Overview of Query Evaluation
Overview of Query Optimization

• **Goal:** Choose an **efficient execution plan** for a query (hence avoiding the really inefficient plans)

• Given a SQL query, define and analyze cost of many different execution plans that produce the correct answer for the query, but with potentially very different run times

• First, we will focus on how to evaluate individual relational algebra operators efficiently

• Afterwards, we will discuss how to evaluate full-fledged SQL queries efficiently

Overview of Query Optimization

• We will express each alternative query execution plan as a **tree of relational algebra operations**, annotated with choice of algorithm for each operator

• We will discuss two main issues:
  • What family of execution plans should we consider for a given query?
  • How do we estimate the cost (or running time) of each execution plan in this search space of possible plans, so that we can choose a good plan?

• We will not attempt to find the best possible plan, but rather to find a good plan and avoid the worst plans
Database Statistics and Catalogs

• To analyze execution plans, we need information about the relations and indexes involved
• Such information is kept in the database catalogs, which at least contain:
  • For each relation, # tuples and # pages
  • For each index, # distinct values for the search key attributes of the index and # pages occupied by index
  • For each B+ tree index, index height, lowest and highest values for the search key attributes of index
• Catalogs are updated periodically
  • Updating whenever data changes would be too expensive; lots of approximation anyway, so slight inconsistencies are fine
  • More detailed information (e.g., histograms of the values in some attribute) is often available as well

Access Paths: Methods for Retrieving Tuples of a Relation

• Alternative access paths: scan relation file (always available); or use indexes that match a selection condition
• Consider an attribute A as search key
  • Case 1: A B+ tree index on A matches selection condition A op k if op is =, <, >, ≤, ≥
  • Case 2: A hash index on A matches only equality selection condition A = k
Processing a Selection Condition

- Consider **all access paths** that could be used for the relation and selection condition: scan plus any indexes that match condition
- Find the **most selective** access path (i.e., requiring **fewest I/Os**), retrieve tuples using it, and apply any remaining conditions that don’t match the index

---

**Processing a Selection Condition: Example**

*Employees*(ssn, name, sal, did), with

*B* tree on name, hash table on did, *B* tree on sal

\( \sigma_{\text{name} = \text{`Jane`} \ AND \ \text{did=214 AND sal}>50\text{K}} \) *(Employees)*

Alternative access paths and corresponding execution plans:

1. **Scan** relation and check full selection condition for each retrieved tuple
2. Use **B* tree index on name** to find the set *N* of rids of all tuples with name=`Jane`; retrieve those tuples and filter with other conditions (i.e., did=214 and sal>50K)
3. Use **hash table on did** to find the set *D* of rids of all tuples with did=214; retrieve those tuples and filter with other conditions (i.e., name=`Jane` and sal>50K)
4. Use **B* tree index on sal** to find the set *S* of rids of all tuples with sal>50K; retrieve those tuples and filter with other conditions (i.e., name=`Jane` and did=214)
Processing a Selection Condition: Example

Employees(ssn, name, sal, did), with
B⁺ tree on name, hash table on did, B⁺ tree on sal

\[ \sigma_{\text{name}='Jane' \text{ AND } \text{did}=214 \text{ AND } \text{sal}>50K}(\text{Employees}) \]

Yet another possible plan uses all three indexes:

5. Retrieve three sets of rids using the three indexes that match the selection condition:
   - Use **B⁺ tree index on name** to find the set \( N \) of rids of all tuples with name='Jane'
   - Use **hash table on did** to find the set \( D \) of rids of all tuples with did=214
   - Use **B⁺ tree index on sal** to find the set \( S \) of rids of all tuples with sal>50K
   - Retrieve and return only the tuples in \( N \cap D \cap S \) (i.e., the tuples whose rid is in the intersection of the three sets of rids and hence satisfy all three conditions)

Sometimes “index-only” query execution plans are possible

Factors Influencing Choice of Plan

- # of tuples matching index condition
- index characteristics

For our example, we know (from database catalogs) that Employees has:

- 100,000 tuples in 1,000 blocks (i.e., 100 tuples/block)
- 1,000 distinct name values
- 500 distinct did values
- values of sal between 20K and 250K

Cost of plan (1), Scan? 1,000 I/Os
Cost of Plan (2), B$^+$ Tree on Name

- **2 or 3 I/Os** to navigate from root of B$^+$ tree to leaf node for name='Jane'
- Cost of retrieving all leaf nodes with data entries with name='Jane':
  - on average, 100,000/1,000=100 such data entries, based on the number of employees and the number of distinct name values
  - we can assume that all 100 data entries fit in one leaf, hence no further I/Os to get them all
- Cost of retrieving the actual tuples associated with the matching rids, to filter with other conditions
  - in worst case, **100 extra I/Os** (each tuple might be in a different block in worst case)

Cost of plan (2), B$^+$ tree on name: **102 or 103 I/Os**

Cost of Plan (3), Hash Table on Did

- **1 I/O** to get hash table bucket/block with all data entries for did=214
- Cost of retrieving the actual tuples associated with the matching rids, to filter with other conditions
  - on average, 100,000/500=200 such data entries, based on the number of employees and the number of distinct did values
  - in worst case, **200 extra I/Os** (each tuple might be in a different block in worst case)

Cost of plan (3), hash table on did: **201 I/Os**

Note this is worse than plan (2), because the condition on did is **less selective** than that on name
Cost of Plan (4), B⁺ Tree on Sal

- **2 or 3 I/Os** to navigate from root of B⁺ tree to leaf node for sal=50K
- Cost of retrieving all leaf nodes with data entries with sal>50K:
  - “fraction” of sal space covered by sal>50K condition: 
    \( \frac{250K-50K}{250K-20K} = 0.87 \), based on range of values of sal
  - on average, we then need \( 100,000 \times 0.87 = 87,000 \) such data entries
- Cost of retrieving the actual tuples associated with the matching rids, to filter with other conditions
  - most likely retrieving the actual tuples for the 87,000 data entries will require retrieving all 1,000 blocks of the relation
  - so we are better off just retrieving all blocks of the relation without using the B⁺ tree index on sal

**Plan (4), B⁺ tree on sal: dominated by other plans**

The condition on sal is not selective enough to merit the use of the B⁺ tree on sal

Cost of Plan (5), Using 3 Indexes

- Cost of using each of the three indexes:
  - 2 or 3 I/Os for B⁺ tree on name
  - 1 I/O for hash table on did
  - 2 or 3 I/Os plus 87% of all leaf nodes for B⁺ tree on sal
- B⁺ tree on sal not competitive, so use just B⁺ tree on name to get set of rids \( N \) and hash table on did to get set of rids \( D \), at a cost of **3 or 4 I/Os**
- Cost of retrieving the actual tuples with rids in \( N \cap S \):
  - crude analysis: assuming that condition on name is independent of condition on did, then we expect \( 100,000 \times \frac{1}{1000} \times \frac{1}{500} < 1 \) rids in intersection, so at most 1 more I/O expected

Cost of **plan (5)**, using 2 (of 3) indexes: **4 or 5 I/Os**

Best plan, but analysis depended on crude independence assumption
Processing a Selection Condition With ORs

\[
\text{Employees}(\text{ssn, name, sal, did}), \text{ with } \\
B^+ \text{ tree on name, hash table on did, } B^+ \text{ tree on sal} \\
\sigma_{\text{name}='\text{Jane}' \text{ OR did}=214 \text{ OR sal}>50K}(\text{Employees})
\]

Alternative access paths and corresponding execution plans:

1. **Scan** relation and check full selection condition for each retrieved tuple
2. Retrieve three sets of rids using the three indexes that match the selection condition:
   - **Use** \(B^+\) **tree index on name** to find the set \(N\) of rids of all tuples with name='Jane'
   - **Use** hash table on did to find the set \(D\) of rids of all tuples with did=214
   - **Use** \(B^+\) **tree index on sal** to find the set \(S\) of rids of all tuples with sal>50K
   - Retrieve and return only the tuples in \(N \cup D \cup S\) (i.e., the tuples whose rid is in the union of the three sets of rids)

**Scan** is likely to prevail given the low selectivity of the condition on sal

---

Processing a Projection

\[
\text{SELECT DISTINCT R.sid, R.bid} \\
\text{FROM Reserves R}
\]

- **Without DISTINCT**: No duplicate elimination, so just use **scan**
- **With DISTINCT**: Need duplicate elimination; two main options:
  - **Sorting**: Sort on \(<\text{sid}, \text{bid}>\) and remove duplicates; drop unneeded attributes while sorting, for efficiency
  - **Hashing**: Hash on \(<\text{sid}, \text{bid}>\) to create buckets; then load buckets into memory one at a time, build in-memory hash table for bucket (with a different hash function), and eliminate duplicates
Processing Joins

• Joins are a highly optimized operation in DBMSs
• Several well studied algorithms available:
  • “Nested loops” join family of algorithms
  • “Sort-merge” join
  • “Hash” join
  • …
• Best choice for a query depends on characteristics of the query and the relations, as well as on available indexes

Processing Joins: Example

Reserves: \( p_R = 100 \) tuples/page; 100K tuples; \( M=1,000 \) pages
Sailors: \( p_S = 80 \) tuples/page; 40K tuples; \( N=500 \) pages

\( \text{SELECT * FROM Reserves R, Sailors S WHERE R.sid=S.sid; } \)

• We will describe different join execution algorithms and estimate their cost in number of I/Os
• Our analysis will ignore the cost of writing the output, which is equal across execution algorithms

Options:
• Naïve nested loops join
• Page-oriented nested loops join
• Block nested loops join
• (Index nested loops join)
• (Sort-merge join)
• (Hash join)
• …
Naïve Nested Loops Join

For each tuple r in R do:
   For each tuple s in S do:
      if r.sid = s.sid then add r \Join s to result

• R is the “outer” relation; S is the “inner” relation

• cost = M + pR*M*N I/Os = 1,000 + 100K*500 I/Os ≈ 5*10^7 I/Os
• @ 10 ms/I/O: about 140 hours (!)

Page-Oriented Nested Loops Join

Improvement: Read inner relation only as many times as there are pages of the outer relation, not once per outer tuple

For each page B_R of R do:
   For each page B_S of S do:
      For each tuple r in B_R and tuple s in B_S:
         if r.sid = s.sid then add r \Join s to result

• R is the “outer” relation; S is the “inner” relation

• cost = M + M*N I/Os = 1,000 + 1,000*500 I/Os = 501K I/Os
• @ 10 ms/I/O: about 1.4 hours
Block Nested Loops Join

Improvement: Exploit main-memory buffer pool as much as possible to reduce number of I/Os

Assumption: B pages of main memory available for processing join

For each “chunk” $C_R$ of B-2 pages of R do:
  For each page $B_S$ of S do:
    For each tuple $r$ in $C_R$ and tuple $s$ in $B_S$:
      if $r.sid = s.sid$ then add $r \bowtie s$ to result

- cost, for $B=102 = M + \text{ceiling}(M/(B-2)) * N$ I/Os = 1,000 + ceiling(1,000/100)*500 I/Os = 6K I/Os