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

1. $\sigma_{\text{salary} > 25000} (\text{emp})$
2. $\sigma_{\text{ssn}=123456789} (\text{emp})$
3. $\sigma_{\text{dept}._{\text{number}}>5} (\text{dept})$
4. $\sigma_{\text{dnum}=6} (\text{emp})$
5. $\sigma_{\text{sex}='m'} (\text{emp})$
6. $\sigma_{\text{dnum}=6 \text{ AND salary}>25000 \text{ AND sex}='f' } (\text{emp})$
7. $\sigma_{\text{salary}>25000 \text{ AND salary}<35000} (\text{emp})$

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:
      - $\sigma_{\text{sex}='m'} (\text{emp})$
      - $\sigma_{\text{ssn}=123456789} (\text{emp})$
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\(_2\), ..., R\(_M\)
- S into M partitions: S\(_1\), S\(_2\), ..., S\(_M\)
  - IDEA: R\(_i\) only needs to be joined with 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
    - Probing Phase: R: read once
    - S: read 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

\[
\Pi_{p\text{.pnumber}, p\text{.dnum}, e\text{.last}, e\text{.address}, e\text{.dob}} \\
\sigma_{p\text{.location} = 'Pgh'} \\
\Pi_{p\text{.pnumber}, p\text{.dnum}, e\text{.last}, e\text{.address}, e\text{.dob}} \\
(((\sigma_{p\text{.location} = 'Pgh'} P) \bowtie_{d\text{.dnum}=p\text{.dnum}} D) \bowtie_{d\text{.mgrssn}=e\text{.ssn}} E)
\]

Example of Query Tree – 2nd version

\[
\Pi_{p\text{.pnumber}, p\text{.dnum}, e\text{.last}, e\text{.address}, e\text{.dob}} \\
\sigma_{P\text{.location} = 'Pgh' \text{ AND } P\text{.dnum}=D\text{.dnum} \text{ AND } D\text{.mgrssn}=E\text{.ssn}} \\
\Pi_{p\text{.pnumber}, p\text{.dnum}, e\text{.last}, e\text{.address}, e\text{.dob}} \\
\text{Select } P\text{.PNUMBER, P\text{.DNUM, E.LAST, E.ADDRESS, E.DOB}} \\
\text{From Project as P, Department as D, Employee as E} \\
\text{Where } P\text{.DNUM = D\text{.DNUMBER} \text{ and D\text{.MGRSSN} = E\text{.SSN} \text{ and P\text{.LOCATION} = 'Pgh'}}}
\]
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 \( \sigma \)
  - \( \sigma_{c_1 \land c_2 \land \ldots \land c_n}(R) = \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R)) \ldots)) \)

- Commutativity of \( \sigma \)
  - \( \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R)) \)

- Cascade of \( \Pi \)
  - \( \Pi_{A_1 A_2 \ldots A_m}(\Pi_{B_1 B_2 \ldots B_n}(R)) = \Pi_{B_1 B_2 \ldots B_n}(\Pi_{A_1 A_2 \ldots A_m}(R)) \)

- Commuting \( \sigma \) with \( \Pi \)
  - \( \Pi_{A_1 A_2 \ldots A_m}(\sigma_c(R)) = \sigma_c(\Pi_{A_1 A_2 \ldots A_m}(R)) \)

General Transformation Rules - 2

- Commutativity of join and cartesian product
  - \( R \cup S = S \cup R \)
  - \( R \times S = S \times R \)
    - Note: order of attributes will not be the same

- Commuting \( \sigma \) with join (or cartesian product)
  - \( \sigma_c(R \cup S) = (\sigma_c(R)) \cup S \)
    - If \( c = c_1 \land c_2 \), with \( c_1 \) referring to \( R \), \( c_2 \) referring to \( S \):
      - \( \sigma_{c_1}(R \cup S) = (\sigma_{c_1}(R)) \cup (\sigma_{c_2}(S)) \)
    - Same rules apply for cartesian product

General Transformation Rules - 3

- Commuting \( \Pi \) 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 \( S \)
  - \( \Pi_c(R \cup S) = (\Pi_{A_1 A_2 \ldots A_n}(R)) \cup (\Pi_{B_1 B_2 \ldots B_m}(S)) \)
    - 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, $x$, $U$, $\cap$
  - $(R * S) * T = R * (S * T)$
    where * can be any of join, $x$, $U$, $\cap$

- Commuting $\sigma$ with set operations
  - $\sigma_c (R \# S) = (\sigma_c (R)) \# (\sigma_c (S))$
    where # can be any of $\sim$, $U$, $\cap$

- Commuting of $P$ operation:
  - $\Pi_L (R \cup S) = (\Pi_L (R)) \cup (\Pi_L (S))$

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

General Transformation Rules - 5

- Converting a $(\sigma, x)$ sequence into a join operation
  - If $c$ corresponds to a join condition, then
    - $\sigma_c (R \times S) = R \text{ join}_c S$

- Other transformations:
  - Any boolean transformation can still be applied, e.g.:
    - not $(C_1 \text{ and } C_2) = (\text{not } C_1) \text{ or } (\text{not } C_2)$
    - not $(C_1 \text{ or } C_2) = (\text{not } C_1) \text{ and } (\text{not } C_2)$

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 shipping)
How to estimate costs?

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

- Histograms