Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#15: Query Optimization

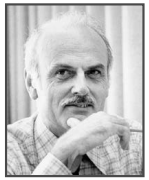
Today's Class

- History & Background
- Relational Algebra Equivalences
- Plan Cost Estimation
- Plan Enumeration

Query Optimization

- Remember that SQL is declarative.
  - User tells the DBMS *what* answer they want, not *how* to get the answer.
- There can be a big difference in performance based on plan is used:
  - See last week: 5.7 days vs. 45 seconds

1970s – Relational Model


Codd

- Ted Codd saw the maintenance overhead for IMS/Codasyl.
- Proposed database abstraction based on relations:
  - Store database in simple data structures.
  - Access it through high-level language.
  - Physical storage left up to implementation.

IBM System R

- Skunkworks project at IBM Research in San Jose to implement Codd's ideas.
- Had to figure out all of the things that we are discussing in this course themselves.
- IBM never commercialized System R.

IBM System R

- First implementation of a query optimizer.
- People argued that the DBMS could never choose a query plan better than what a human could write.
- A lot of the concepts from System R's optimizer are still used today.

Today's Class

- History & Background
- Relational Algebra Equivalences
- Plan Cost Estimation
- Plan Enumeration
- Nested Sub-queries

Relational Algebra Equivalences

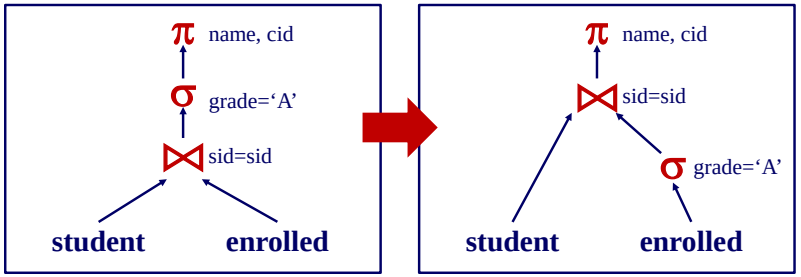
- A query can be expressed in different ways.
- The optimizer considers variations and choose the one with the lowest cost.
- How do we know whether two queries are equivalent?

Relational Algebra Equivalences

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

Predicate Pushdown

```
SELECT name, cid
  FROM student, enrolled
 WHERE student.sid =
       enrolled.sid
   AND enrolled.grade = 'A'
```



Relational Algebra Equivalences

```
SELECT name, cid
  FROM student, enrolled
 WHERE student.sid =
       enrolled.sid
   AND enrolled.grade = 'A'
```

$$\pi_{\text{name, cid}}(\sigma_{\text{grade='A'}}(\text{student} \bowtie \text{enrolled}))$$

$$=$$

$$\pi_{\text{name, cid}}(\text{student} \bowtie (\sigma_{\text{grade='A'}}(\text{enrolled})))$$

Relational Algebra Equivalences

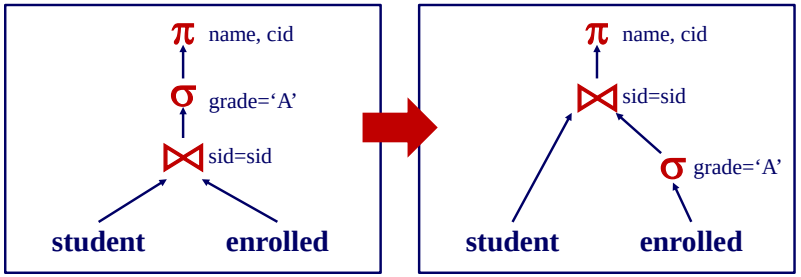
- **Selections:**
  - Perform them early
  - Break a complex predicate, and push down
    $\sigma_{p1 \wedge p2 \wedge \ldots pn}(R) = \sigma_{p1}(\sigma_{p2}(\sigma \ldots_{pn}(R)) \ldots)$
- Simplify a complex predicate
  - (`X=Y AND Y=3`) → `X=3 AND Y=3`

Relational Algebra Equivalences

- **Projections:**
  - Perform them early
    - Smaller tuples
    - Fewer tuples (if duplicates are eliminated)
  - Project out all attributes except the ones requested or required (e.g., joining attr.)
- *This is <u>not</u> important for a column store…*

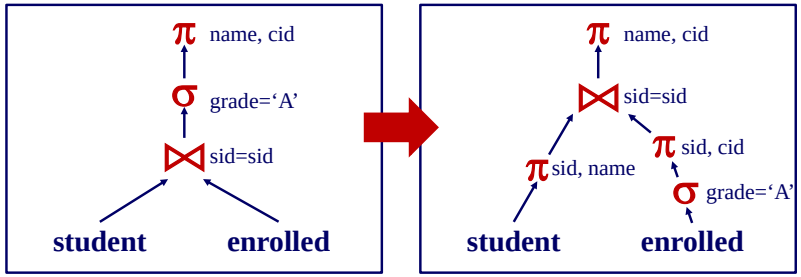
Projection Pushdown

```
SELECT name, cid
  FROM student, enrolled
 WHERE student.sid =
       enrolled.sid
   AND enrolled.grade = 'A'
```



Projection Pushdown

```
SELECT name, cid
  FROM student, enrolled
 WHERE student.sid =
       enrolled.sid
   AND enrolled.grade = 'A'
```



Relational Algebra Equivalences

- **Joins:**
  - Commutative, associative

$$R \bowtie S = S \bowtie R$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

- Q: How many different orderings are there for an *n*-way join?

Relational Algebra Equivalences

- **Joins:** How many different orderings are there for an n-way join?
- A: Catalan number ~ 4^n
  - Exhaustive enumeration: too slow.
- We'll see in a second how an optimizer limits the search space...

Today's Class

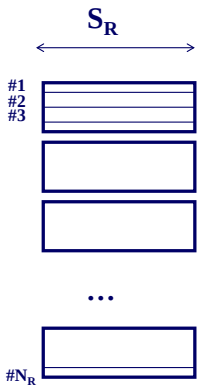
- History & Background
- Relational Algebra Equivalences
- Plan Cost Estimation
- Plan Enumeration

Cost Estimation

- How long will a query take?
  - **CPU**: Small cost; tough to estimate
  - **Disk**: # of block transfers
  - **Memory**: Amount of DRAM used
  - **Network**: # of messages
- How many tuples will be read/written?
- What statistics do we need to keep?

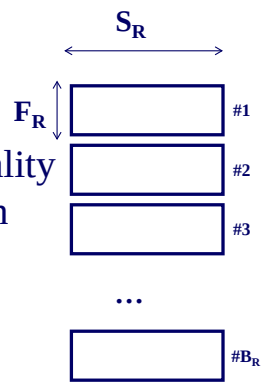
Cost Estimation – Statistics

- For each relation **R** we keep:
  - N_R → # tuples
  - S_R → size of tuple in bytes
  - **V(A,R)** → # of distinct values of attribute 'A'



Derivable Statistics

- F_R → max# records/block
- B_R → # blocks
- **SC(A,R)** → selection cardinality
  avg# of records with A=given

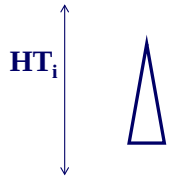


Derivable Statistics

- **SC(A,R)** → Selection Cardinality
  avg# of records with A=given
  → N_R / **V(A,R)**
- Note that this assumes data uniformity
  - 10,000 students, 10 colleges – how many students in SCS?

Additional Statistics

- For index **i**:
  - F_i → average fanout (~50-100)
  - HT_i → # levels of index **i** (~2-3)
    ~ log(#entries)/log(F_i)
  - LB_i # → blocks at leaf level



Statistics

- Where do we store them?
- How often do we update them?

- Manual invocations:
  - Postgres/SQLite: `ANALYZE`
  - MySQL: `ANALYZE TABLE`

Selection Statistics

- We saw simple predicates (`name="Kayne"`)
- How about more complex predicates, like
  - `salary > 10000`
  - `age=30 AND jobTitle="Costermonger"`
- What is their selectivity?

Selections – Complex Predicates

- Selectivity **sel(P)** of predicate **P**:
  == fraction of tuples that qualify
- Formula depends on type of predicate.
  - Equality
  - Range
  - Negation
  - Conjunction
  - Disjunction

Selections – Complex Predicates

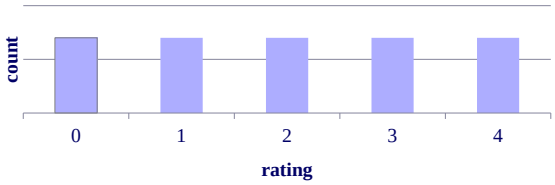
- Selectivity **sel(P)** of predicate **P**:
  == fraction of tuples that qualify
- Formula depends on type of predicate.
  - Equality
  - Range
  - Negation
  - Conjunction
  - Disjunction

Selections – Complex Predicates

- Assume that **V(rating, sailors)** has 5 distinct values (0–4) and $\mathbf{N_R} = 5$
- **Equality Predicate**: A=constant
  - **sel(A=constant)** = **SC(P) / V(A,R)**
  - Example: **sel(rating='2') =**

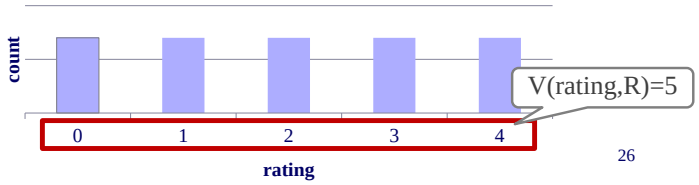
Selections – Complex Predicates

- Assume that **V(rating, sailors)** has 5 distinct values (0–4) and $\mathbf{N_R}$ = 5
- **Equality Predicate**: A=constant
  - **sel(A=constant)** = **SC(P) / V(A,R)**
  - Example: **sel(rating=‘2’) =**



Selections – Complex Predicates

- Assume that **V(rating, sailors)** has 5 distinct values (0–4) and $\mathbf{N_R}$ = 5
- **Equality Predicate**: A=constant
  - **sel(A=constant)** = **SC(P) / V(A,R)**
  - Example: **sel(rating=‘2’) =**

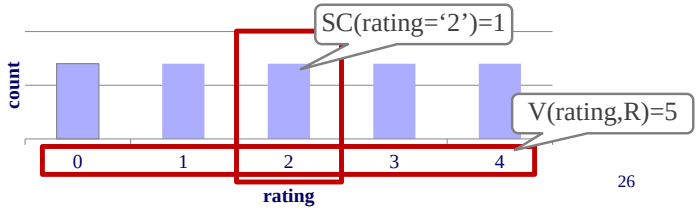


Selections – Complex Predicates

- Assume that **V(rating, sailors)** has 5 distinct values (0–4) and $\mathbf{N_R}$ = 5
- **Equality Predicate**: A=constant
  - **sel(A=constant)** = **SC(P) / V(A,R)**
  - Example: **sel(rating=‘2’) =**

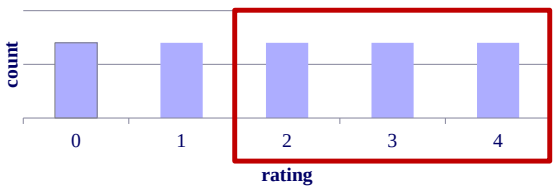
Selections – Complex Predicates

- Assume that **V(rating, sailors)** has 5 distinct values (0–4) and $\mathbf{N_R}$ = 5
- **Equality Predicate**: A=constant
  - **sel(A=constant)** = **SC(P) / V(A,R)**
  - Example: **sel(rating=‘2’) = 1/5**



Selections – Complex Predicates

- **Range Query:**
  - **sel(**A>a**)** = **(A_{max} – a) / (A_{max} – A_{min})**
  - Example: **sel(rating >= ‘2’)**



Selections – Complex Predicates

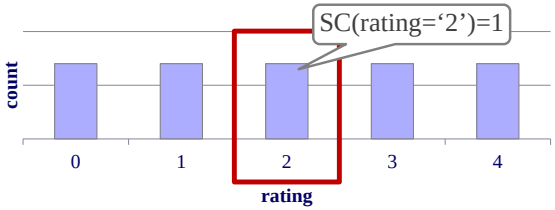
- **Range Query:**
  - **sel(**A>a**)** = **(A_{max} – a) / (A_{max} – A_{min})**
  - Example: **sel(rating >= ‘2’) = (4 – 2) / (4 – 0)**
    = 1/2

Selections – Complex Predicates

- **Negation Query**
  - **sel(not** P**) = 1 – sel(**P**)**
  - Example: **sel(rating != ‘2’)**

Selections – Complex Predicates

- **Negation Query**
  - **sel(not** P**) = 1 – sel(**P**)**
  - Example: **sel(rating != ‘2’)**



Selections – Complex Predicates

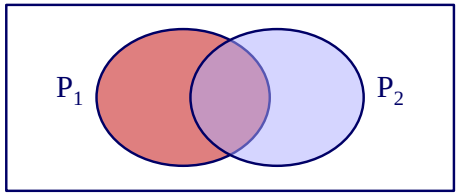
- **Negation Query**
  - **sel(not** P**) = 1 – sel(**P**)**
  - Example: **sel(rating != '2')**

Selections – Complex Predicates

- **Negation Query**
  - **sel(not** P**) = 1 – sel(**P**)**
  - Example: **sel(rating != '2') = 1 – (1/5) = 4/5**
- Observation: selectivity ≈ probability

Selections – Complex Predicates

- **Conjunction:**
  - **sel(**rating = '2' **AND** name LIKE 'C%'**)**
  - **sel(**$P_1 \wedge P2$**) = sel(**P_1**) · sel(**P_2**)**
  - INDEPENDENCE ASSUMPTION

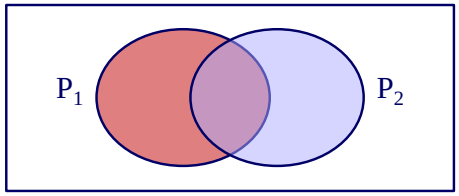


Selections – Complex Predicates

- **Disjunction**:
  - **sel(**rating = '2' **OR** name LIKE 'C%'**)**
  - **sel(**$P_1 \vee P_2$**)**
    $= \mathbf{sel}(P_1) + \mathbf{sel}(P_2) - \mathbf{sel}(P_1 \vee P_2)$
    $= \mathbf{sel}(P_1) + \mathbf{sel}(P_2) - \mathbf{sel}(P_1) \cdot \mathbf{sel}(P_2)$
  - INDEPENDENCE ASSUMPTION, again

Selections – Complex Predicates

- **Disjunction, in general**:
  - **sel**(P_1 **OR** P_2 **OR** … P_n) =
  - 1 - (1- **sel**(P_1)) · (1 - **sel**(P_2)) · … (1 - **sel**(P_n))



Joins

- Q: Given a join of **R** and **S**, what is the range of possible result sizes in #of tuples?

Result Size Estimation for Joins

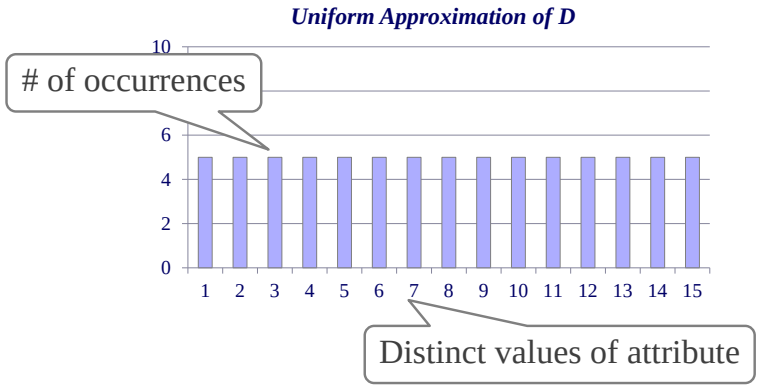
- General case: $\mathbf{R_{cols}} \cap \mathbf{S_{cols}} = \{A\}$ where A is not a key for either table.
- *Hint: for a given tuple of R, how many tuples of S will it match?*

Result Size Estimation for Joins

- General case: $\mathbf{R_{cols}} \cap \mathbf{S_{cols}} = \{A\}$ where A is not a key for either table.
  - Match each **R**-tuple with **S**-tuples:
    estSize ≈ $\mathbf{N_R \cdot N_S / V(A,S)}$
  - Symmetrically, for **S**:
    estSize ≈ $\mathbf{N_R \cdot N_S / V(A,R)}$
- Overall:
  - estSize ≈ $\mathbf{N_R \cdot N_S / \max(\{V(A,S), V(A,R)\})}$

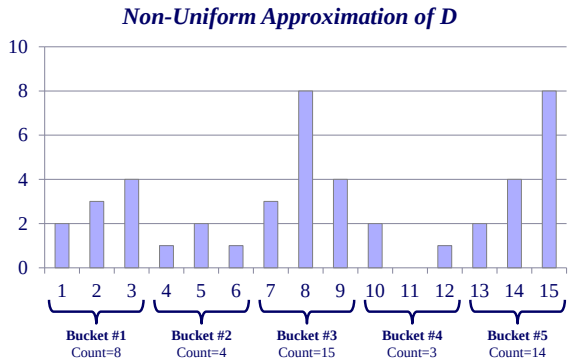
Cost Estimations

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



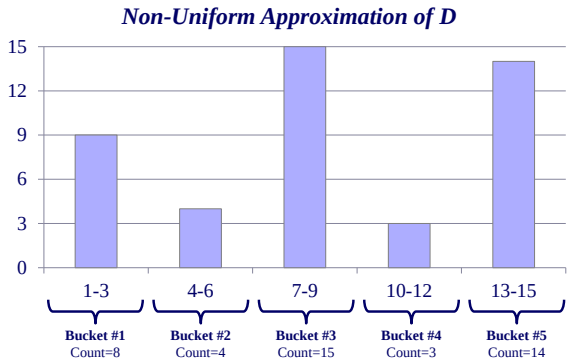
Cost Estimations

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



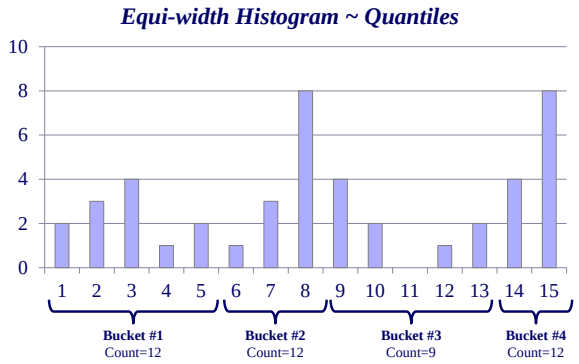
Cost Estimations

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



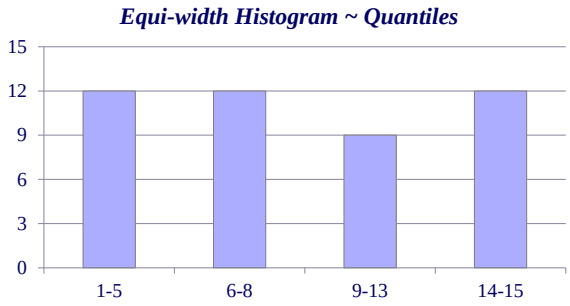
Histograms with Quantiles

- A histogram type wherein the “spread” of each bucket is same.



Histograms with Quantiles

- A histogram type wherein the "spread" of each bucket is same.



Sampling

- Modern DBMSs also employ sampling to estimate predicate selectivities.

```
SELECT * FROM sailors
 WHERE rating > 100
```

sid	sname	rating	age
1	Kayne	999	45.0
3	Obama	50	52.0
2	Tupac	32	26.0
6	Bieber	10	19.0

⋮

1 billion tuples

Sampling

- Modern DBMSs also employ sampling to estimate predicate selectivities.

```
SELECT * FROM sailors
 WHERE rating > 100
```

sel(rating>100**)** =

Kayne	999
Tupac	32
Bieber	10

= 1/3

sid	sname	rating	age
1	Kayne	999	45.0
3	Obama	50	52.0
2	Tupac	32	26.0
6	Bieber	10	19.0

⋮

1 billion tuples

Today's Class

- History & Background
- Relational Algebra Equivalences
- Plan Cost Estimation
- Plan Enumeration

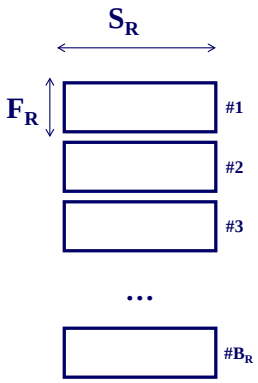
Query Optimization

- Bring query in internal form into "canonical form" (syntactic q-opt)
- Generate alternative plans.
  - Single relation.
  - Multiple relations.
  - Nested sub-queries.
- Estimate cost for each plan.
- Pick the best one.

Plan Generation

```
SELECT *
  FROM sailors
WHERE rating = 2
```

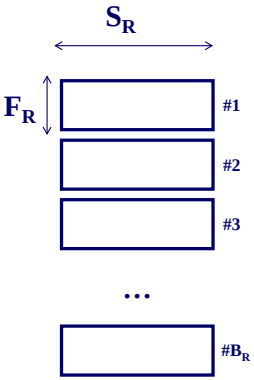
- What are our plan options?



Plan Generation

```
SELECT *
  FROM sailors
WHERE rating = 2
```

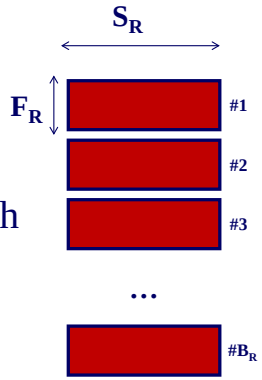
- Sequential Scan
- Binary Search
  - *if sorted & consecutive*
- Index Search
  - *if an index exists*



Sequential Scan

```
SELECT *
  FROM sailors
WHERE rating = 2
```

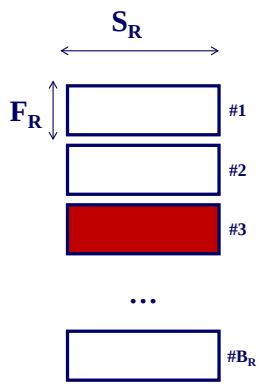
- **B_R** (worst case)
- **B_R /2** (on average, if we search for primary key)



Binary Search

```
SELECT *
  FROM sailors
WHERE rating = 2
```

- **~log(B_R) + SC(A,R)/ F_R**
- Extra blocks are ones that contain qualifying tuples

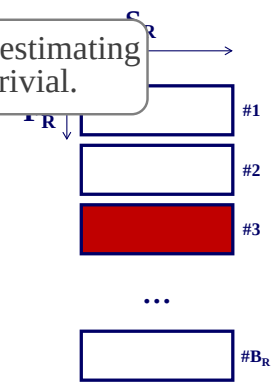


Binary Search

```
SELECT *
  FROM sailors
WHERE rating = 2
```

We showed that estimating this is non-trivial.

- **~log(B_R) + SC(A,R)/ F_R**
- Extra blocks are ones that contain qualifying tuples

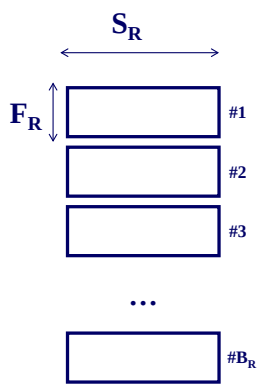


Index Search

```
SELECT *
  FROM sailors
WHERE rating = 2
```

- Index Search:
  - levels of index + blocks w/ qual. tuples

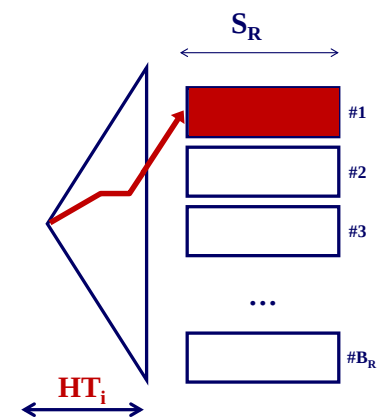
Case#1: Primary Key
Case#2: Secondary key – clustering index
Case#3: Secondary key – non-clust. index



Index Search: Case #1

```
SELECT *
  FROM sailors
WHERE rating = 2
```

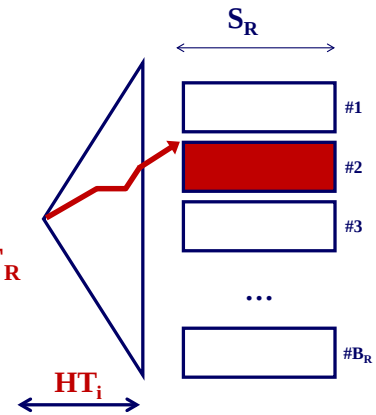
- Primary Key
  - **cost: HT_i + 1**



Index Search: Case #2

```
SELECT *
  FROM sailors
WHERE rating = 2
```

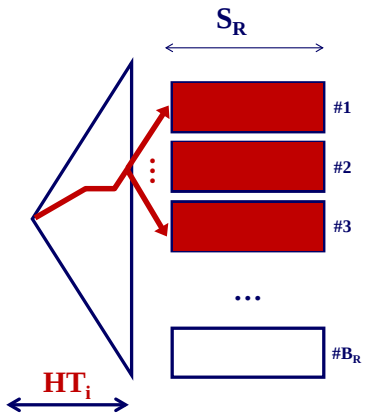
- Secondary key with clustering index:
  - **cost: $HT_i + SC(A,R)/F_R$**



Index Search: Case #3

```
SELECT *
  FROM sailors
WHERE rating = 2
```

- Secondary key with non-clustering index:
  - **cost: $HT_i + SC(A,R)$**



Query Optimization

- Bring query in internal form into "canonical form" (syntactic q-opt)
- Generate alternative plans.
  - Single relation.
  - Multiple relations.
  - Nested sub-queries.
- Estimate cost for each plan.
- Pick the best one.

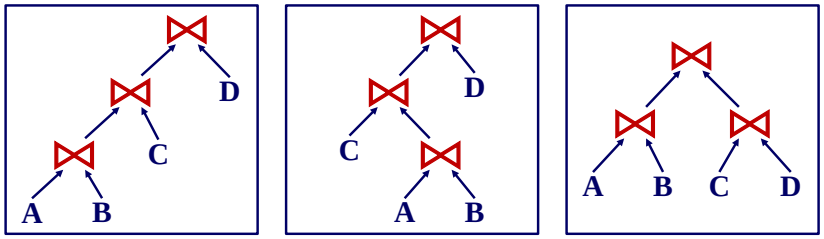
Queries over Multiple Relations

- As number of joins increases, number of alternative plans grows rapidly
  - We need to restrict search space.
- **Fundamental decision in System R:** only left-deep join trees are considered.

Newer DBMSs don't make this assumption anymore

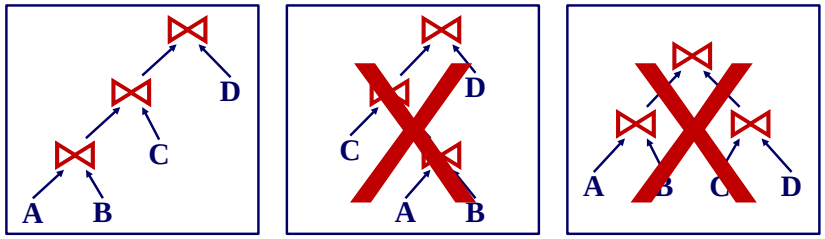
Queries over Multiple Relations

- **Fundamental decision in System R:** only left-deep join trees are considered.



Queries over Multiple Relations

- **Fundamental decision in System R:** only left-deep join trees are considered.



Queries over Multiple Relations

- **Fundamental decision in System R:** only left-deep join trees are considered.
  - Allows for fully pipelined plans where intermediate results not written to temp files.
  - Not all left-deep trees are fully pipelined.

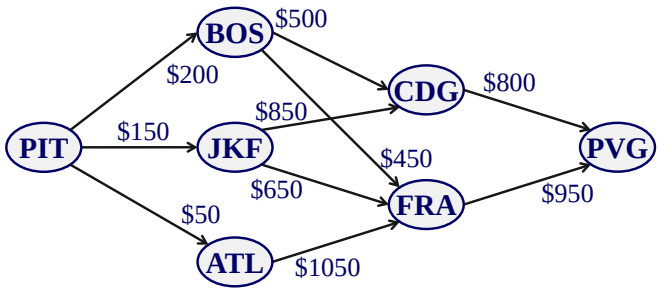
Queries over Multiple Relations

- **Enumerate the orderings**
  - *Example: Left-deep tree #1, Left-deep tree #2…*
- **Enumerate the plans for each operator**
  - *Example: Hash, Sort-Merge, Nested Loop…*
- **Enumerate the access paths for each table**
  - *Example: Index #1, Index #2, Seq Scan…*

Queries over Multiple Relations

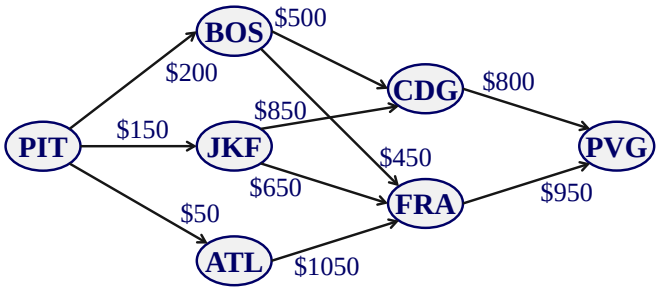
- **Enumerate the orderings**
  - *Example: Left-deep tree #1, Left-deep tree #2…*
- **Enumerate the plans for each operator**
  - *Example: Hash, Sort-Merge, Nested Loop…*
- **Enumerate the access paths for each table**
  - *Example: Index #1, Index #2, Seq Scan…*
- Use **dynamic programming** to reduce the number of cost estimations.

Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

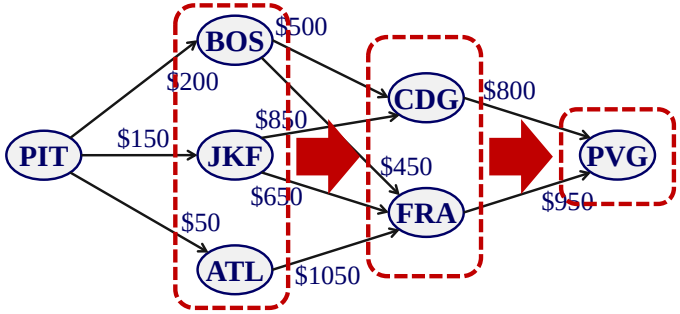
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

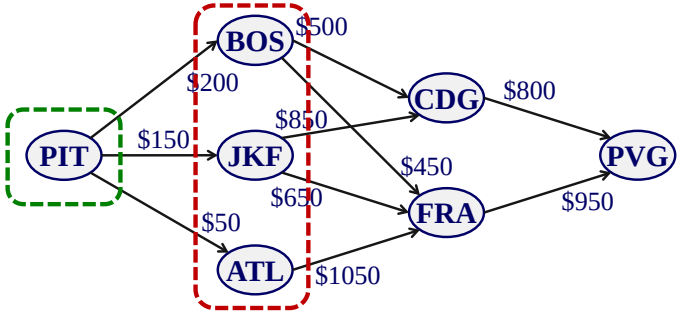
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

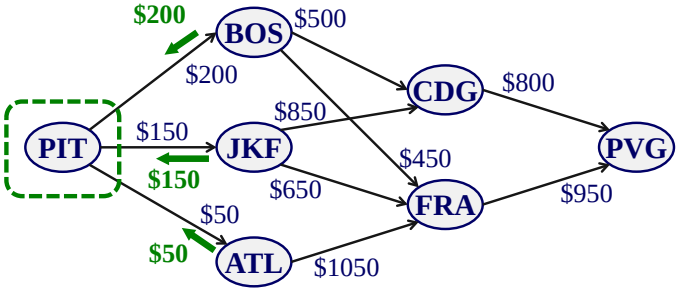
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

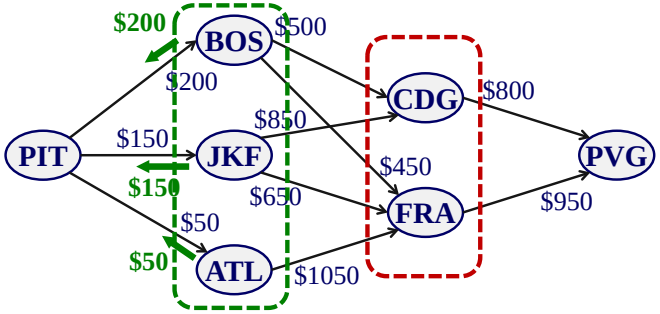
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

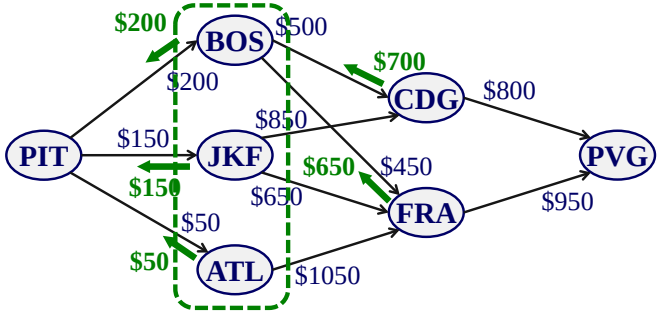
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

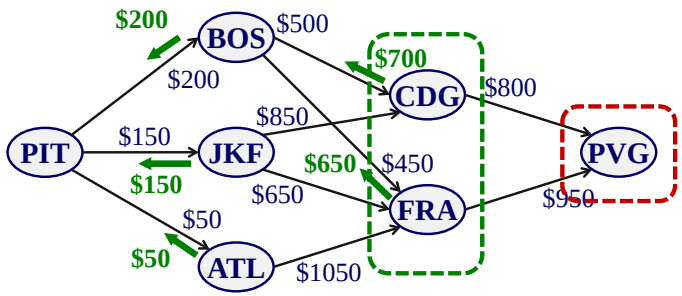
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

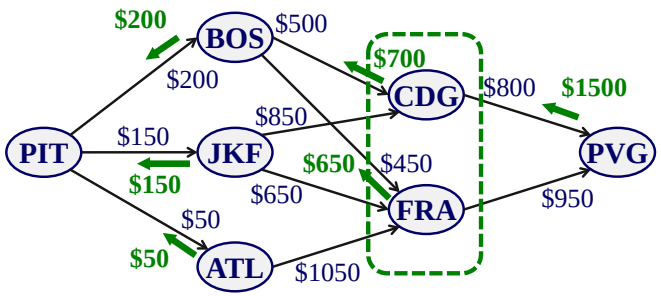
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

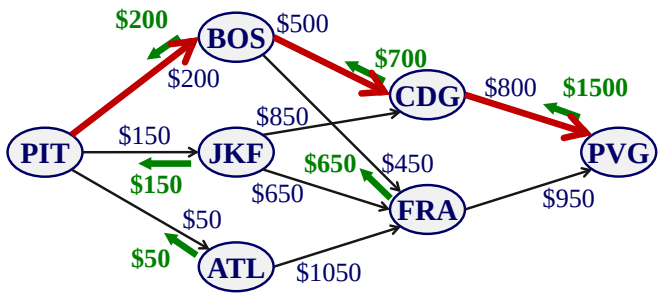
Dynamic Programming Example



Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

Dynamic Programming Example

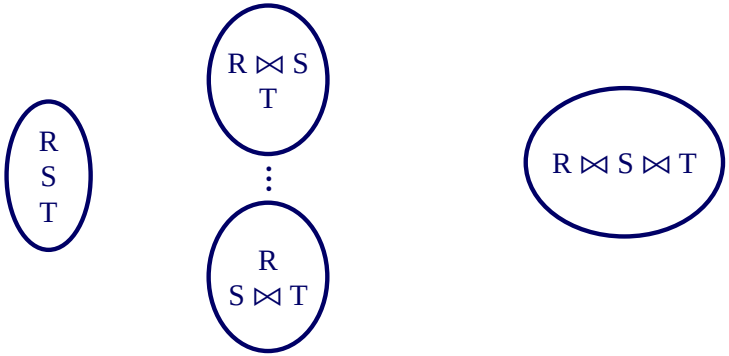


Compute the cheapest flight PIT -> PVG

Solution: Compute partial optimal, left-to-right

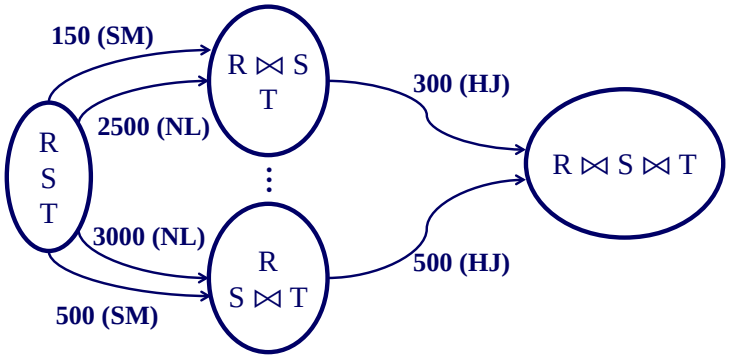
Q-Opt + Dynamic Programming

- Example: R ⋈ S ⋈ T



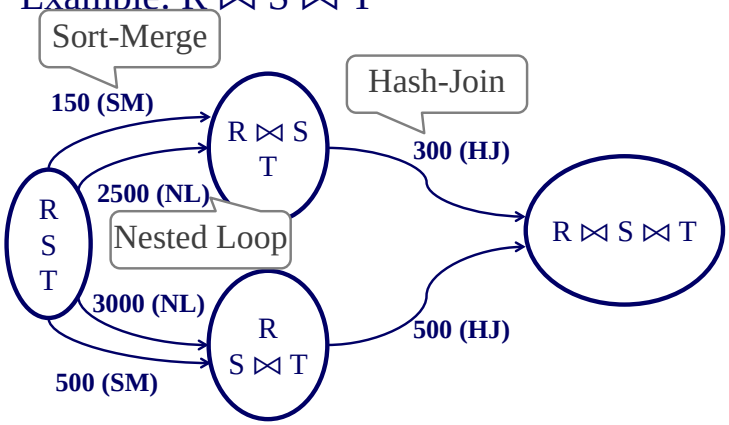
Q-Opt + Dynamic Programming

- Example: R ⋈ S ⋈ T



Q-Opt + Dynamic Programming

- Example: R ⋈ S ⋈ T



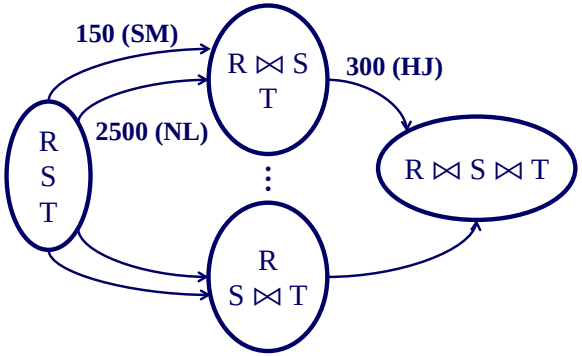
Q-Opt + Dynamic Programming

- How to plan a query where **R** is sorted on **R.a**?
- Consider the following query:

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.a AND S.b = T.b
 ORDER BY R.a
```

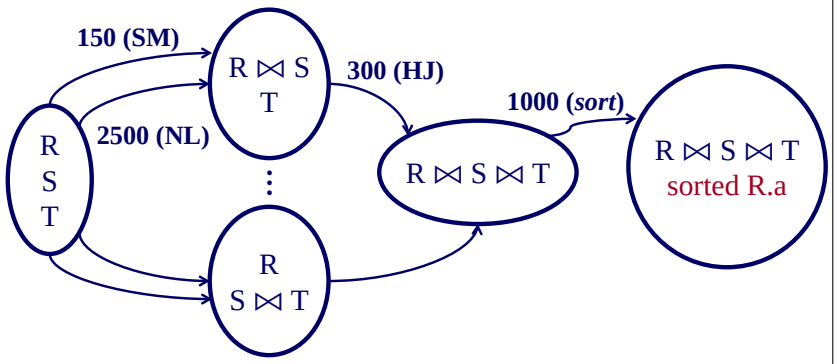

Q-Opt + Dynamic Programming

- R ⋈ S ⋈ T order by R.a



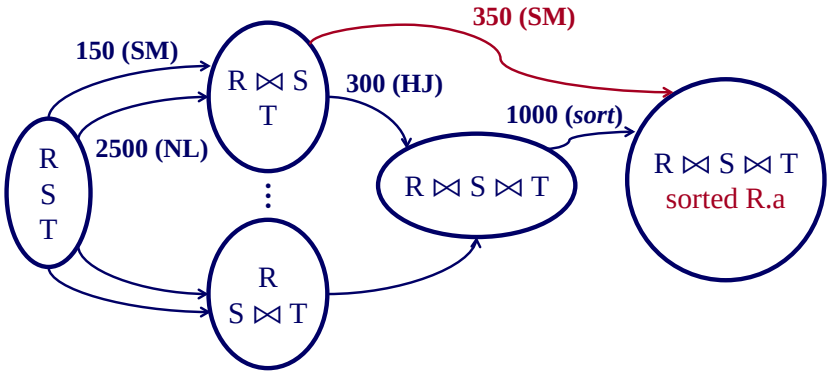
Q-Opt + Dynamic Programming

- R ⋈ S ⋈ T order by R.a



Q-Opt + Dynamic Programming

- R ⋈ S ⋈ T order by R.a



Candidate Plan Example

```
SELECT sname, bname, day
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

- How to generate plans for search algorithm:
  1. Enumerate relation orderings
  2. Enumerate join algorithm choices
  3. Enumerate access method choices

Candidate Plan Example

```
SELECT sname, bname, day
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

No real DBMSs does it this way. It's actually more messy…

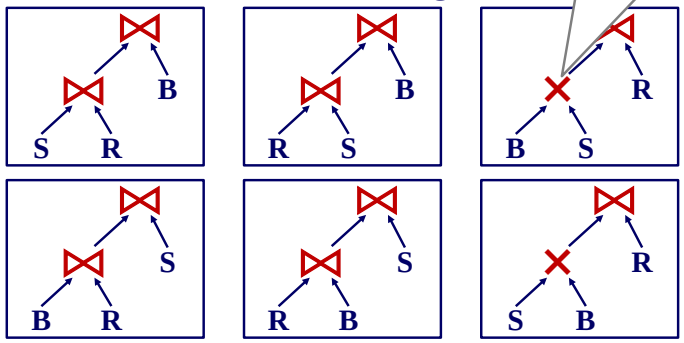
- How to generate plans for search algorithm:
  1. Enumerate relation orderings
  2. Enumerate join algorithm choices
  3. Enumerate access method choices

Candidate Plans

```
SELECT sname, bname, day
  FROM sailors S, reserves R, b
 WHERE S.sid = R.sid AND R.bid
```

Prune plans with cross-products immediately!

1. Enumerate relation orderings:

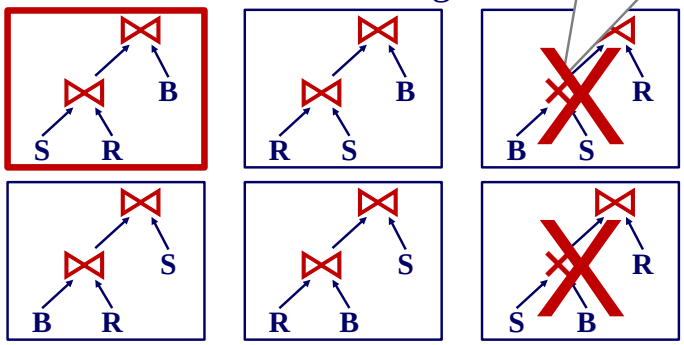


Candidate Plans

```
SELECT sname, bname, day
  FROM sailors S, reserves R, b
 WHERE S.sid = R.sid AND R.bid
```

Prune plans with cross-products immediately!

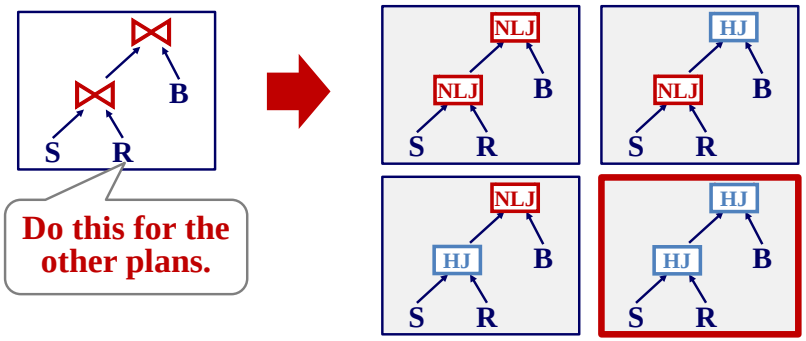
1. Enumerate relation orderings:



Candidate Plans

```
SELECT sname, bname, day
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

2. Enumerate join algorithm choices:

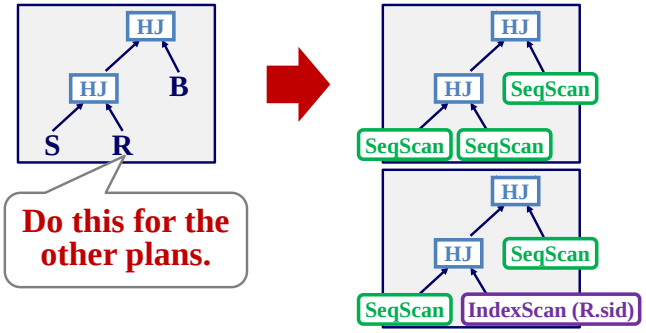


Do this for the other plans.

Candidate Plans

```
SELECT sname, bname, day
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid AND R.bid = B.bid
```

3. Enumerate access method choices:

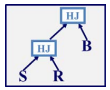


Do this for the other plans.

Postgres Optimizer

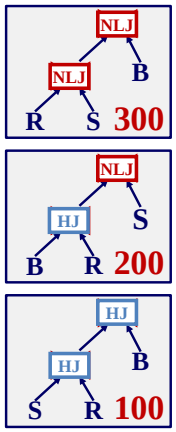
- Examines all types of join trees
  - Left-deep, Right-deep, bushy
- Two optimizer implementations:
  - Traditional Dynamic Programming Approach
  - Genetic Query Optimizer (GEQO)
- Postgres uses the traditional one when # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more.

Postgres GEQO

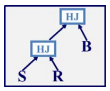


Best: 100

1st Generation

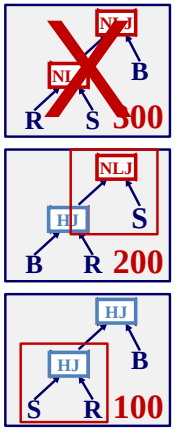


Postgres GEQO

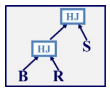


Best: 100

1st Generation

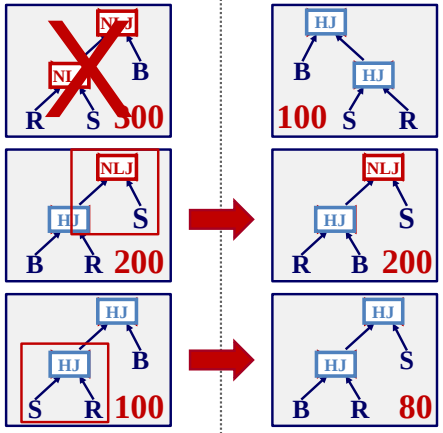


Postgres GEQO

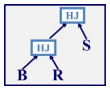


Best: 80

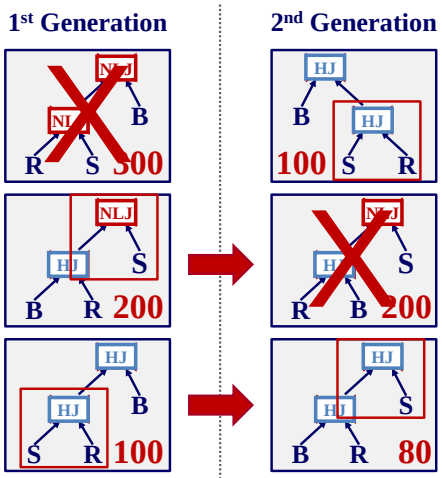
1st Generation **2nd Generation**



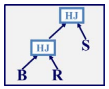
Postgres GEQO



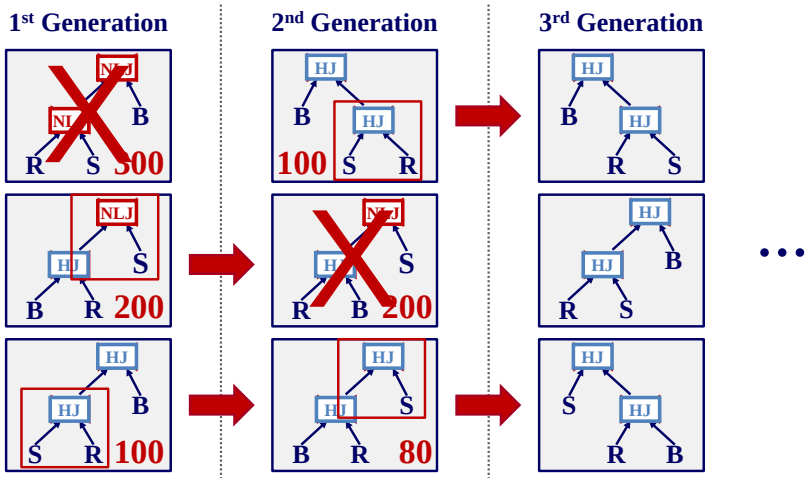
Best: 80



Postgres GEQO



Best: 80



Query Optimization

- Bring query in internal form into "canonical form" (syntactic q-opt)
- Generate alternative plans.
  - Single relation.
  - Multiple relations.
  - Nested sub-queries.
- Estimate cost for each plan.
- Pick the best one.

Nested Sub-Queries

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

Nested Sub-Queries: Rewrite

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
      WHERE S.sid = R.sid
        AND R.day = "2016-10-24"
 )
```


Nested Sub-Queries: Rewrite

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
      WHERE S.sid = R.sid
        AND R.day = "2016-10-24"
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = "2016-10-23"
```


Nested Sub-Queries: Decompose

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                   FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

Nested Sub-Queries: Decompose

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                   FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

Decomposing Queries

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

Decomposing Queries

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                   FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

Decomposing Queries

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                   FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

Decomposing Queries

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Decomposing Queries

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

What Optimizers are Still Bad At

- Cardinality estimations are still hard.
- Problem Areas:
  - Prepared Statements
  - Correlated Columns
  - Plan Stability

More Information: Guy Lohman, *Is Query Optimization a "Solved" Problem?* SIGMOD Blog (April 2014), http://wp.sigmod.org/?p=1075

Prepared Statements

```
SELECT S.name, B.bid
  FROM sailors AS S,
       reserves AS R, boats AS B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating > 1000
```


Prepared Statements

```
PREPARE myQuery AS
SELECT S.name, B.bid
  FROM sailors AS S,
       reserves AS R, boats AS B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating > 1000
```

Prepared Statements

```
PREPARE myQuery AS
SELECT S.name, B.bid
  FROM sailors AS S,
       reserves AS R, boats AS B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating > 1000
```

```
EXECUTE myQuery;
```


Prepared Statements

```
PREPARE myQuery AS
SELECT S.name, B.bid
  FROM sailors AS S,
       reserves AS R, boats AS B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating > 1000
```

```
EXECUTE myQuery;
```

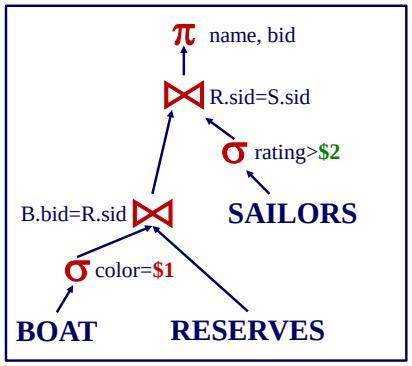

Prepared Statements

```
PREPARE myQuery (varchar, int) AS
SELECT S.name, B.bid
  FROM sailors AS S,
       reserves AS R, boats AS B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = $1
   AND S.rating > $2
```

```
EXECUTE myQuery ('red', 1000);
```


Prepared Statements

Prepared Statement Query Plan



- What should be the join order for **SAILORS**, **BOATS**, and **RESERVES?**

Prepared Statements

- **Solution #1 –** Rerun optimizer each time the query is invoked.
- **Solution #2 –** Generate multiple plans for different values of the parameters.
- **Solution #3 –** Choose the average value for a parameter and use that for all invocations.

Correlated Columns

- We showed how selectivities are modeled as probabilities on whether a predicate on any given row will be satisfied.
- We then multiply these individual selectivities together.

Correlated Columns

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

Column Group Statistics

- Tell the DBMS that it should keep track of statistics for groups of columns together rather than just treating them all as independent variables.
- Only supported in commercial systems.

Plan Stability

- We want to deploy a new version of a DBMS but need to make sure that there are no performance regressions.
- What if 99% of the query plans are faster on the newer DBMS version, but 1% are slower?

Plan Stability

- **Solution #1 –** Allow tuning hints in plans.
- **Solution #2 –** Set the optimizer version number and migrate queries one-by-one to the new optimizer.
- **Solution #3 –** Save query plan from old version and provide it to the new DBMS.

Conclusions

- Ideas to remember:
  - Filter early as possible.
  - Selectivity estimations (uniformity, indep.; histograms; join selectivity)
  - Dynamic programming for join orderings
  - Rewrite nested queries
- Query optimization is hard…

Next Class

- How to refine database schemas to remove redundancies and prevent loss data.