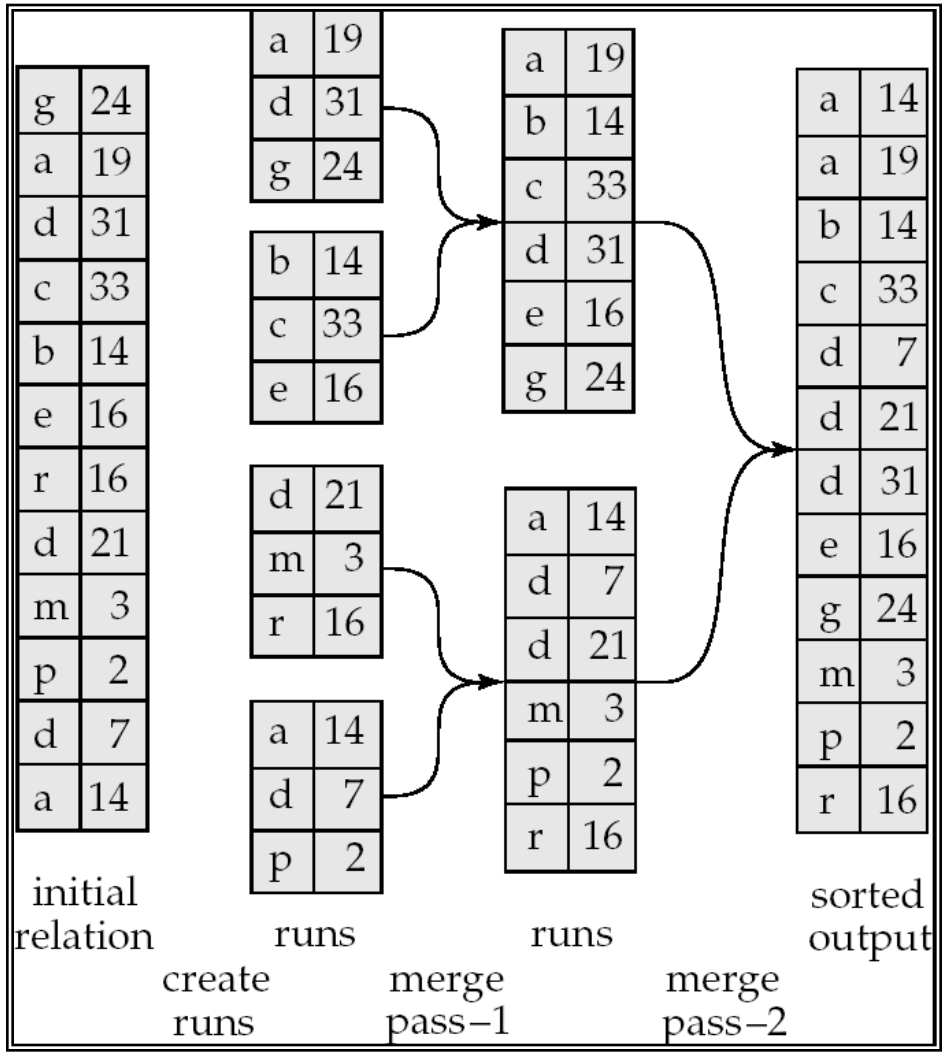
  – Returns a sorted relation, which can be useful for other operations (e.g., duplicate elimination)

- Hash join:

  A hash function h is used to partition relations R and S

  Join corresponding partitions

Example: External Sorting Using Sort-Merge



Join Size Estimation

- The Cartesian product r x s contains $n_r . n_s$ tuples
- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *depositor* ⋈ *customer, customer-name* in *depositor* is a foreign key of *customer*
  - hence, the result has $n_{depositor}$ tuples

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.

Capturing Value Distributions

- Common assumption: values are uniformly distributed
  - Not always valid
  - E.g., there are more bank accounts in the Wells Fargo Manhattan, NY branch than in Manhattan, Kansas
- Histograms are often used to better reflect the actual value distributions – lead to more accurate estimates
  - Manhattan, NY = 1,000,000; Manhattan, Kansas = 1,000; SLC, UT = 200,0000
  - Cost associated with building and maintaining histograms as data changes

Enumeration of Alternative Plans

- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans
- For queries over a single relation, queries consist of a combination of selects, projects, and aggregate ops:
  - Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

Cost Estimation

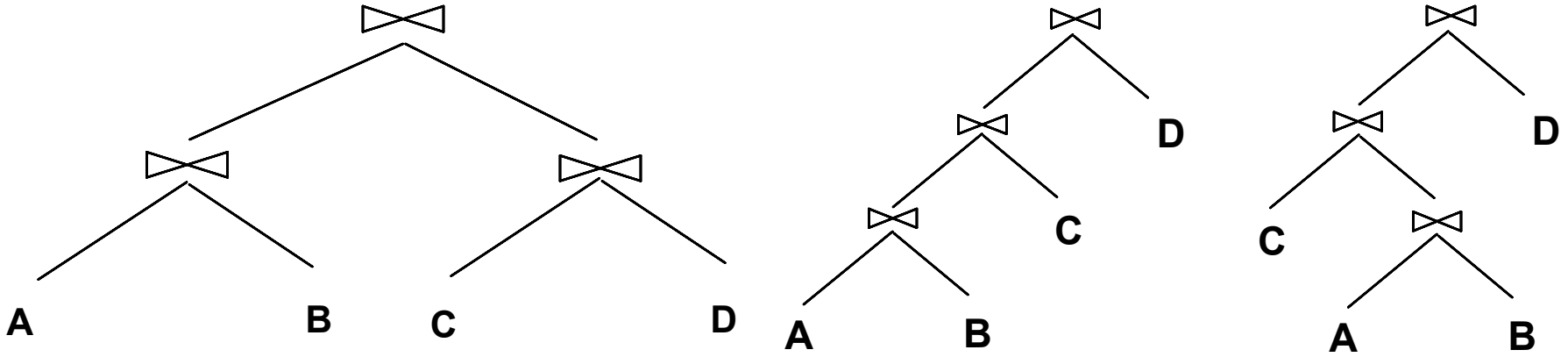
- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the n alternatives.
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Queries Over Multiple Relations

- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).

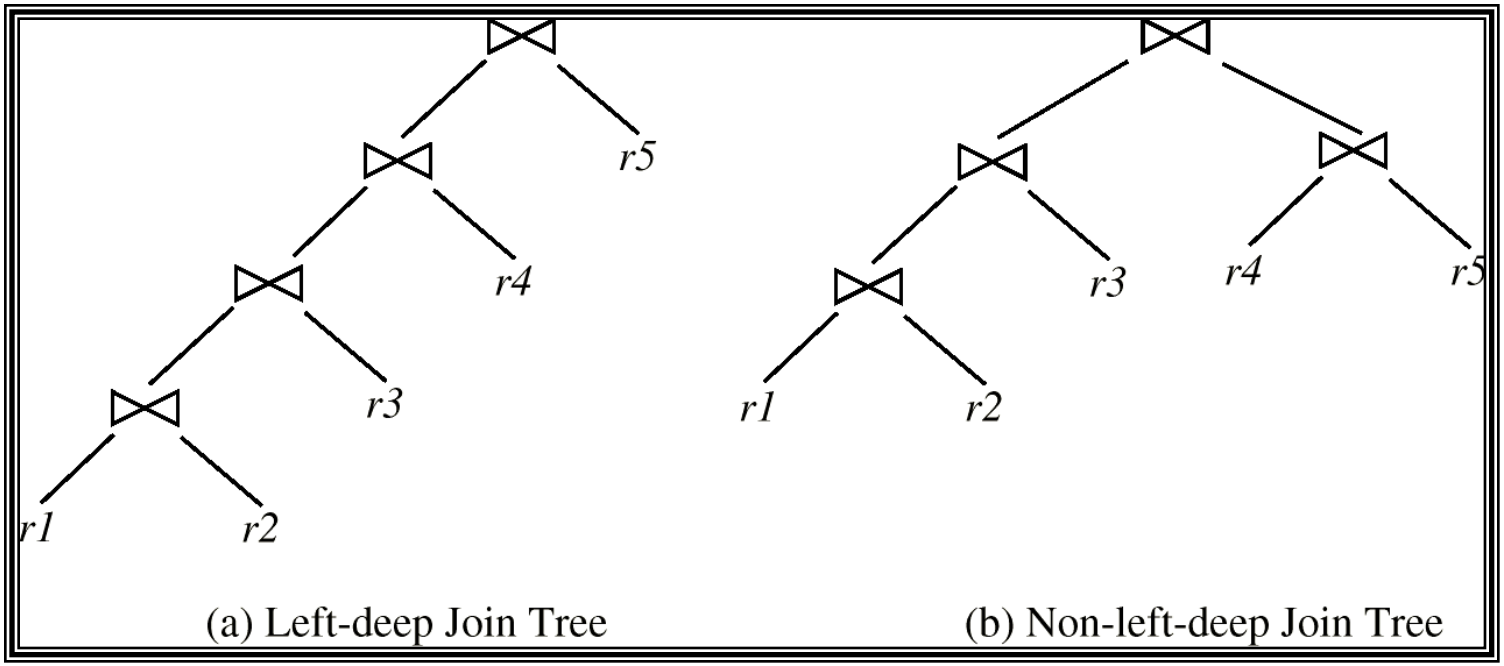


Highlights of System R Optimizer

- Impact:
  - Most widely used currently; works well for < 10 joins.
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an over-simplification, but serves for now.)

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2
     GROUP BY  S2.rating)
```

Outer block *Nested block*

- For each block, the plans considered are:
  - All available access methods, for each reln in FROM clause.
  - All *left-deep join trees* (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)