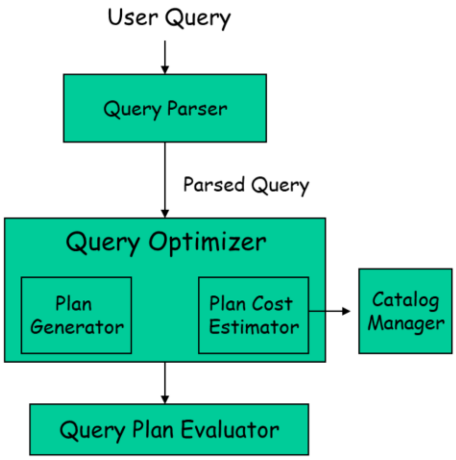
Overview of Query Evaluation

- A user query is expressed using SQL or other languages, ideally natural languages
- A parsed query is essentially treated as a relational algebra expression
  - Selection (б)
  - Join (⋈)
  - Project(Π)
  - Union, intersection and difference, cross product
- Query optimizer
  - Enumerates the possible plans to evaluate expression,
  - Select a small subset of these plans and estimate their cost



Query Blocks: Units of Optimization

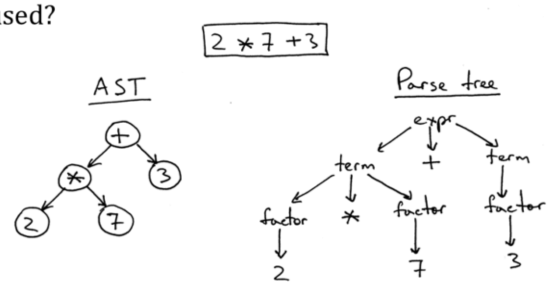
- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.
  - A query block is an SQL query with no nesting and exactly one SELECT clause and one FROM clause and at most one WHERE clause, etc.
- Nested blocks are usually treated as calls to a subroutine, 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*

Relational Algebra Tree and Evaluation Plan

- A plan consists of an extended relational algebra tree
  - Similar to a parse tree for an arithmetic expression
- Additional annotations are used to indicated the access and/or implementation method
  - Plan with/without indexes, etc.
- Problems
  - Which access method should be used for an operation?
  - Which plan should be used?



1) query block
2) each block is translated into a relational algbraic expression
3) each expresison is represented by a tree
4) each tree can have a number of execution plans because of relational algebra equivalence
5) The number of plans is large
6) Estimating each plan needs to sovle these issues: estimate the cost each operator and the output size
7) General strategy is 1) do selection as early as possible; 2) do projection as early as possible; 3) do join last
8) when do join, consider only left-deep plans

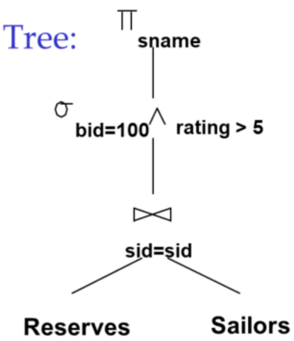
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserve(sid, bid, day)

Find the names of sailors who reserve a boat with bid = 100 and whose rating is greater than 5

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

$$\Pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserve \bowtie_{sid=sid} Sailors))$$

RA Tree:



A plan is a tree with annotations that specify the access method for each operator

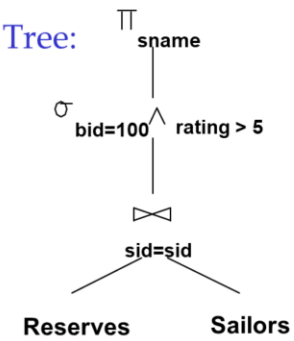
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserve(sid, bid, day)

Find the names of sailors who reserve a boat with bid = 100 and whose rating is greater than 5

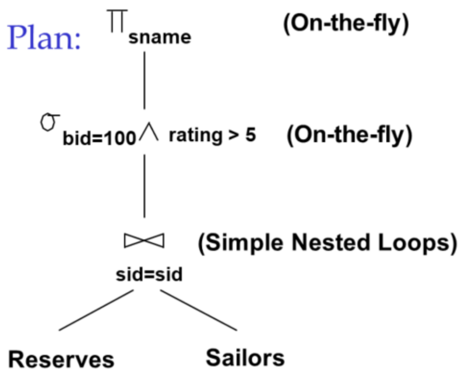
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

$$\Pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserve \bowtie_{sid=sid} Sailors))$$

RA Tree:

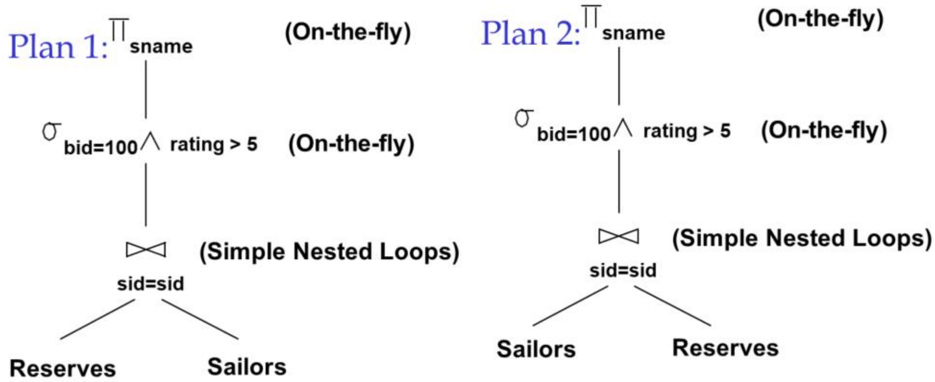


Plan:



There are usually many plans. Which one is the best? Optimization issue!

$$\Pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserve \bowtie_{sid=sid} Sailors))$$



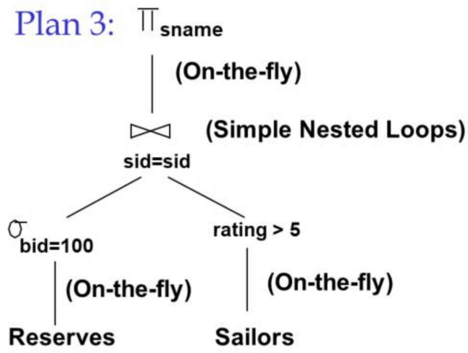
Question 1: are they the same?

Question 2: which one might be faster?

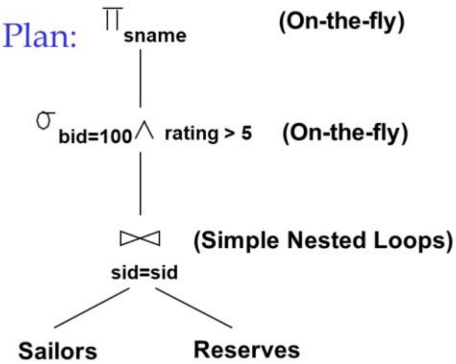
There are usually many plans. Which one is the best? Optimization issue!

$$\Pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserve \bowtie_{sid=sid} Sailors))$$

Plan 3:



Plan:



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: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

Question 1: think about what those symbols mean

Question 2: think about why they are the same

More Equivalences To Explore

$\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$

$\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$

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

Example: R(A, B, C, D), S(E, F, G)

$\sigma_{F=3}(R \bowtie_{D=E} S) =$?

$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$?

More Equivalences

- 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
- A selection on just attributes of R commutes with R $\bowtie$ S (i.e., σ (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

Challenges of Query Optimization

- Two main issues:
  - For a given query, what plans are considered?
    - Search space is huge
  - How to estimate the cost of a plan?
- Solutions
  - Ideally: find best plan
  - Practically: avoid worst plans

Highlights of IBM System R Optimizer (1979)

- Impact
  - Most widely used currently; works well for < 10 joins
- Cost estimation
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes
  - Considers combination of CPU and I/O costs
- Plan Space
  - 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

Cost Estimation

- Given a plan, we need to
  - Estimate *cost* of each operation in plan tree
    - Use the information recorded in statistics and system catalogs
    - Depends on input cardinalities
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Estimate *size of result* for each operation in tree
    - Use information about the input relations
    - For selections and joins, assume independence of predicates

System Catalogs

- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

Catalogs are themselves stored as relations!

Statistics stored in Catalog

- Cardinality
  - Number of tuples/rows NTuples(R) for each relation R
- Size
  - Number of pages NPages(R) for each relation R
- Index Cardinality
  - Number of distinct key values NKeys(I) for each index I
- Index Size
  - Number of pages INPages(I) for each index I
  - For example, for a B+ tree index, we take the number of leaf pages to be the index size
- Index Height
  - Number of nonleaf levels IHeight(I) for each tree index I
- Index Range
  - The minimum present key value ILow(I) and the maximum present key value IHigh(I) for each index I

These statistics are updated periodically

Size Estimation and Reduction Factors

- Consider a query block:

```
SELECT attribute list
FROM relation list
WHERE term1 AND ... AND termk
```

- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause
- Reduction factor (RF) associated with each term reflects the impact of the term in reducing result size
- Result cardinality = Max # tuples * product of all RF's.
  - Implicit assumption that terms are independent!
  - Term col=value has RF 1/NKeys(I), given index I on col
  - Term col1=col2 has RF 1/MAX(NKeys(I1), NKeys(I2))
  - Term col>value has RF (High(I)-value)/(High(I)-Low(I))

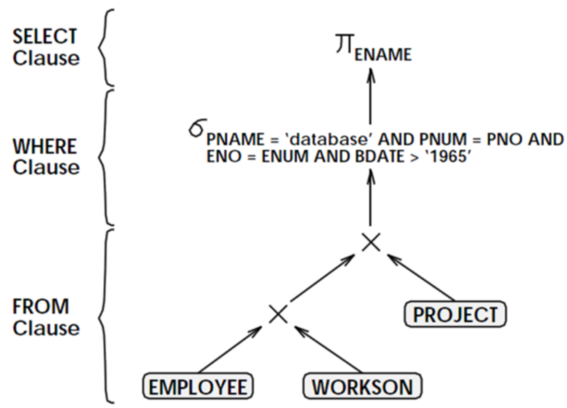
Optimization Strategies

- Move SELECTs and PROJECTS as far down as possible
- Among the SELECTs, order them such that the lowest selectivity factor is performed first
- Among Joins, order them such that the join with the lowest join selectivity factor is performed first

EMPLOYEE(ENUM, ENAME, BDATE)
PROJECT(PNUM, PNAME)
WORKSON(ENO, PNO) where ENO is a FK to Employee(ENUM) and PNO is a FK to PROJECT(PNUM)

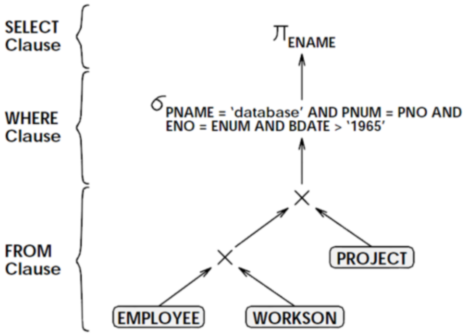
SELECT ENAME
FROM EMPLOYEE, PROJECT, WORKSON
WHERE PNAME=‘Database’
AND PNUM=PNO
AND ENUM=ENO
AND BDATE>’1965’

Canonical Query Tree



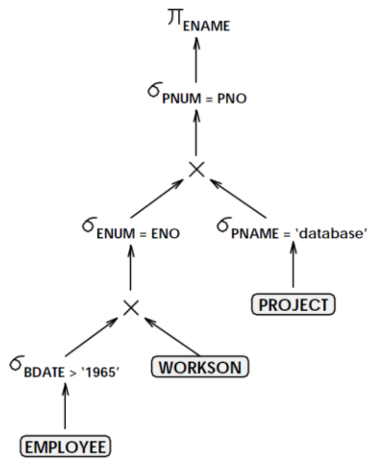
IBM’s System R

Canonical Query Tree



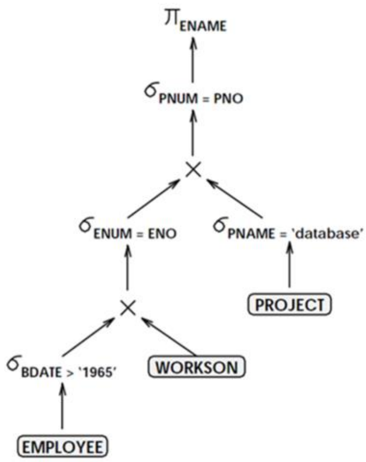
Move SELECTs down

After Optimization

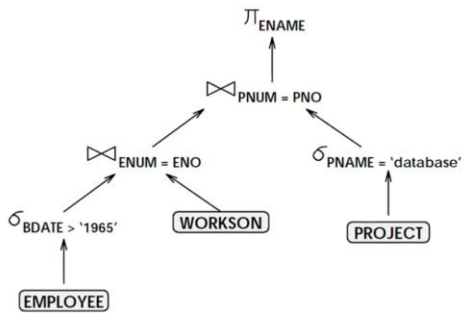


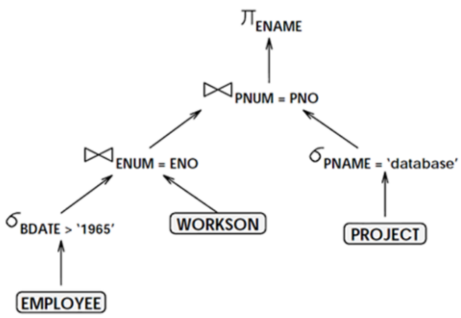
EMPLOYEE(ENUM, ENAME, BDATE)
PROJECT(PNUM, PNAME)
WORKSON(ENO, PNO)

Replace "$\sigma - \times$" by "$\bowtie$"



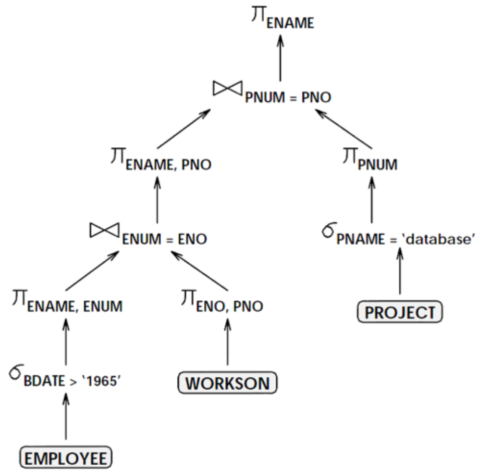
After Optimization





Move PROJECTs down

After optimization



Next consider different join costs

(Employee join Workson) join Project
Employee join (Workson join Project)

Question: Can you recall the cost of join?

Join Optimization

- Consider just three relations R, S, and T

```
Form 1:
   ( R ⋈ S ) ⋈ T       ( S ⋈ R ) ⋈ T
   ( R ⋈ T ) ⋈ S       ( T ⋈ R ) ⋈ S
   ( S ⋈ T ) ⋈ R       ( T ⋈ S ) ⋈ R

Form 2:
   R ⋈ ( S ⋈ T )       R ⋈ ( T ⋈ S )
   S ⋈ ( R ⋈ T )       S ⋈ ( T ⋈ R )
   T ⋈ ( R ⋈ S )       T ⋈ ( S ⋈ R )
```

```
Form 1:  ( • ⋈ • ) ⋈ •

Form 2:  • ⋈ ( • ⋈ • )
```

The **number** of ***different*** **join orderings** of ***n*** **relations** is **exponentially large** !!!

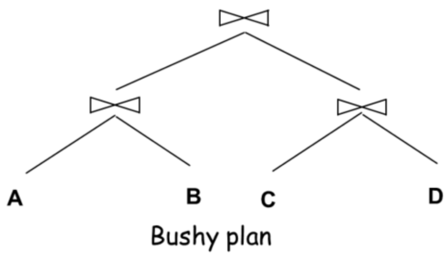
(**Because** the **number** of **permutations** is **exponentially large**)

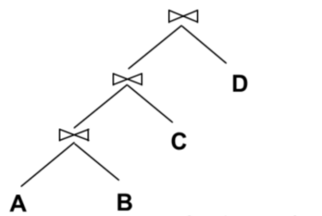
- The **number** of ***possible*** **join trees** to **consider** is just ***too*** **large**

- We **need** to **reduce** the **search space**....

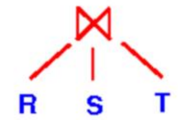
Plans to consider

- 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.

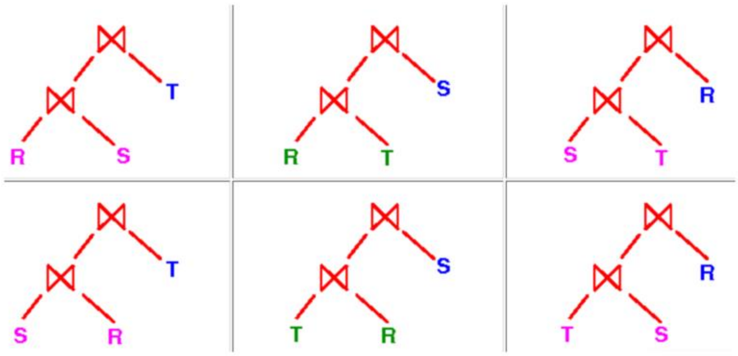




Three-way Join



- Possible left-deep join trees:



Enumeration of Left-Deep Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation *(All 2-relation plans)*
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation *(All N-relation plans)*
- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples
- In spite of pruning plan space, this approach is still exponential in the # of tables. (works well for most queries with less than 15 tables)
- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator.

Pass 1 question: what access methods should be used for this selection? (B+ tree? Hash index? Sequential scan?)

Pass2 question: which two relations should join? How to join?

Summary

- Two parts to optimize a query
  - Consider a set of alternative plans, 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
- Single-relation queries
  - All access paths considered, cheapest is chosen
  - Issues: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order
- Multiple-relation queries
  - All single-relation plans are first enumerated
    - Selections/projections considered as early as possible
  - For each 1-relation plan, all ways of joining another relation are considered
  - For each 2-relation plan that is `retained', all ways of joining another relation are considered, etc.