# Query Optimization in Relational Database Systems

It is safer to accept any chance that offers itself, and extemporize a procedure to fit it, than to get a good plan matured, and wait for a chance of using it.

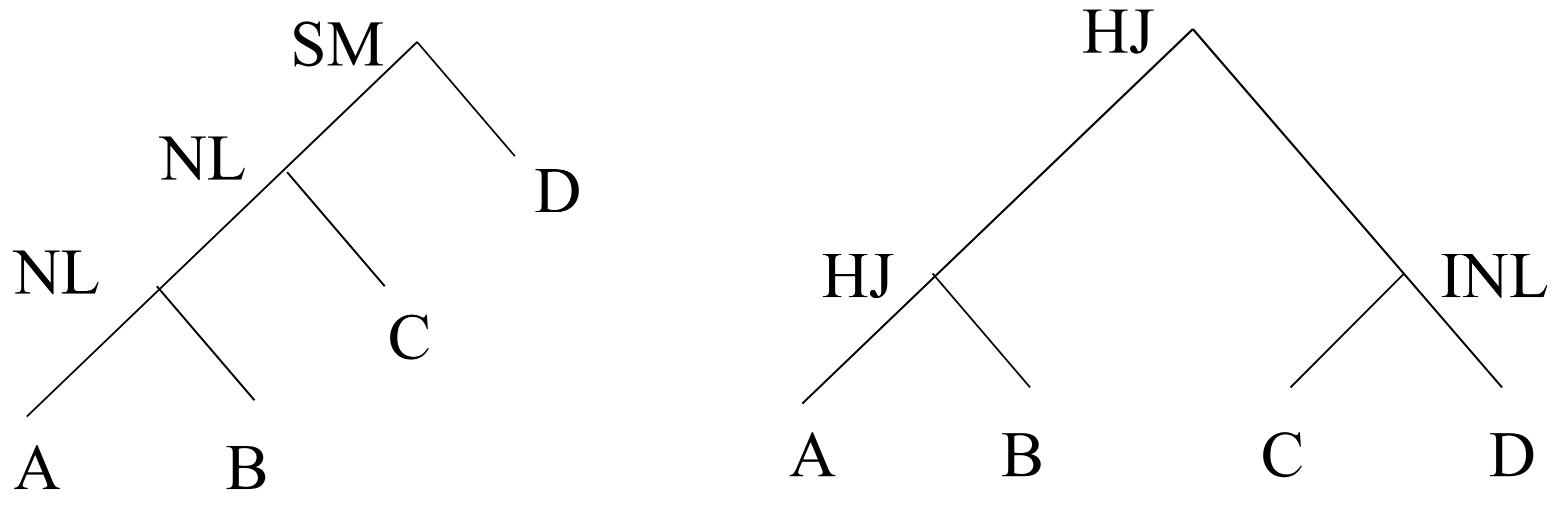
Thomas Hardy (1874)
in *Far from the Madding Crowd*

# *Query Optimization*

- Since each relational op returns a relation, ops can be *composed*!
- Queries that require multiple ops to be composed may be composed in different ways - thus *optimization* is necessary for good performance, e.g. $A \bowtie B \bowtie C \bowtie D$ can be evaluated as follows:
  - $(((A \bowtie B) \bowtie C) \bowtie D)$
  - $((A \bowtie B) \bowtie (C \bowtie D))$
  - $((B \bowtie A) \bowtie (D \bowtie C))$
  - …

# Query Optimization

- Each strategy can be represented as a query evaluation plan (QEP) - Tree of R.A. ops, with choice of algorithms for each op.




$(((A \bowtie B) \bowtie C) \bowtie D)$ $((A \bowtie B) \bowtie (C \bowtie D))$

- Goal of optimization: To find the "best" plan that compute the same answer (to avoid "bad" plans)

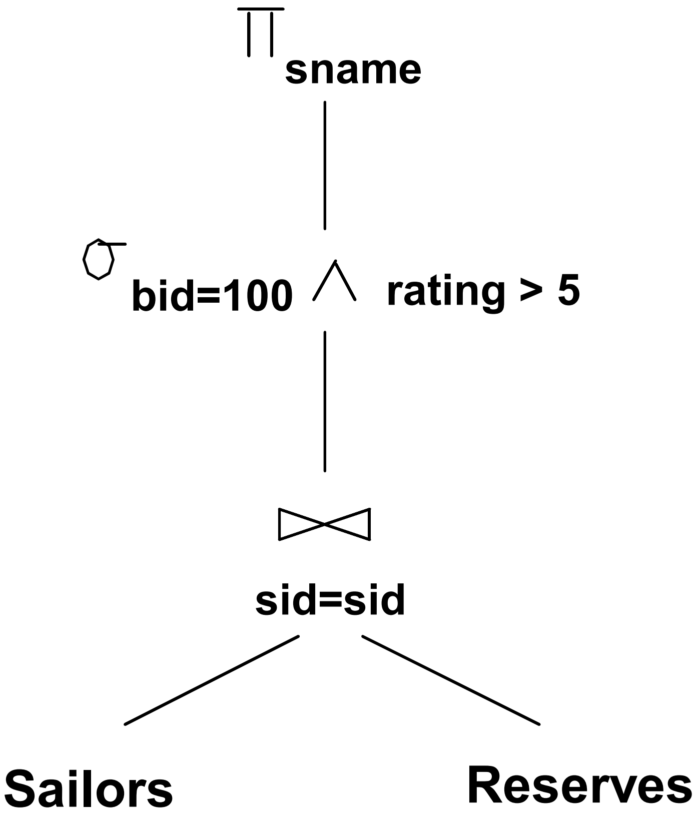
# Motivating Examples

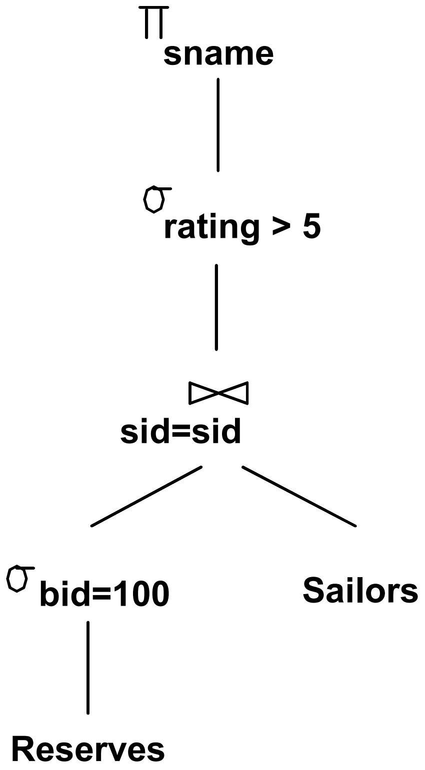
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

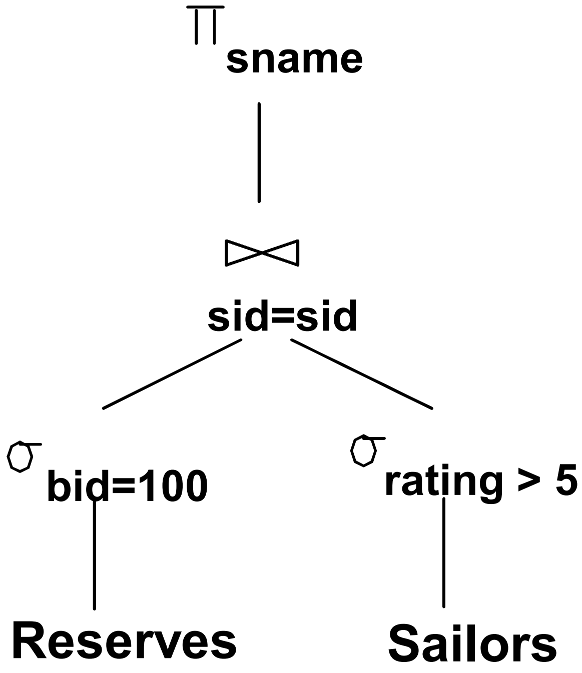
# Example

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```




this one is just joining much smaller tables together
- especially after the selection



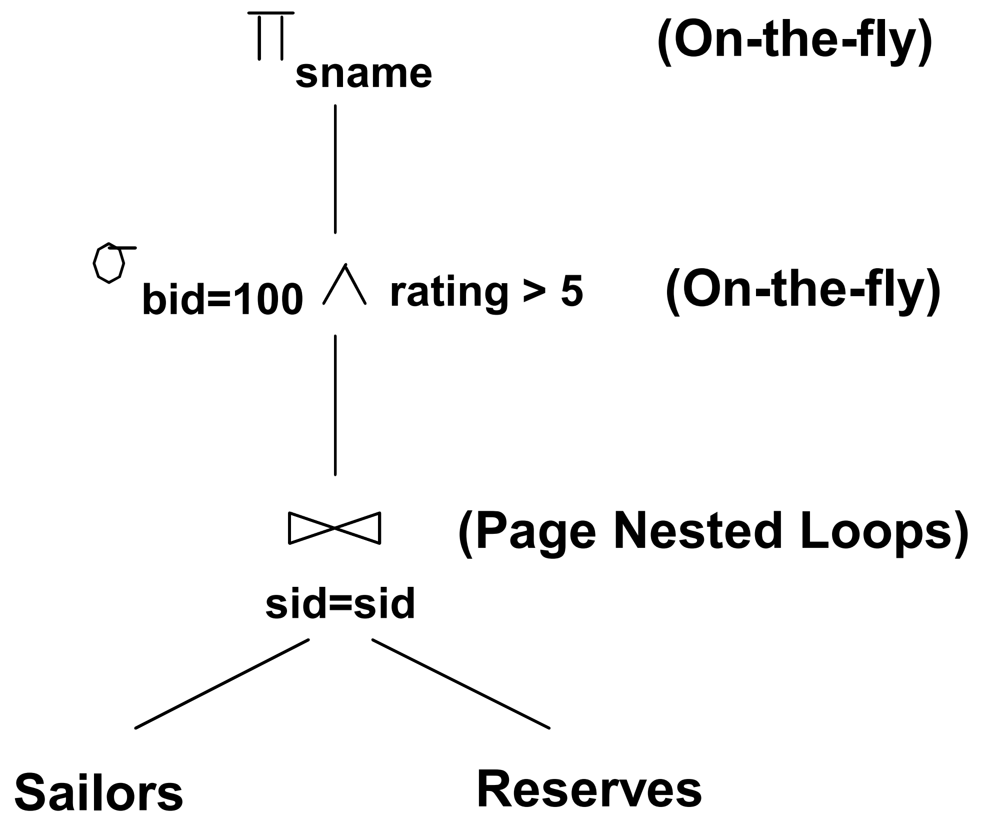

Logical plans

# Example (Cont)

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

- Cost: 500 + 500*1000 I/Os
- Memory: 1 for R
  1 for S
  1 for output
  total = 3

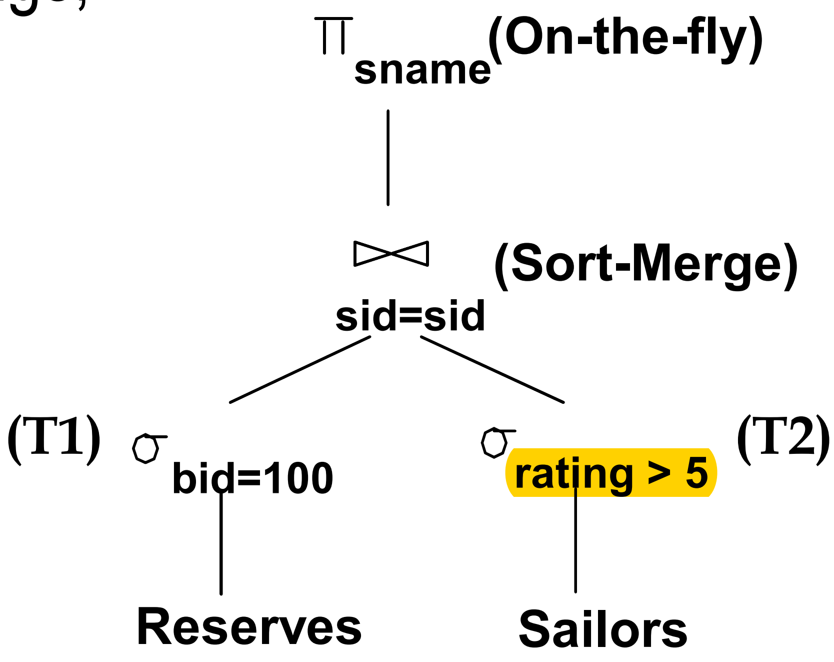
Query Evaluation Plan:




Physical plan

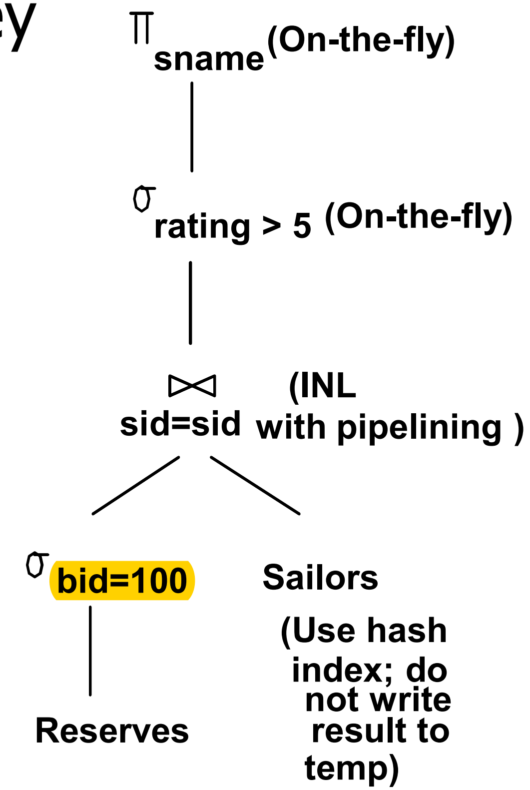
# Alternative Plan 1 (No Indexes)

- Main difference: push selections down
- Assume 5 buffers, T1 = 10 pages (100 boats, uniform distribution), T2 = 250 pages (10 ratings, uniform distribution)
- Cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution)
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings)
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os



# Alternative Plan 2 (With Indexes)

- Clustered index on bid of Reserves
  - 100,000/100 = 1000 tuples on 1000/100 = 10 pages
- Hash index on sid (format 2). Join column sid is a key for Sailors
- INL with pipelining (outer is not materialized)
  - Project out unnecessary fields from outer doesn't help
- At most one matching tuple, unclustered index on sid OK
- Did not push "rating>5" before the join. Why?
- Cost?
  - Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*2.2); total 2210 I/Os



# Query Optimizaton: Find Optimal Plan From A Set of QEPs

QEPs

P1 P2 …. Pn

C1 C2 …. Cn

Pick best!

# Relational Algebra Equivalences

What about $\sigma_{p1 \vee p2 \ldots \vee pn}(R)$?

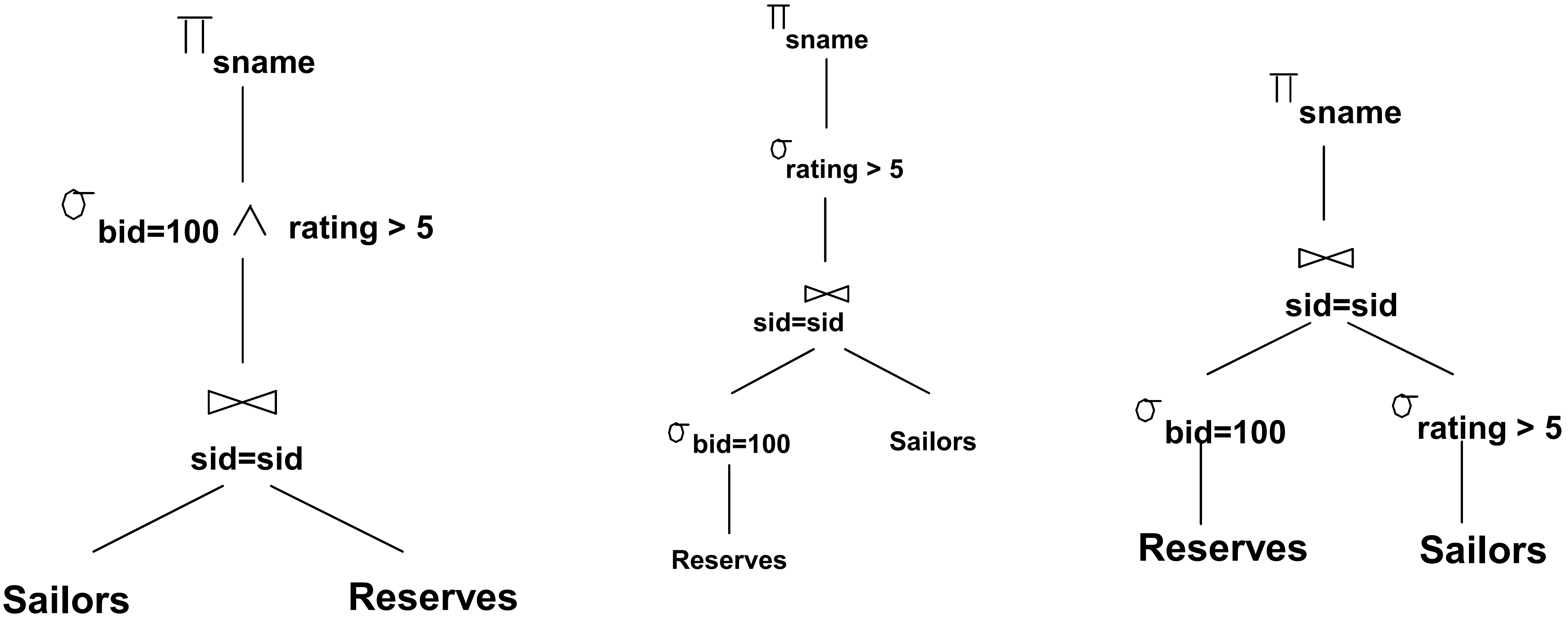
- **Cascading of selections**: $\sigma_{p_1 \wedge p_2 \wedge \cdots \wedge p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\cdots(\sigma_{p_n}(R))\cdots))$
- **Commutativity of selections**: $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
- **Cascading of projections**: $\pi_{L_1}(R) \equiv \pi_{L_1}(\pi_{L_2}(\cdots(\pi_{L_n}(R))\cdots))$, where $L_i \subseteq L_{i+1}$ for $i \in [1, n)$
- **Commutativity of cross-products**: $R \times S \equiv S \times R$
- **Associativity of cross-products**: $R \times (S \times T) \equiv (R \times S) \times T$
- **Commutativity of joins**: $R \bowtie S \equiv S \bowtie R$
- **Associativity of joins**: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
- **Others**: $R \cup S = S \cup R$, $R \cap S = S \cap R$, $R \cup (S \cup T) = (R \cup S) \cup T$, $R \cap (S \cap T) = (R \cap S) \cap T$, etc.

# Relational Algebra Equivalences

- $\pi_L(\sigma_p(R)) \equiv \sigma_p(\pi_L(R))$ if $\sigma$ involves only attributes retained by $\pi$
- $R \bowtie_p S \equiv \sigma_p(R \times S)$
- $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if $\sigma$ refers to attributes only in $R$ but not in $S$
- $\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ if $\sigma$ refers to attributes only in $R$ but not in $S$
- $\pi_L(R \times S) \equiv \pi_{L_1}(R) \times \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$ & $L_2 = L \cap attr(S)$
- $\pi_L(R \bowtie_p S) \equiv \pi_{L_1}(R) \bowtie_p \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$, $L_2 = L \cap attr(S)$, & every attribute in $p$ also appears in $L$
- Others: $\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$, etc.

# *Example*

$\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ if $\sigma$ refers to attributes only in $R$ but not in $S$



# ***Bags vs. Sets***

$R = \{a,a,b,b,b,c\}$

$S = \{b,b,c,c,d\}$

$R \cup S = ?$

- SUM is implemented: $R \cup S = \{a,a,b,b,b,b,b,c,c,c,d\}$
- Some rules cannot be used for bags
  - e.g. $A \cap_s (B \cup_s C) = (A \cap_s B) \cup_s (A \cap_s C)$

Let A, B and C be $\{x\}$

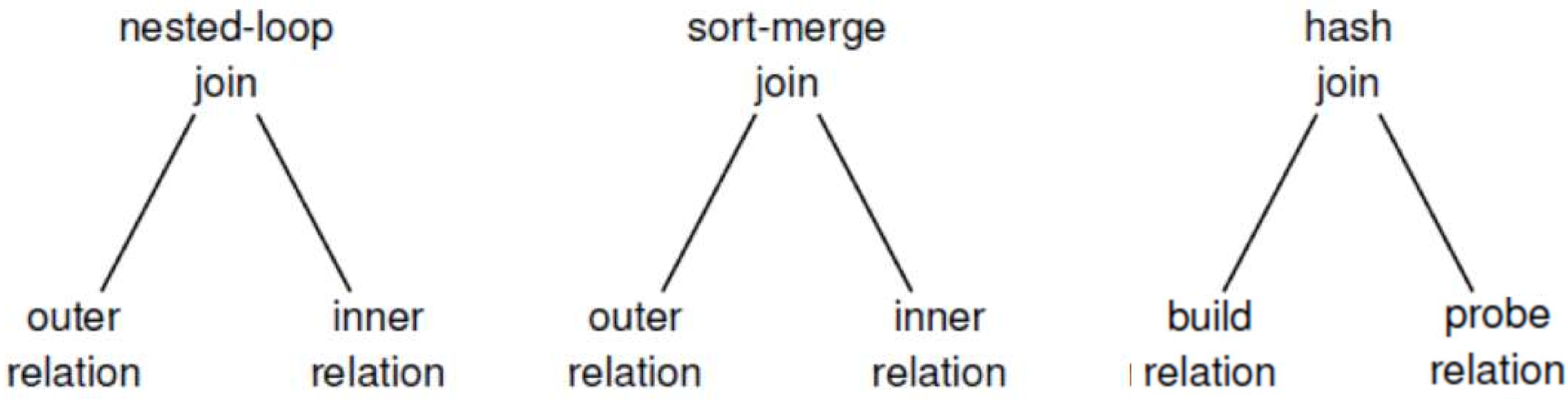
$B \cup_B C = \{x, x\}$ $\quad$ $\mathbf{A \cap_B (B \cup_B C) = \{x\}}$

$A \cap_B B = \{x\}$ $\quad$ $A \cap_B C = \{x\}$ $\quad$ $\mathbf{(A \cap_B B) \cup_B (A \cap_B C) = \{x, x\}}$

# Query Optimizer

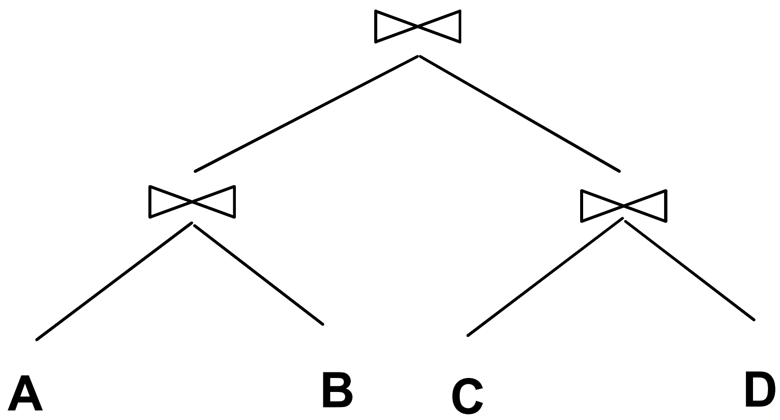
- Find the "best" plan (more often avoid the bad plans)
- Comprises the following
  - Plan space
    - huge number of alternative, semantically equivalent plans
    - computationally expensive to examine all
    - Conventional wisdom: avoid bad plans
      - need to include plans that have low cost
  - Enumeration algorithm (Search space)
    - search strategy (optimization algorithm) that *searches through the plan space*
    - has to be efficient (low optimization overhead)
  - Cost model
    - facilitate comparisons of alternative plans
    - has to be "accurate"

# Join Plan Notation

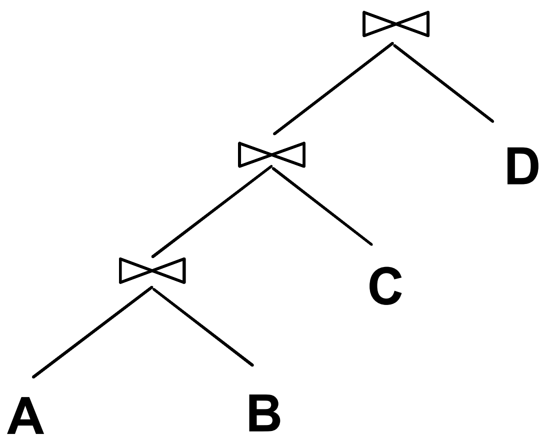


# *Plan Space*

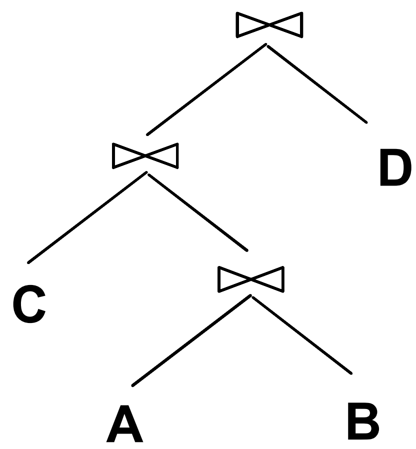
- Left-deep trees: right child has to be a base table
- Right-deep trees: left child has to be a base table
- Deep trees: one of the two children is a base table
- Bushy tree: unrestricted




Bushy tree

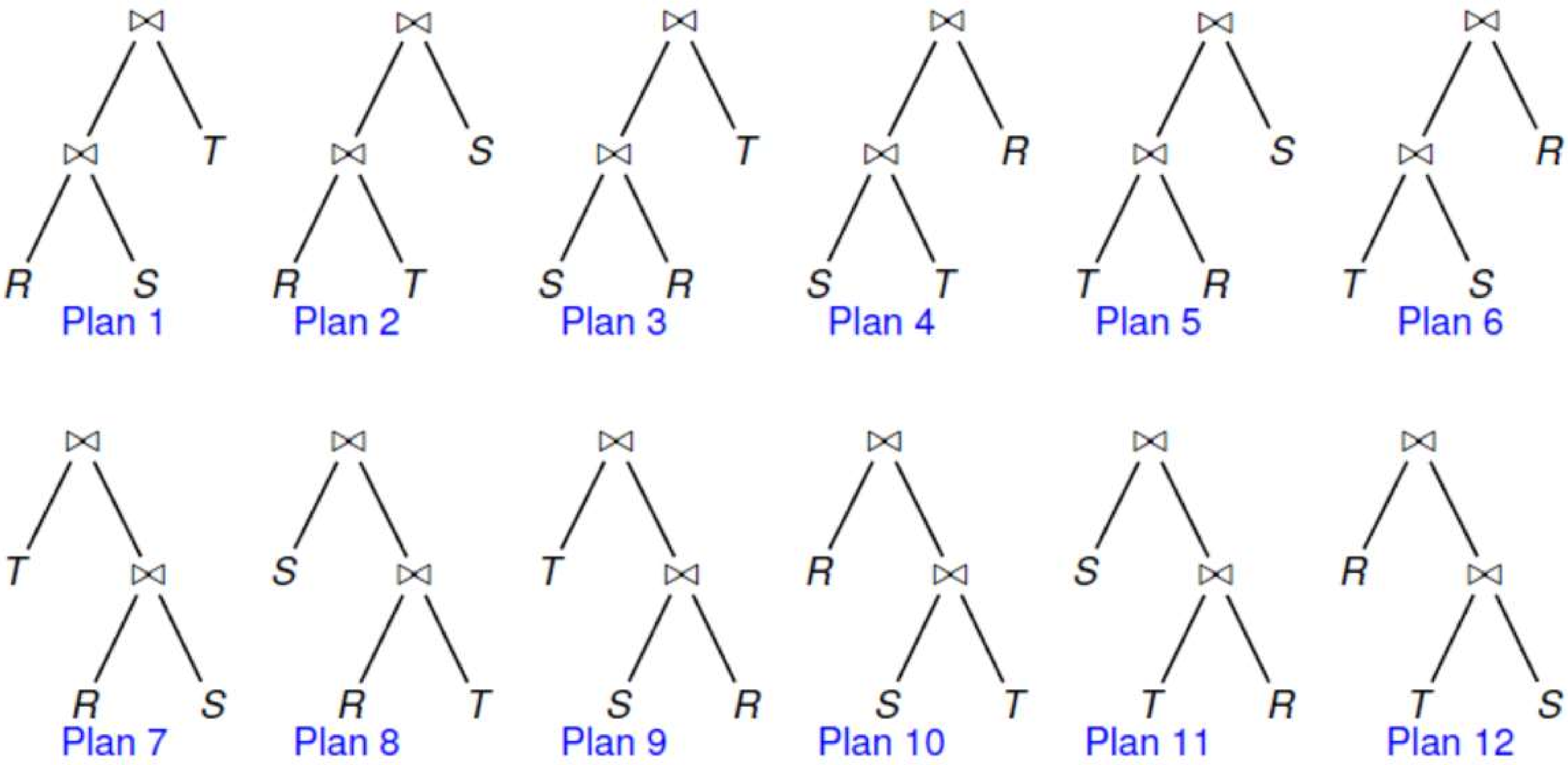



Left-deep tree




Deep tree

# Query Plan Space for $R \bowtie S \bowtie T$




This has not accounted for the algorithms!

# *Search Algorithms (and Search Space)*

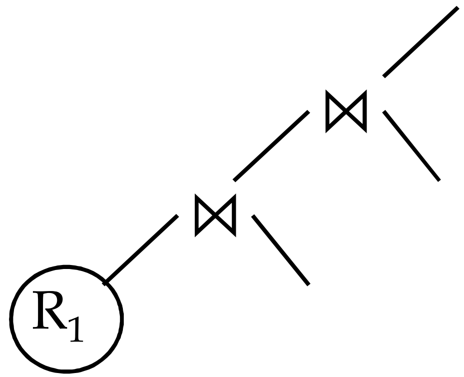
- Exhaustive (*Complete space*)
  - enumerate each possible plan, and pick the best
- Greedy Techniques (Very small – polynomial)
  - smallest relation next
  - smallest result next
  - typically polynomial time complexity
- Randomized/Transformation Techniques (Large space – can be complete if you run the algorithms indefinitely)
- System R approach (Almost complete)
  - Dynamic Programming with Pruning

# Multi-Join Queries

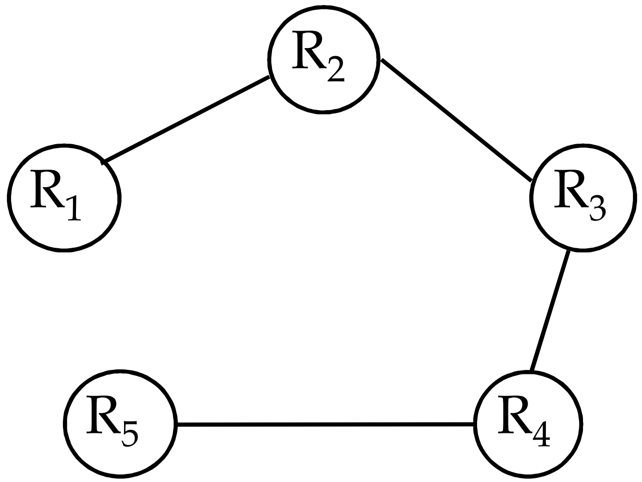
- Focus on multi-join queries first
  - Join is the most expensive operations
  - Selections and projections can be pushed down as early as possible
- Query
  - A query graph whose nodes are relations and edges represent a join condition between the two nodes

# Greedy Algorithm (Example)

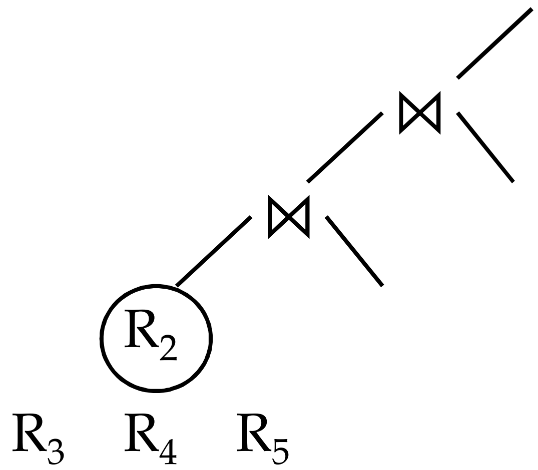
- Heuristic 1: Smallest relation next
  - Suppose $R_i < R_k$ for $i < k$




All plans must begin with $R_1$



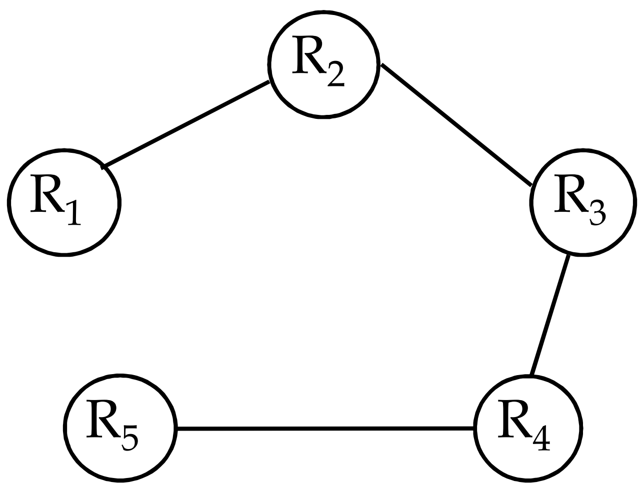

Join Graph




All plans beginning with $R_2$-$R_5$ have been pruned!

# Greedy Algorithm (Example)

- Smallest relation next
  - What if R1 < R5 < R3 < R2 < R4???

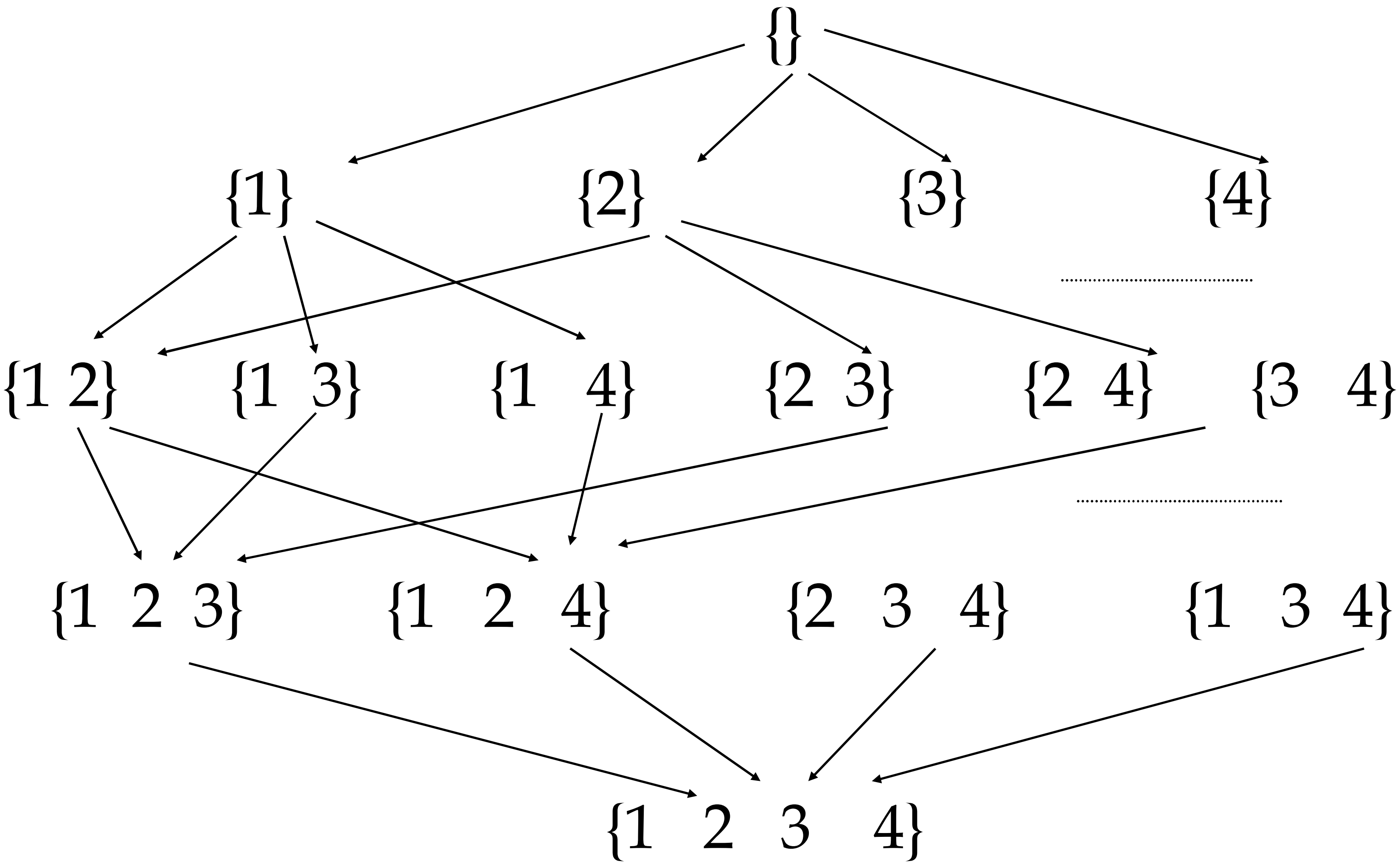



Another heuristic:
Smallest result next?

# *Dynamic Programming (Left-Deep Trees) (System R)*

- The algorithm proceeds by considering increasingly larger subsets of the set of all relations
  - Builds a plan bottom-up (beginning from 1 table, then to 2, and so on)
  - Plans for a set of cardinality *i* are constructed as extensions of the best plan for a set of cardinality *i*-1
    - For each set of cardinality *i*, we only keep ONE best plan

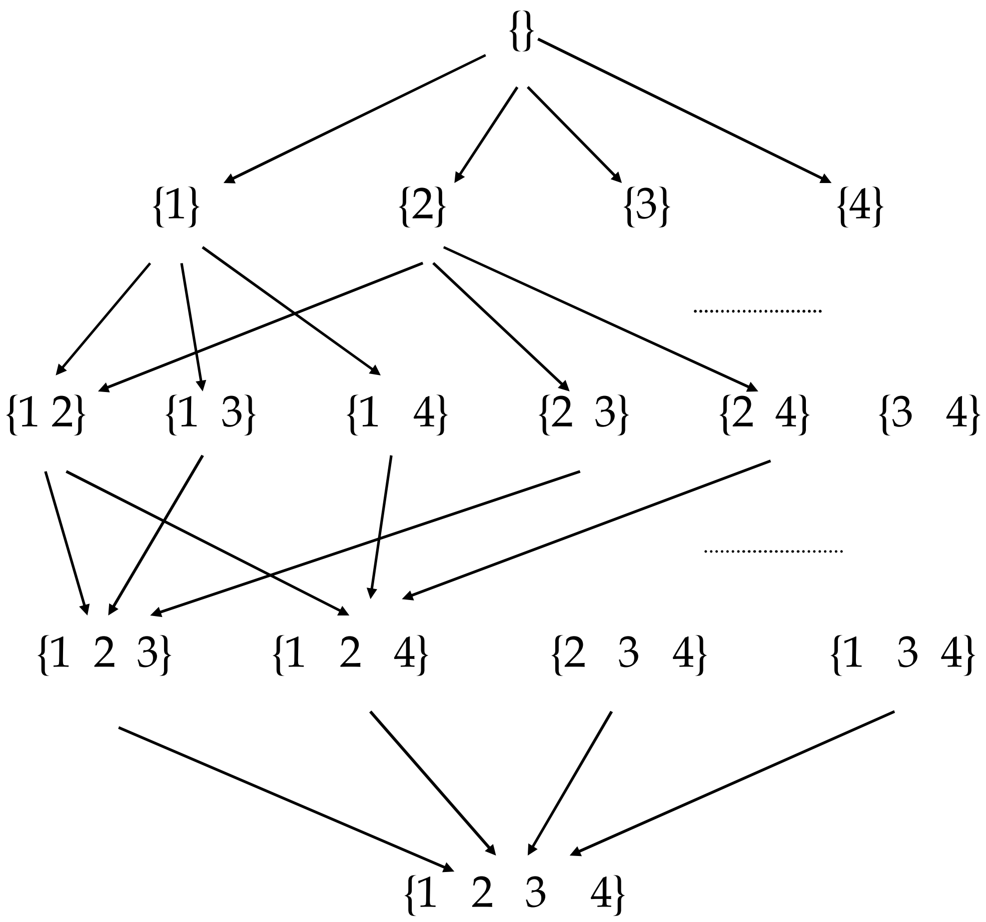
# *Dynamic Programming (Cont)*



# *Dynamic Programming (Left-Deep Trees) (System R)*

- The algorithm proceeds by considering increasingly larger subsets of the set of all relations
  - Builds a plan bottom-up (beginning from 1 table, then to 2, and so on)
  - Plans for a set of cardinality *i* are constructed as extensions of the best plan for a set of cardinality *i*-1
    - Keep only ONE best plan for each set of cardinality *i*
- Search space can be pruned based on the principal of optimality
    - if two plans differ only in a subplan, then the plan with the better subplan is also the better plan

# Principle of Optimality



# *Dynamic Programming (Left-Deep Trees) (System R)*

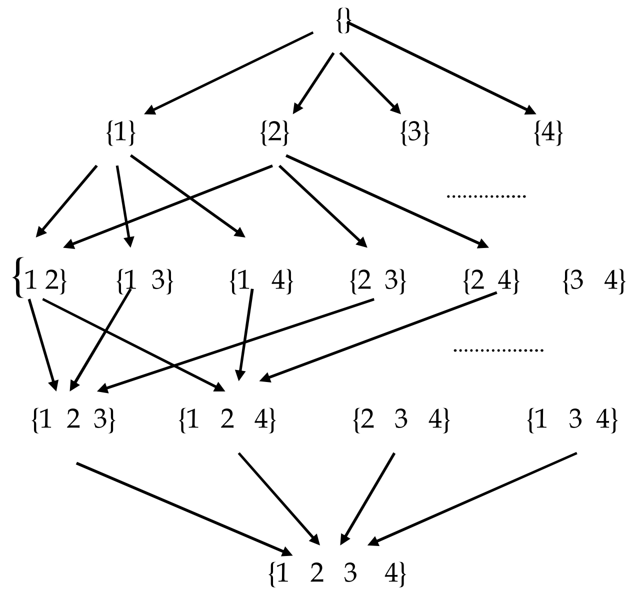
- The algorithm proceeds by considering increasingly larger subsets of the set of all relations
  - Builds a plan bottom-up (beginning from 1 table, then to 2, and so on)
  - Plans for a set of cardinality *i* are constructed as extensions of the best plan for a set of cardinality *i*-1
    - Keep only ONE best plan for each set of cardinality *i*
- Search space can be pruned based on the principal of optimality
    - if two plans differ only in a subplan, then the plan with the better subplan is also the better plan
- Computation overhead reduced due to overlapping subproblems
  - Multiple sets of cardinality *i* uses same set at cardinality (*i*-1)

# Dynamic Programming (Left-Deep Trees)

- accessPlan(R) produces the best plan for relation (single table) R
- joinPlan(p1,R) extends the (partial) join plan p1 into another plan p2 in which the result of p1 is joined with R in the best possible way
    - p1 = R1 JOIN R2 JOIN R3
    - p2 = joinPlan(p1, R) = (R1 JOIN R2 JOIN R3) JOIN R4
- Optimal plans for subsets are stored in optplan() array and are reused rather than recomputed

# Dynamic Programming (Cont)

```
for i = 1 to N
    optPlan({Ri}) = accessPlan(Ri)
for i = 2 to N {
    forall S subset of {R1, R2, … Rn} such that |S|=i {
            bestPlan = dummy plan with infinite cost
            forall Rj, Sj, |Sj| = i-1   such that S = {Rj} U Sj {
                    p = joinPlan(optPlan(Sj), Rj)
                    if cost(p) < cost(bestPlan)
                        bestPlan = p
            }
            optPlan(S) = bestPlan
    }
}
Popt = optPlan{R1, R2, … Rn}
```



# Dynamic Programming: A Concrete Example

- Schema: **R**(A,B,C,D), **S**(X,Y), **T**(E,F,G)
- Query:

  **select** *
  **from** R **join** S **on** R.A = S.X **join** T on R.D = T.F
  **where** R.B > 10
  **and** R.C = 20
  **and** T.E < 100

- Available B$^+$-tree indexes: $I_B$, $I_C$, $I_E$
- Assumptions on database system
  - Supports only one join algorithm: hash join
  - Avoids cartesian products

$$\sigma_p(R \bowtie_{R.A=S.X} S \bowtie_{R.D=T.F} T),\ p = (R.B > 10) \wedge (R.C = 20) \wedge (T.E < 100)$$

# Enumeration of Single-relation Plans

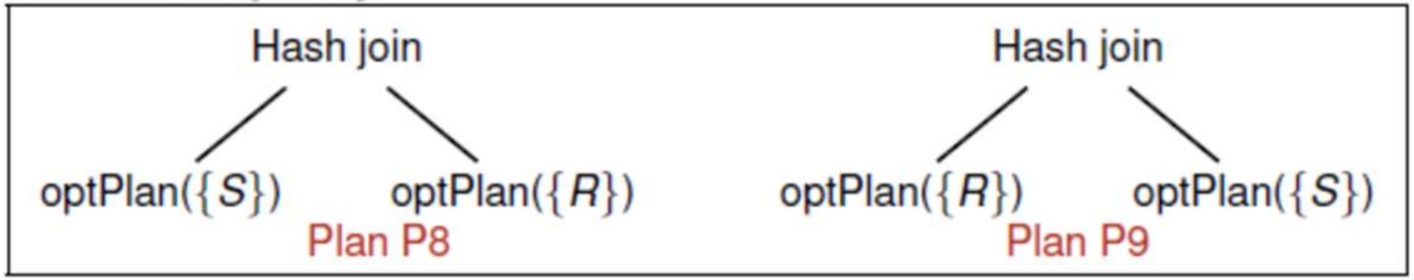
$$\sigma_p(R \bowtie_{R.A=S.X} S \bowtie_{R.D=T.F} T),\ p = (R.B > 10) \wedge (R.C = 20) \wedge (T.E < 100)$$

- **Plans for $\{$R$\}$**
  - Plan P1: Table scan with "$(B > 10) \wedge (C = 20)$"
  - Plan P2: Index seek with $I_B$ & RID-lookups with "$C = 20$"
  - Plan P3: Index seek with $I_C$ & RID-lookups with "$B > 10$"
  - Plan P4: Index intersection with $I_B$ & $I_C$, and RID-lookups
  - Assume $cost(P3) < cost(P4) < cost(P2) < cost(P1)$
  - optPlan($\{R\}$) = P3
- **Plans for $\{$S$\}$**
  - Plan P5: Table scan of S
  - optPlan($\{S\}$) = P5
- **Plans for $\{$T$\}$**
  - Plan P6: Table scan of T with "$(E < 100)$"
  - Plan P7: Index seek with $I_E$ & RID-lookups
  - Assume $cost(P7) < cost(P6)$
  - optPlan($\{T\}$) = P7

# Enumeration of Two-Relation Plans

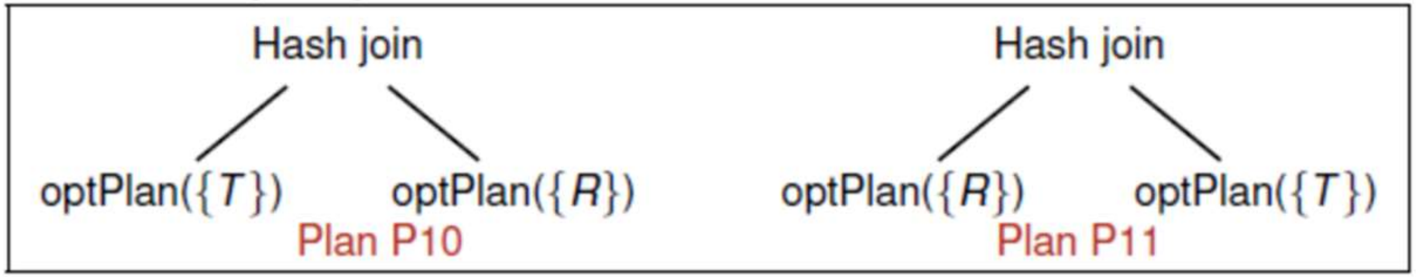
$\sigma_p(R \bowtie_{R.A=S.X} S \bowtie_{R.D=T.F} T)$, $p = (R.B > 10) \wedge (R.C = 20) \wedge (T.E < 100)$

- **Plans for $\{\mathbf{R}, \mathbf{S}\}$**




  - Assume $cost(P8) < cost(P9)$
  - optPlan($\{R, S\}$) = P8

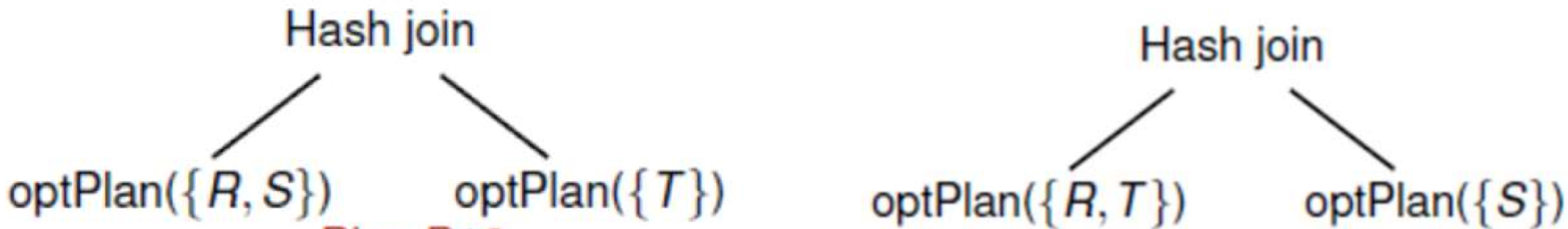
- **Plans for $\{\mathbf{R}, \mathbf{T}\}$**




  - Assume $cost(P11) < cost(P10)$
  - optPlan($\{R, T\}$) = P11

# Enumeration of Three-Relation Plans

$$\sigma_p(R \bowtie_{R.A=S.X} S \bowtie_{R.D=T.F} T),\ p = (R.B > 10) \wedge (R.C = 20) \wedge (T.E < 100)$$

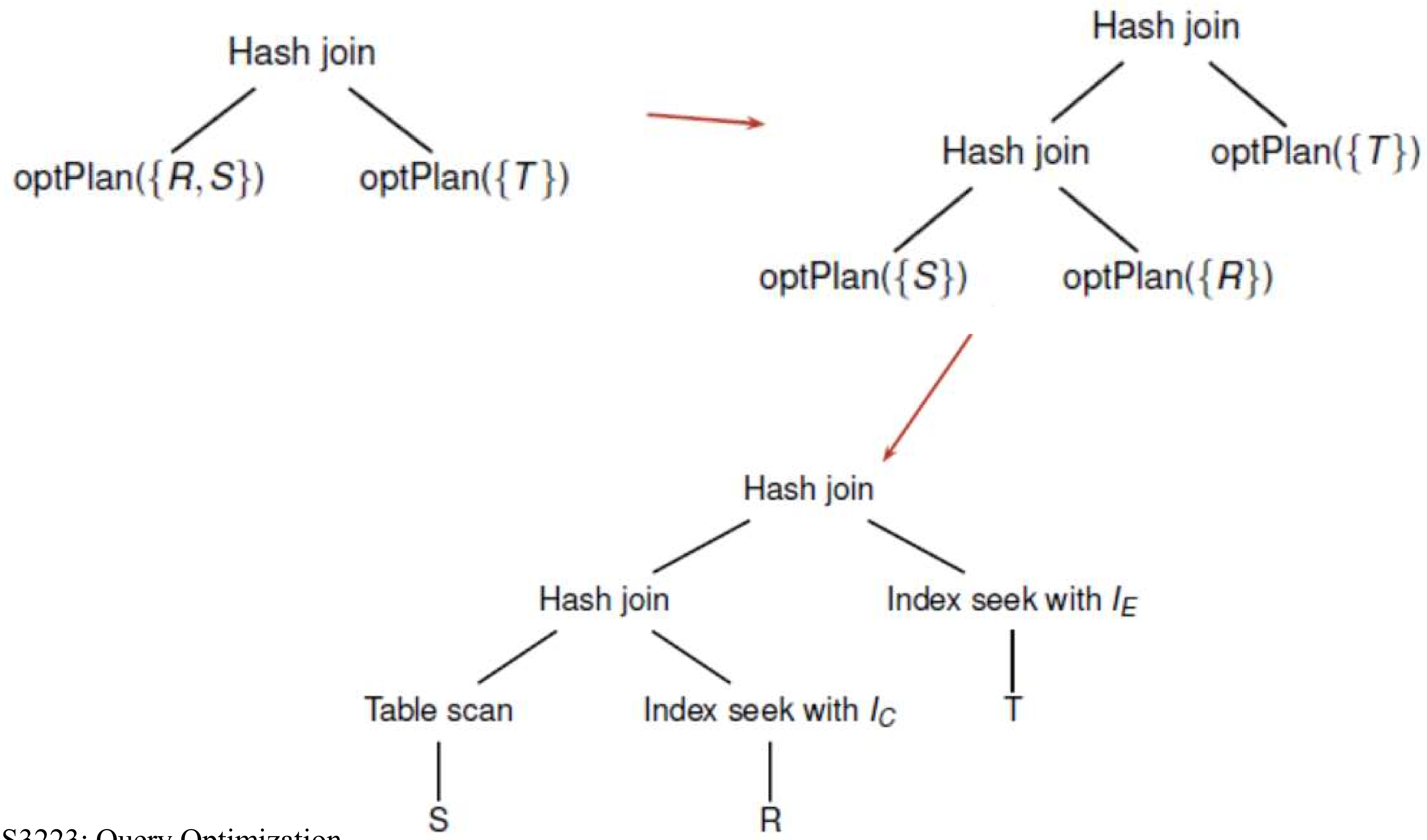



Plan P12

Plan P13

- Assume $cost(P12) < cost(P13)$
- optPlan($\{R, S, T\}$) = P12

# Optimal Plan



# Dynamic Programming (Cont)

- Time & Space complexity
  - For k relations, for left-deep trees, $2^k - 1$ entries!
  - For bushy trees, $O(3^k)$
- Is DP (as presented) optimal?

# Dynamic Programming (Cont)

- Time & Space complexity
  - For k relations, for left-deep trees, $2^k - 1$ entries!
  - For bushy trees, $O(3^k)$
- Is DP optimal?
- DP may maintain multiple plans per subset of relations
  - Interesting orders

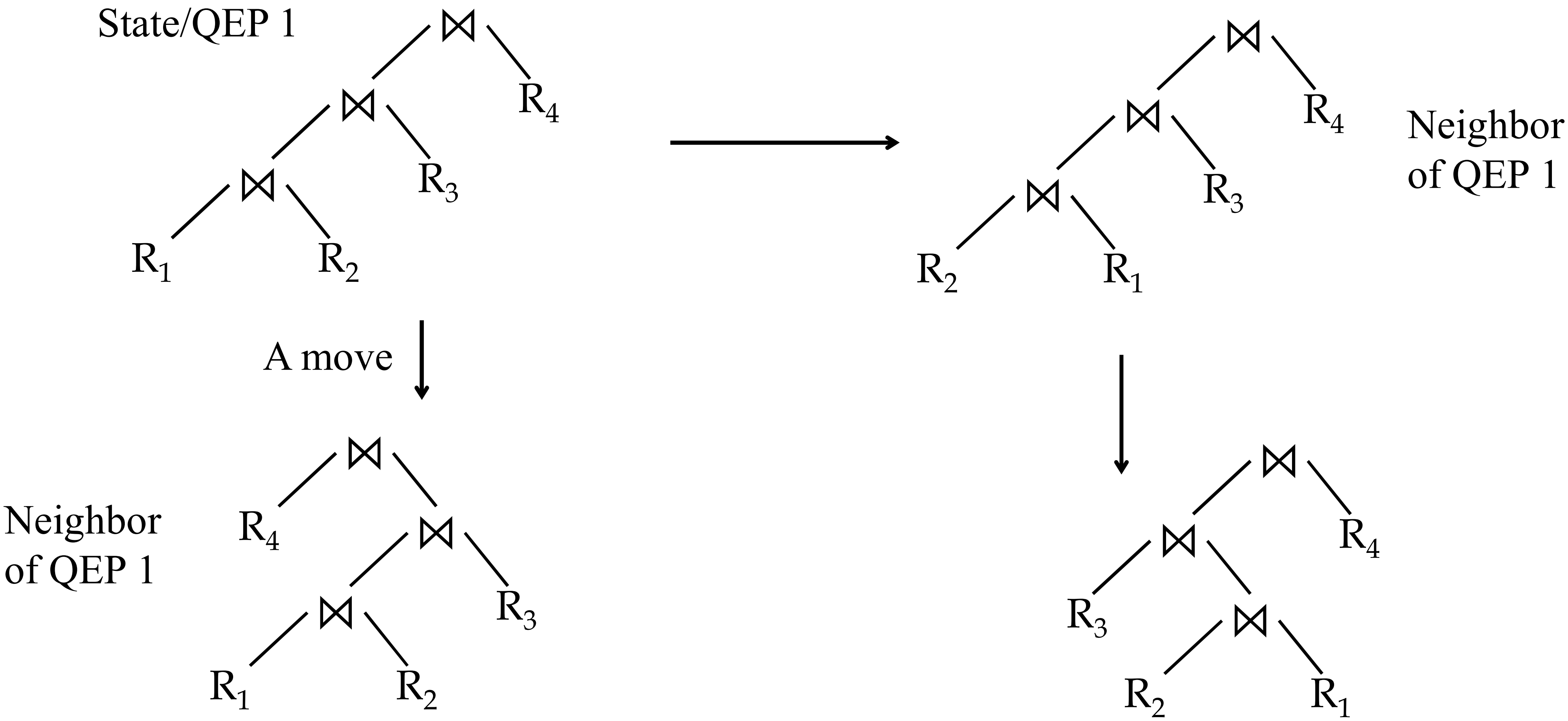
# Dynamic Programming (Cont)

- Time & Space complexity
  - For k relations, for left-deep trees, $2^k - 1$ entries!
  - For bushy trees, $O(3^k)$
- Is DP optimal?
- DP may maintain multiple plans per subset of relations
  - Interesting orders
- Is DP with interesting orders optimal?

# *Randomized Techniques*

- Employ randomized/transformation techniques for query optimization
- State space -- space of plans, State -- plan
- Each state has a cost associated with it
  - determined by some cost model
- A move is a perturbation applied to a state to get to another state
  - a move set is the set of moves available to go from one state to another
  - any one move is chosen from this move set randomly
  - each move set has a probability associated to indicate the probability of selecting the move
- Two states are *neighboring states* if one move suffices to go from one state to the other

# Randomized Algorithm (Example)



# More on Randomized Techniques

- A local minimum in the state space is a state such that its cost is lower than that of all neighboring states
- A global minimum is a state which has the lowest cost among all local minima
  - at most one global minimum
- A move that takes one state to another state with a lower cost is called a downward move; otherwise it is an upward move
  - in a local/global minimum, all moves are upward moves

# Local Optimization

Repeat until a near-optimal minimum Is reached

By doing so repeatedly, a local minimum can be reached

```
S = initialize()  // initial plan
minS = S // cost of plan S – currently the best
repeat {
    repeat {
        newS = move(S)  // move to a new plan
        if (cost(newS) < cost(S))
            S = newS
    } until ("local minimum reached")
    if (cost(S) < cost(minS))
        minS = S
    newStart(S);  // iterate with a different initial plan
} until ("stopping condition satisfied")
return (minS);
```

A move is accepted if it is a downward move, i.e., has a lower cost

# *Issues on Local Optimization*

- How is the start state obtained?
  - The state in which we start a run
  - The start state of the first run is the initial state
  - All start states should be different
  - Should be obtained quickly
    - Random
    - greedy heuristics
    - making a number of moves from the local minimum, except that this time each move is accepted irrespective of whether it increases or decreases the cost
- How is the local minimum detected?
- How is the stopping criterion detected?

*Run*: sequence of moves to a local minimum from the start state

# *Issues on Local Optimization (Cont)*

- How is the local minimum detected?
  - Not practical to examine all neighbors to verify that one has reached a local minimum
  - Based on random sampling
    - examine a sufficiently large number of neighbors
      - if any one is lower, we move to that state, and repeat the process
      - if no tested neighbor is of lower cost, the current state can be considered a local minimum
    - the number of neighbors to examine can be specified as a parameter, and is called the sequence length
      - Can also be time-based

# *Issues in Local Optimization (Cont)*

- How is the stopping criterion detected?
  - Determines the number of times that the outer loop is executed
  - Can be fixed and is given by sizeFactor*N, where sizeFactor is a parameter, N is the number of relations
    - Why N? Can it be a constant?

# What about the MOVEs? Transformation Rules

- Restricted to left-deep trees
  - all possible permutations of the N relations
  - let S be the current state, S = (… i … j … k …)
  - swap
    - select two relations, say i and j at random. Swapped i and j to get the new state newS = ( … j … i … k … )
  - 3Cycle
    - select three relations, say i and j and k at random. The move consists of cycling i, j and k: i is moved to the position of j, j is moved to the position of k and k is moved to the position of i. The resultant new state newS = ( … k … i … j … )
- Other methods (e.g., join methods)? Bushy trees?

sometimes there is a need to accept a upward plans - to jump out of the current local minimum

# Comparison between Exhaustive, Greedy and Randomized Algorithms

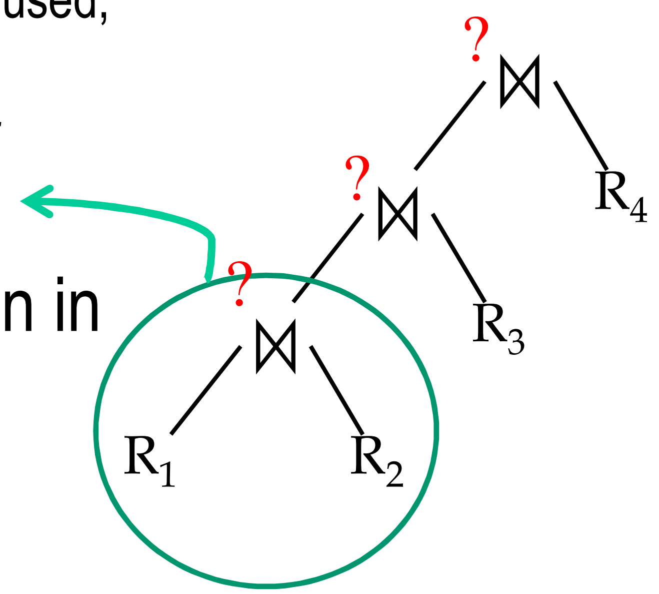
- Search space
- Plan quality
- Optimization overhead

# Cost Models

- Typically, a combination of CPU and I/O costs
- Objective is to be able to rank plans
  - exact value is not necessary
- Relies on
  - statistics on relations and indexes
  - formulas to estimate CPU and I/O cost
  - formulas to estimate selectivities of operators and intermediate results

# Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree
    - Depends on input cardinalities
    - Depends on buffer size, availability of indexes, algorithms used, etc.
      - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations
    - Typical assumptions like *uniform distribution* of data and *independence* of predicates can simplify size estimation but is error prone



# Statistics and Catalogs

- Need information about the relations and indexes involved
  Catalogs typically contain at least:
  - # tuples of R (||R||), #bytes in each R tuple (S(R))
  - # blocks/pages to hold all R tuples (|R|)
  - # distinct values in R for attribute A (V(R,A))
  - NPages for each index
  - Index height, low/high key values (Low/High) for each tree index
- Catalogs updated periodically
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok

# *Estimation Assumptions*

- Uniformity assumption
  - Uniform distribution of attribute values
- Independence assumption
  - Independent distribution of values in different attributes
- Inclusion assumption
  - For $R \bowtie_{R.A=S.B} S$, if $V(R, A) \leq V(R, B)$, then $\pi_A(R) \subseteq \pi_B(S)$
  - $V(R, A)$ is the number of distinct values of R.A

# *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 string

D: 5 byte string

$||R|| = 5$ $\quad S(R) = 37$

$V(R,A) = 3$ $\quad V(R,C) = 5$

$V(R,B) = 1$ $\quad V(R,D) = 4$

# *Size estimate for W* $= \sigma_{Z=val}$ *(R)*

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

$$\|W\| = \frac{\|R\|}{V(R,Z)}$$

S(W) = S(R)

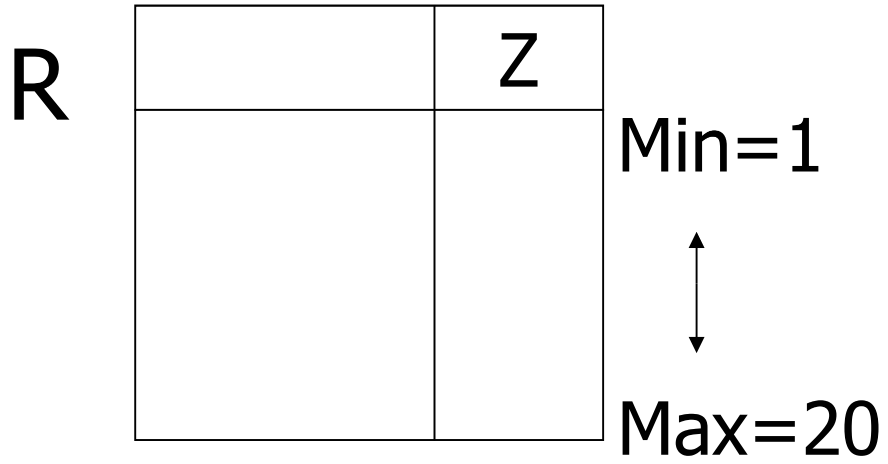
Assumption:

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

Alternative assumption: use DOM(R,Z)

# *What about W* = $\sigma_{z \geq val}$ *(R)?*

Solution: Estimate values in range




V(R,Z)=10

W= $\sigma_{z \geq 15}$ (R)

$$\text{f (fraction of range)} = \frac{20-15+1}{20-1+1} = \frac{6}{20}$$

$$||W|| = f \times ||R||$$

Alternative: (Max(Z)-value)/(Max(Z)-Min(Z))

# $W = R1 \bowtie R2$

R1

| A | B | C |
|---|---|---|
| | | |

R2

| A | D |
|---|---|
| | |

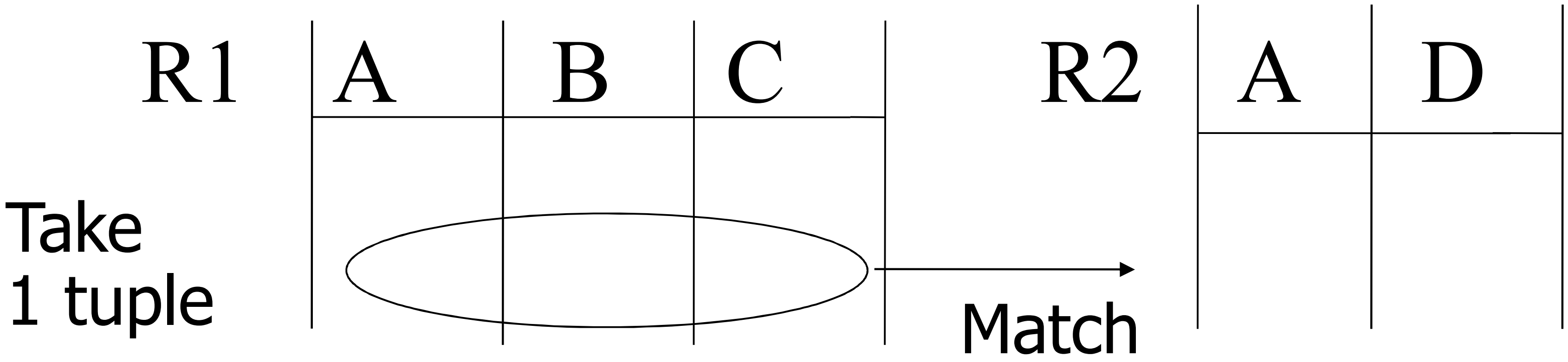
<u>Assumption:</u>

$V(R1,A) \leq V(R2,A) \Rightarrow$ Every A value in R1 is in R2

$V(R2,A) \leq V(R1,A) \Rightarrow$ Every A value in R2 is in R1

"containment of value sets"

# Computing T(W) when $V(R1,A) \leq V(R2,A)$




1 tuple of R1 matches with $\dfrac{||R2||}{V(R2,A)}$ tuples of R2

so $||W|| = ||R1|| \times \dfrac{||R2||}{V(R2,A)}$

If $V(R2,A) \leq V(R1,A)$ $\quad ||W|| = \dfrac{||R2|| \times ||R1||}{V(R1,A)}$

# *For complex expressions, need intermediate T,S,V results*

E.g. $W = [\underbrace{\sigma_{A=a}(R1)}_{\text{Treat as relation U}}] \bowtie R2$

$\|U\| = \|R1\| / V(R1,A) \qquad S(U) = S(R1)$

Also need V (U, *) !!

# Example

R1

| A | B | C | D |
|---|---|---|---|
| cat | 1 | 10 | 10 |
| cat | 1 | 20 | 20 |
| dog | 1 | 30 | 10 |
| dog | 1 | 40 | 30 |
| bat | 1 | 50 | 10 |

V(R1,A)=3

V(R1,B)=1

V(R1,C)=5

V(R1,D)=3

$U = \sigma_{A=a}(R1)$

V(U,A) = ? 1

this is 1 since selecting 1 of the distinct value

V(U, B) = ? 1

1 since only 1 distinct value

V(U,C) = ? $\frac{||R1||}{V(R1,A)}$

this will be the number of tuples in the result since all values are distinct

V(D,U) … somewhere in between V(U,B) and V(U,C)

# *For Joins* $U = R1(A,B) \bowtie R2(A,C)$

$V(U,A) = \min \{ V(R1, A), V(R2, A) \}$

$V(U,B) = V(R1, B)$

$V(U,C) = V(R2, C)$

(Assumption: Preservation of value sets)

A problem would be when U is smaller than all the other tables

## *Example*

$$Z = R1(A,B) \bowtie R2(B,C) \bowtie R3(C,D)$$

| | |
|---|---|
| $\boxed{R1}$ | $\|R1\| = 1000 \quad V(R1,A)=50 \quad V(R1,B)=100$ |
| $\boxed{R2}$ | $\|R2\| = 2000 \quad V(R2,B)=200 \quad V(R2,C)=300$ |
| $\boxed{R3}$ | $\|R3\| = 3000 \quad V(R3,C)=90 \quad V(R3,D)=500$ |

$$Z = R1(A,B) \bowtie R2(B,C) \bowtie R3(C,D)$$

$\|R1\| = 1000$ $V(R1,A)=50$ $V(R1,B)=100$

$\|R2\| = 2000$ $V(R2,B)=200$ $V(R2,C)=300$

$\|R3\| = 3000$ $V(R3,C)=90$ $V(R3,D)=500$

# *Partial Result:* $U = R1 \bowtie R2$

$$\|U\| = \frac{1000 \times 2000}{200}$$

$V(U,A) = 50$

$V(U,B) = 100$

$V(U,C) = 300$

## $Z = U \bowtie R3$

$$\|Z\| = \frac{1000 \times 2000 \times 3000}{200 \times 300}$$

$V(Z,A) = 50$

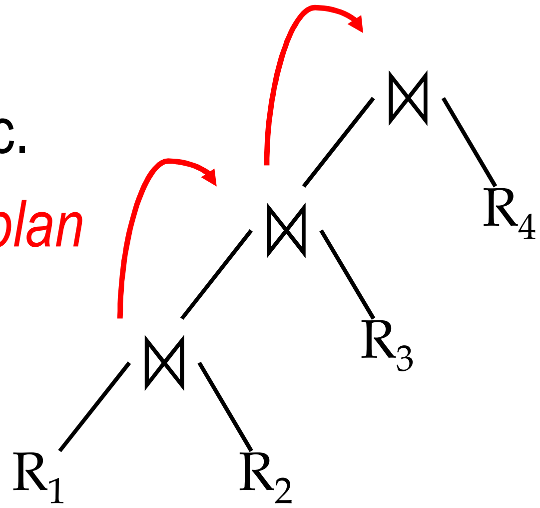
$V(Z,B) = 100$

$V(Z,C) = 90$

$V(Z,D) = 500$

# Errors in Estimating Size of Plan

- Errors
  - source include uniformity assumption, and inability to capture correlation, accuracy of cost model, statistics, etc.
  - *propagated to other operators at the higher level of the plan tree*



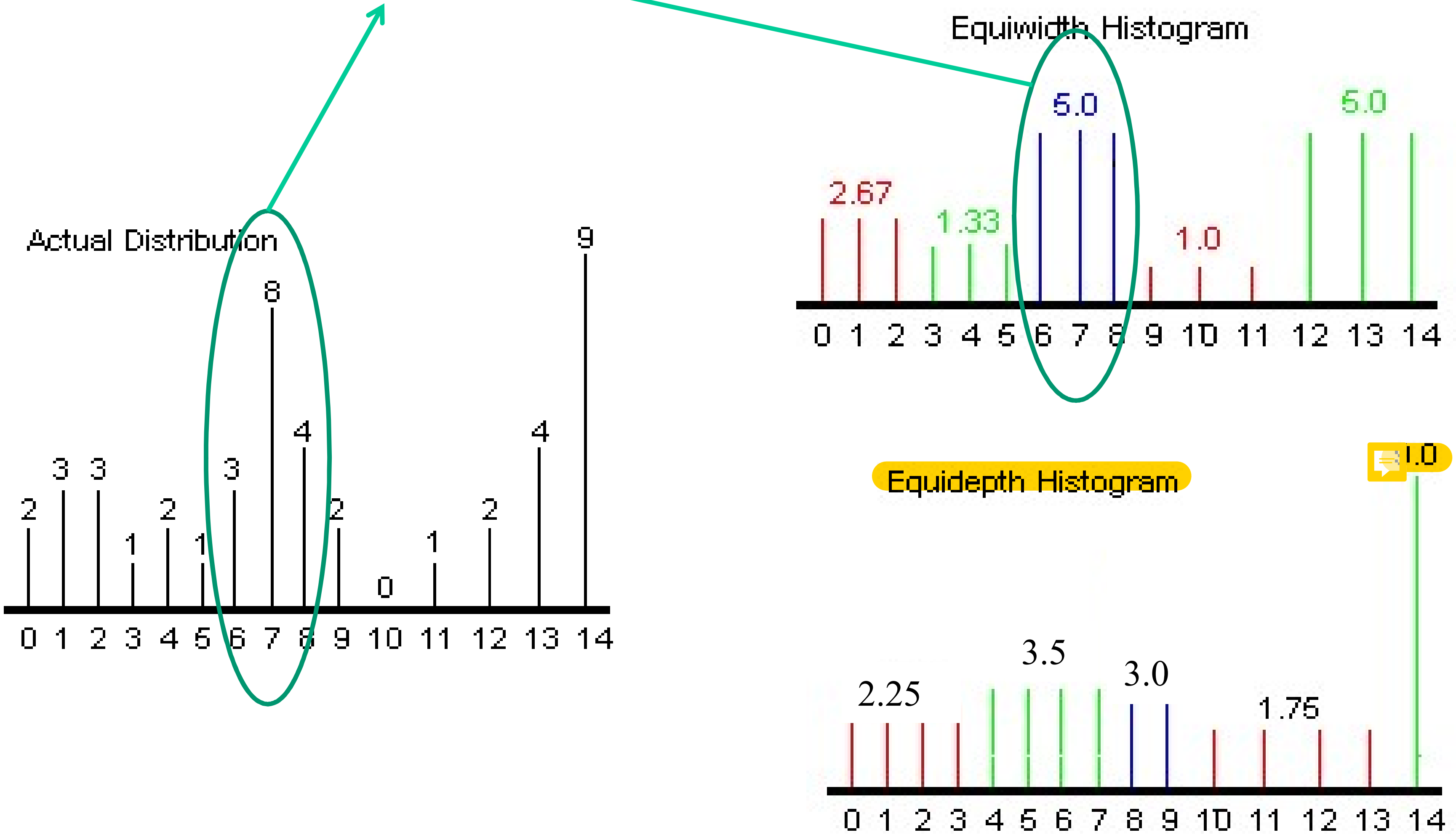

- Dealing with errors
  - Maintain more detailed statistics (at finer granularity)
  - During runtime, may need to sample the actual intermediate results
    - *dynamic* query optimization

# *Statistical Summaries of Data*

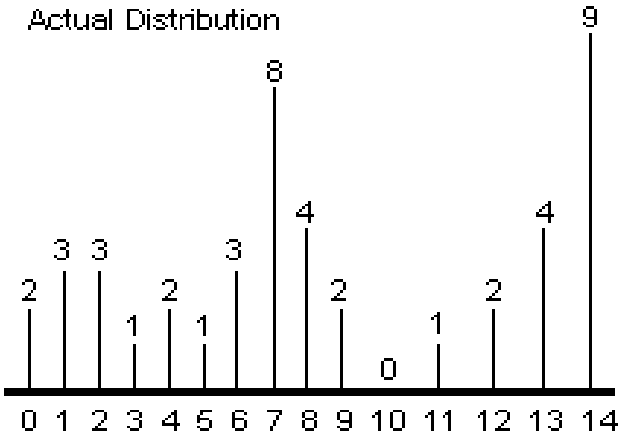
- More detailed information are sometimes stored e.g., histograms of the values in some attributes
  - a histogram divides the values on a column into k buckets
    - k is predetermined or computed based on space allocation
  - several choices for “bucketization” of values
    - If a table has n records, an equi-depth histogram divides the set of values on a column into k ranges such that each range has *approximately* the same number of records, i.e., n/k
    - Equi-width histogram – each bucket has (almost) equal number of values
    - Witihin each bucket, records are uniformly distributed across the range of the bucket
    - Frequently occurring values may be placed in singleton buckets

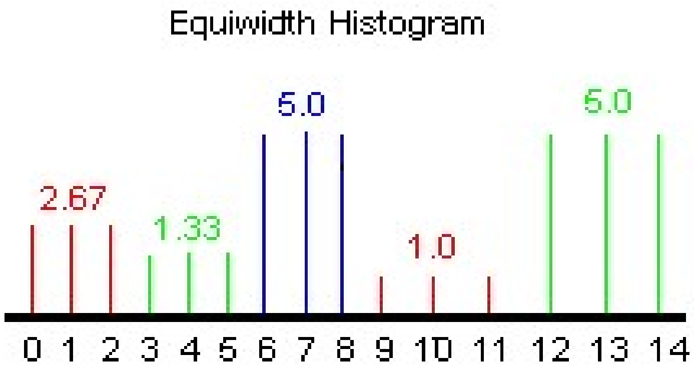
# *Histograms*

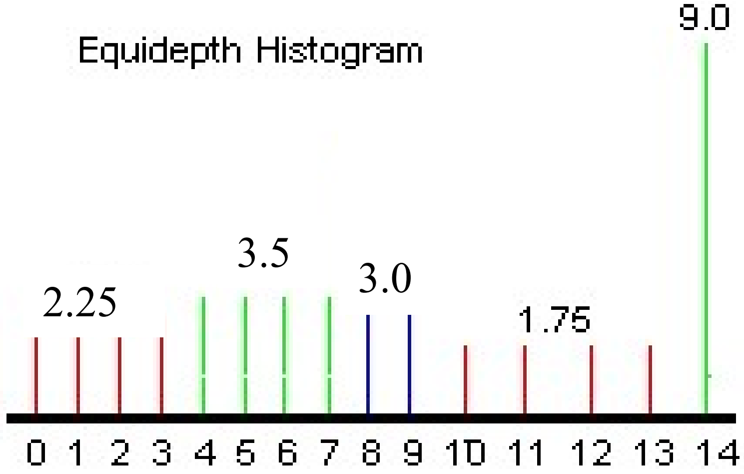
15 records are uniformly distributed



# *Estimations with Histograms*




Query Q: $\sigma_{A=6}$ (R)

Actual value, $||Q|| = 3$
Without histogram, $||Q|| = 45/15 = 3$
Equiwidth histogram, $||Q|| = 15/3 = 5$
Equidepth histogram, $||Q|| = 14/4 = 3.5$

Query Q: $\sigma_{A=10}$ (R)

Actual value, $||Q|| = 0$
Without histogram, $||Q|| = 45/15 = 3$
Equiwidth histogram, $||Q|| = 1$
Equidepth histogram, $||Q|| = 1.75$

# *Summary*

- Query optimization is NP-hard
- Instead of finding the best, the objective is largely to avoid the bad plans
- Many different optimization strategies have been proposed
  - greedy heuristics are fast but may generate plans that are far from optimal
  - dynamic programming is effective at the expense of high optimization overhead