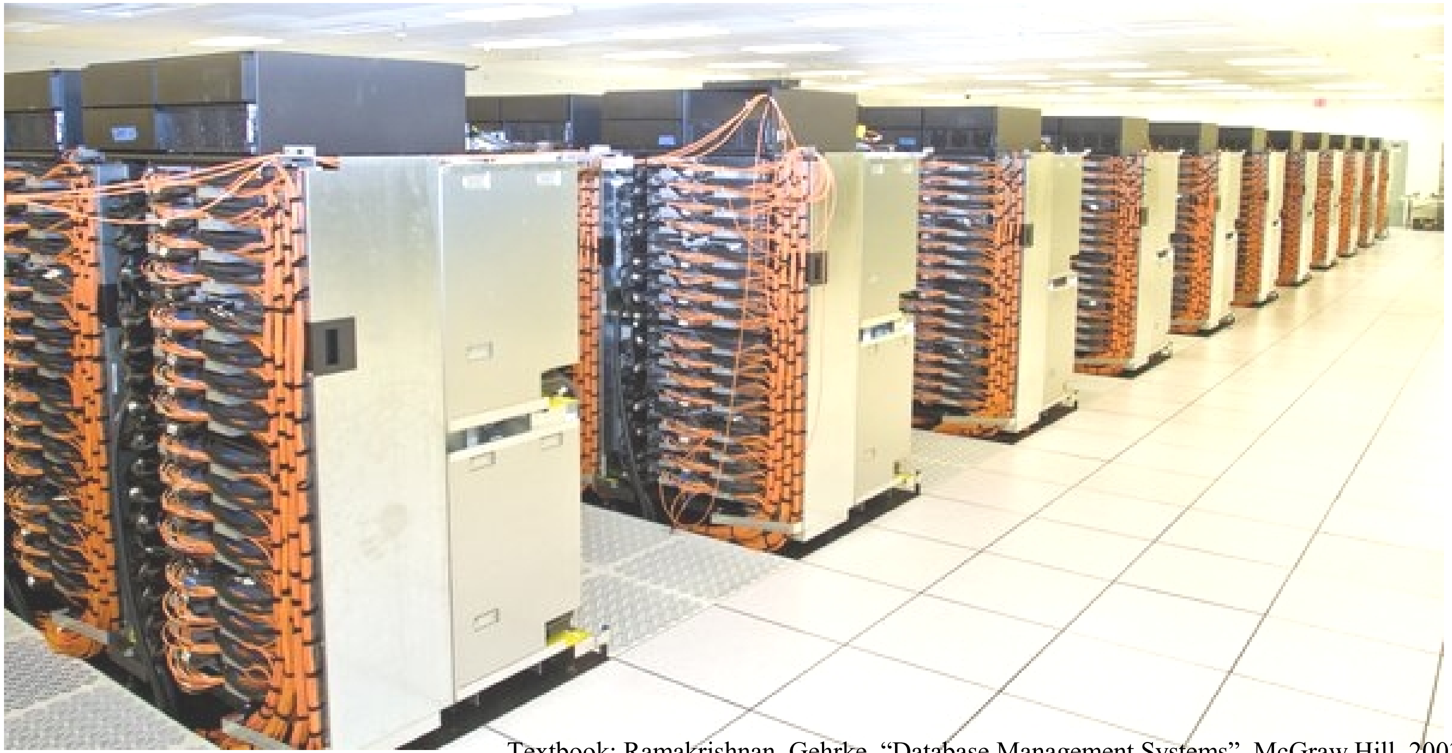
Databases

Cap. 9. SQL Execution Plan. Introduction to Query Optimization



Textbook: Ramakrishnan, Gehrke, “Database Management Systems”, McGraw Hill, 2003

2017 UPT

Conf.Dr. Dan Pescaru

Relational operators

1. Useful for SQL query decomposition
   - Used for representing a query execution plan (close to expression tree in compiler lexical analysis)
2. A query is applied to relation instances
3. The result of a query is also a relation instance
4. The schema for the result of a given query is fixed

Basic operators

1. Selection (σ)
   - Selects a subset of rows from a relation
2. Projection (π)
   - Removes unwanted columns from a relation
3. Cross-product (x)
   - Combine two relations
4. Set-difference (–)
   - Tuples in Relation 1, but not in Relation 2
5. Union ($\cup$)
   - Tuples in Relation 1 and in Relation 2

Additional operators

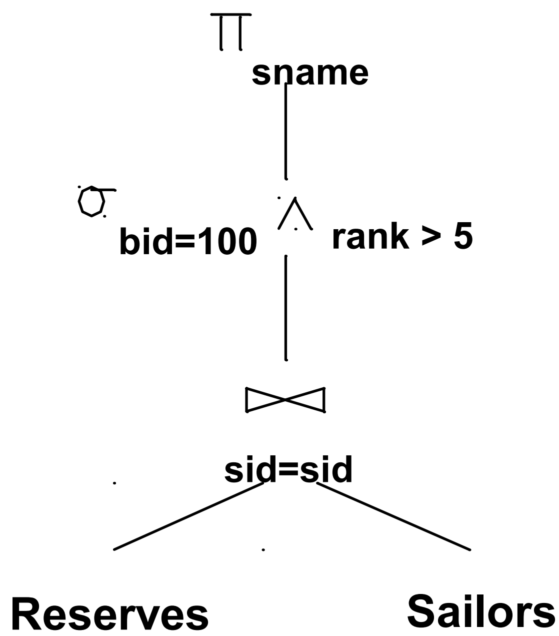
1. Join ($\circ$)
   - Corresponding tuples based on a relationship between certain columns in relations
2. Intersection ($\cap$)
   - Tuples that exists in both relations
3. Renaming
   - Change an attribute name
4. Since each operation returns a relation, operations can be composed. Relational algebra is "closed"

Query evaluation

1. Plan: Tree of relational algebra operators, with choice of algorithms for each of them
2. Two main issues in query optimization:
   a. For a query, what plans are considered?
   b. Algorithm to search plan space for cheapest (estimated) plan
3. How is the cost of a plan estimated?
4. Ideally: Want to find best plan. Practically: Avoid worst plans!

Execution plan example

- E.g. SELECT sname FROM Sailors s
  INNER JOIN Reserves r ON s.sid=r.sid
  WHERE r.bid=100 AND s.rank>5



Basic optimization techniques

1. Algorithms for evaluating relational operators use some simple ideas
2. Iteration: sometimes, faster to scan all tuples even if there is an index. Sometimes, faster to scan the data entries in an index instead of the table
3. Indexing: can use WHERE conditions to retrieve small set of tuples (σ,$\circ$)
4. Partitioning: by using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs

Statistics and catalogs

1. Catalogs contain metadata and statistics:
   - # tuples (NTuples) and # pages (NPages) for each relation
   - # distinct key values (NKeys) and NPages for each index
   - Index height, Low/High key values for each tree index
2. Catalogs updated periodically (efficiency)
3. Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok

Cost estimation

1. For each execution plan considered, must estimate cost
   - Cost of each operation in plan tree
     - Depends on input cardinalities
     - Depends on tuple size
2. Must also estimate size of result for each operation in tree!
   - Use information about the input relations
   - For selections and joins, assume independence of predicates

Access paths

1. An access path is a method of retrieving tuples
   - File scan, or index that matches a selection (in the query)
2. A tree index matches a conjunction of terms that involve only attributes in a prefix of the search key
   - E.g., Tree index on <a, b, c> matches the selection a=5 AND b=3, and a=5 AND b>6, but not b=3
3. A hash index matches a conjunction of terms that has a term attribute = value for every attribute in the search key of the index
   - E.g., Hash index on <a, b, c> matches a=5 AND b=3 AND c=5; but it does not match b=3 or a>5 AND b=3 AND c=5

Optimizing selection

1. Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
   1. Most selective access path: an index or file scan that we estimate will require the fewest page I/Os
   2. Terms that match this index reduce the number of tuples retrieved; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched

Optimizing selection example

1. Consider: day<8/9/94 AND bid=5 AND sid=3
2. A B+ tree index on day can be used
   - bid=5 and sid=3 must be checked for each retrieved tuple
3. Similarly, a hash index on <bid, sid> could be used
   - day<8/9/94 must then be checked

Optimizing projection

1. The expensive part is removing duplicates
   - SQL systems do not remove duplicates unless the keyword DISTINCT is specified in a query
2. *Sorting* Approach: sort on keys and remove duplicates. Can optimize this by dropping unwanted information while sorting
3. *Hashing* Approach: Hash on keys to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates

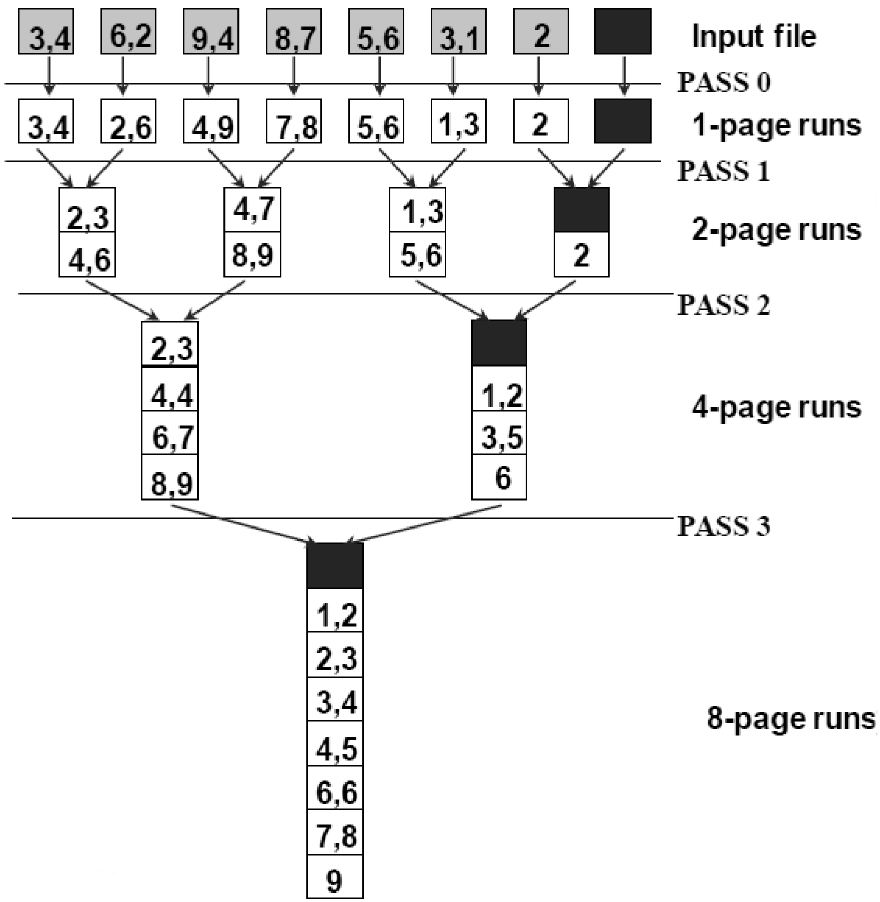
Projection: sorting approach

SELECT DISTINCT sid, bid FROM Reserves

1. An approach based on sorting:
   - Two way external merge sort with elimination of unwanted fields
   - Tuples in result are smaller than input tuples (size ratio depends on # and size of fields that are dropped)
   - Modify merging passes to eliminate duplicates. Thus, number of result tuples is smaller than input (difference depends on # of duplicates)

Two way external merge sort

1. Idea: sort partition files and merge (using divide and conquer), three buffers needed
2. In each pass: read + write each page
3. Total cost is approx. $2N \times \log_2 N$



Projection: hashing approach

SELECT DISTINCT sid, bid FROM Reserves

1. Partitioning phase using B buffers: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function h1 to choose one of B-1 output buffers
   - Result is B-1 partitions (of tuples with no unwanted fields). Two tuples from different partitions guaranteed to be distinct
2. Duplicate elimination phase: For each partition, read it and build an in-memory hash table, using hash function h2 (<> h1) on all fields, while discarding duplicates by checking the equal hash-value

Join: Index Nested Loops

SELECT r.*, s.*

FROM Reserves r, Sailors s WHERE r.sid=s.sid

1. Without an index: approx. $Rows_{R1}$ x $Pages_{R2}$ I/Os (for 50000x1000 = 70h at 5ms/(I/O) on average 7200rpm HDD)!
2. If there is an index on the join column of one relation, can make it the inner and exploit the index
   - Cost: $Rows_{R1}$ x cost of finding matching R2 tuples
3. For each R1 tuple, cost of probing R2 index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding R2 tuples depends on clustering
4. Clustered index: 0 I/O
5. Unclustered: 1 I/O per matching S tuple

Set operation optimization

1. Intersect and Cross-Product special cases of join
2. Sorting based approach to Union:
    - Sort both relations (on combination of all attributes)
    - Scan sorted relations and merge them
3. Hash based approach to Union:
    - Partition R1 and R2 using hash function *h*
    - For each R2-partition, build in-memory hash table (using *h2*), scan correlated R1-partition and add tuples to table while discarding duplicates

Aggregate operators optimization (I)

1. Without grouping:
   - In general, requires scanning the relation
   - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan

Aggregate operators optimization

1. With grouping:
   - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation)
   - Similar approach based on hashing on group-by attributes
   - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order and apply HAVING same time

Query optimization example