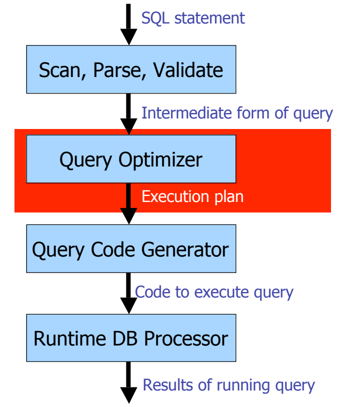
CS 2550 / Spring 2006
Principles of Database Systems

08 – Query processing and optimization

Alexandros Labrinidis
University of Pittsburgh

Steps in processing a query



Query Processing: Selections

- 1) Convert to relational algebra
  select last, first
  from employee
  where salary>25000;

  ➔ $\Pi_{\text{last, first}} (\sigma_{\text{salary>25000}} (\text{employee}))$

- 2) Choose an implementation
  - Factors?
    - Index Type
    - Query Type
    - Statistics

Implementations for Selection

- A1: linear search
  - Full scan
- A2: binary search
  - Assume file is ordered on attribute
- A3: using primary index (or hash key)
  - Equality on key attribute
- A4: using primary/clustering index – multiple records
  - Equality on non-key attribute
- A5: using secondary index
  - Most general method – key/non-key attribute

Implementations for Selection - II

- A6: primary index, comparison
- A7: secondary index, comparison

Implementations for Selection III

- How to handle conjunction (AND) / disjunction (OR)?
- A8: Conjunctive selection using individual index
  - Check simple condition first, if it has index
- A9: Conjunctive selection using composite index
  - Composite on both attributes must exist
- A10: Conjunctive selection by intersection of record ptrs
  - Evaluate simple conditions independently
  - Produce intersection of lists of RIDs
- A11: Disjunctive selection by union of record ptrs

Example Selection Queries

- $\sigma_{\text{salary>25000}}$ **(emp)**
- $\sigma_{\text{ssn=123456789}}$ **(emp)**
- $\sigma_{\text{dept\_number>5}}$ **(dept)**
- $\sigma_{\text{dnum=6}}$ **(emp)**
- $\sigma_{\text{sex='m'}}$ **(emp)**
- $\sigma_{\text{dnum=6 AND salary>25000 AND sex='f'}}$ **(emp)**
- $\sigma_{\text{salary>25000 AND salary<35000}}$ **(emp)**

How to choose?

- Index Type
  - What indexes are available for the given relations?
- Query Type
  - Do we have range query, point query, conjunction?
- Statistics
  - Selectivity
  - Examples:
    - $\sigma_{\text{sex='m'}}$ **(emp)**
    - $\sigma_{\text{ssn='123456789'}}$ **(emp)**

Query Processing: Joins

- J1: Nested-loop join
- J2: Single-loop join
- J3: Sort-merge join
- J4: Hash-join

J1: Nested-loop join

- Join relations R and S
  - A is the common attribute in R, B is the common attribute in S
- For each record t in R (=outer loop)
  - For each record s in S (=inner loop)
    - Test if t[A] = s[B]
- In practice, we are accessing an entire disk block at a time rather than a record at a time.
- Is there any difference which relation will be inner/outer?

J2: Single-loop join

- Join relations R and S
  - A is the common attribute in R, B is the common attribute in S
- Must use an access structure to retrieve the matching records
- Only works if an index (hash) key exists for one of the two join attributes (A or B), say B
- For each record t in R
  - Locate tuples s from S, that satisfy s[B] = t[A]

J3: Sort-merge join

- Join relations R and S
  - A is the common attribute in R, B is the common attribute in S
- **IF** relations R and S are physically sorted (ordered) by the value of the join attributes
  - we simply have to scan the relations
  - produce match or advance pointer
- Q: what happens if the relations are not sorted?

J4: Hash-join

- Join relations R and S
  - A is the common attribute in R, B is the common attribute in S
- 1) Single pass through relation with fewer records (R)
  - Partitioning phase (into hash buckets)
- 2) Single pass through other relation (S)
  - Probing phase (use hash to find matching records of R)
- Q: Will this work if R does not fit in memory?

Query Processing: Joins

- J1: Nested-loop join
- J2: Single-loop join
- J3: Sort-merge join
- J4: Hash-join
  - **Partition Hash Join**
  - **Hybrid Hash Join**

Partition Hash Join

- Join relations R and S
- Partitioning Phase
  - Partition hash function
  - R into M partitions: $R_1, R_2, \ldots, R_M$
  - S into M partitions: $S_1, S_2, \ldots, S_M$
    - IDEA: $\mathbf{R_i}$ only needs to be joined with $\mathbf{S_i}$
- Probing Phase
  - Perform M iterations
    - Join partitions R_i and S_i
    - Can use nested-loop join or hash-join
      - If hash-join, must use different hash function. WHY?

Partition Hash Join – Discussion

- Q: What is the cost?
- A: How many times each block is read/written?
  - Partitioning Phase: R: read once, write once; S: read once, write once
  - Probing Phase: R: read once; S: read once; write results once
  - Total cost = $3*(b_R + b_S) + b_{results}$
- Q: What is the main difficulty of the algorithm?
- A: What is the partitioning phase relying on?
  - Hash function is uniform!
  - i.e. partition sizes are nearly equal in size

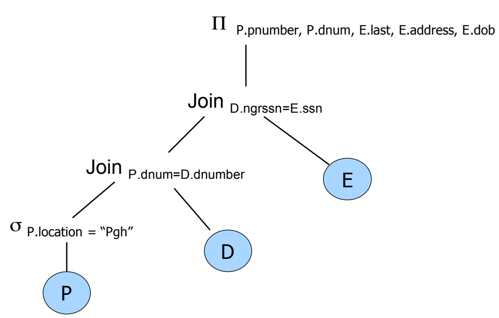
Hybrid Hash Join

- Variation of Partition Hash Join
- Main idea:
  - Get a "free-ride" for joining the first partition during first pass
- Differences:
  - Partition Hash Join:
    - M partitions, single-block in memory for each one
  - Hybrid Hash Join
    - M partitions, store first one fully, M-1 with single-block
  - Partitioning Phase completely joins first partition
  - Probing Phase applied to M-1 partitions

Processing of Complex Queries

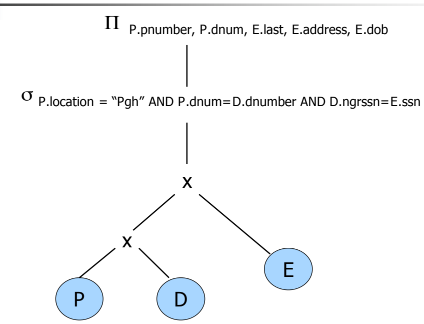
- Query is translated into a **sequence** of relational operators
- Q: Is single-operator-at-a-time appropriate?
  - A: **NO**. Why?
  - A: We would need temporary relations (and extra disk space) to store intermediate results
- What is the alternative?
  - Combine operators (and their execution) into a sequence
  - **Pipelining or Stream-based Processing**

Example of Query Tree



- Π $_{\text{P.pnumber, P.dnum, E.last, E.address, E.dob}}$ $(((\sigma_{\text{P.location = "Pgh"}} (P))$ join $_{\text{dnum=dnumber}}$ (D) join $_{\text{mgrssn=ssn}}(E))$

Example of Query Tree – 2nd version



-
```
Select P.PNUMBER, P.DNUM, E.LAST, E.ADDRESS, E.DOB
From   Project as P, Department as D, Employee as E
Where  P.DNUM = D.DNUMBER and D.MGRSSN = E.SSN and
       P.LOCATION='PGH'
```

Heuristic Optimization of Query Trees

- Get initial query tree
  - Apply CARTESIAN PRODUCT of relations in FROM
  - Selection and Join conditions of WHERE is applied
- This is very inefficient. Why?
  - Cartesian product causes "explosion" in number of tuples
- How to avoid this?
  - Identify Joins
  - Push selections down the tree (reduce number of tuples)
  - Push project operations down the tree (reduce # of attributes)

General Transformation Rules - 1

- Cascade of σ
  - $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } Cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$
- Commutativity of σ
  - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
- Cascade of Π
  - $\Pi_{list1}(\Pi_{list2}(\ldots(\Pi_{listN}(R))\ldots)) = \Pi_{list1}(R)$
- Commuting σ with Π
  - $\Pi_{A1, A2, \ldots, An}(\sigma_c(R)) = \sigma_c(\Pi_{A1, A2, \ldots, An}(R))$

General Transformation Rules - 2

- Commutativity of join and cartesian product
  - R join S = S join R
  - R x S = S x R
    - Note: order of attributes will not be the same
- Commuting σ with join (or cartesian product)
  - σ_c (R join S) = (σ_c (R)) join S
    - If $c = c_1$ AND c_2, with c_1 referring to R, c_2 referring to S:
    - σ_c (R join S) = (σ_{c1} (R)) join (σ_{c2} (S))
  - Same rules apply for cartesian product

General Transformation Rules - 3

- Commuting Π with join or cartesian product
  - Project list $L = \{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$
  - Suppose $A_1, A_2, \ldots, A_n$ are attributes of relation R, and that $B_1, B_2, \ldots, B_m$ are attributes of relation S
  - Π_L (R join$_c$ S) = ($\Pi_{A1, A2, \ldots, An}$ (R)) join$_c$ ($\Pi_{B1, B2, \ldots, Bm}$ (S))
    - Note: c must only involve attributes in L
    - What if c contains additional attributes?
- Commutativity of set operations:
  - Union?
  - Intersection?
  - Set difference?

General Transformation Rules - 4

- Associativity of join, x, ∪, ∩
  - (R * S) * T = R * (S * T)
    - where * can be any of join, x, ∪, ∩
- Commuting σ with set operations
  - σ_c (R # S) = (σ_c (R)) # (σ_c (S))
    - where # can be any of −, ∪, ∩
- Commuting of P operation:
  - Π_L (R ∪ S) = (Π_L (R)) ∪ (Π_L (S))

General Transformation Rules - 5

- Converting a (σ, x) sequence into a join operation
  - If c corresponds to a join condition, then
    - σ_c (R x S) = R $join_c$ S
- Other transformations:
  - Any boolean transformation can still be applied, e.g.:
    - not (C_1 and C_2) = (not C_1) or (not C_2)
    - not (C_1 or C_2) = (not C_1) and (not C_2)

Outline of algebraic optimization

- Break up selections (with conjunctive conditions) into a cascade of selection operators
- Push selection operators as far down in the tree as possible
- Rearrange leaf nodes to:
  - Execute first the most restrictive select operators
    - What is restrictive? (Fewest tuples or Smallest size)
  - Make sure we don't have cartesian products
- Convert cartesian products into joins
- Move projections as far down as possible
- Identify subtrees that represent groups of operations which can be executed by single algorithm

Cost-based query optimization

- Compiled queries VS interpreted queries
- Cost-based query optimization
  - Full-scale optimization, taking costs & selectivities into account
  - Usually happens only for compiled queries
- Costs:
  - Access cost to secondary storage
  - Storage cost (for intermediate results)
  - Computation cost
  - Memory usage cost
  - Communication cost (data/query shippping)

How to estimate costs?

- How can we determine costs without running the query?
  - Cost model
  - Sizes (record, block, …)
  - Selectivities
  - Number of distinct values

- Histograms