# Query Evaluation Techniques for Large Databases

Presented by: Sampoorna Biswas

Adapted from slides by: Daniela Stasa

# Purpose

To survey practical query evaluation techniques for executing “complex queries” over “large databases”

- Complex query: Combination of query processing algorithms
- Large DB: MBs to TBs

# Query Execution

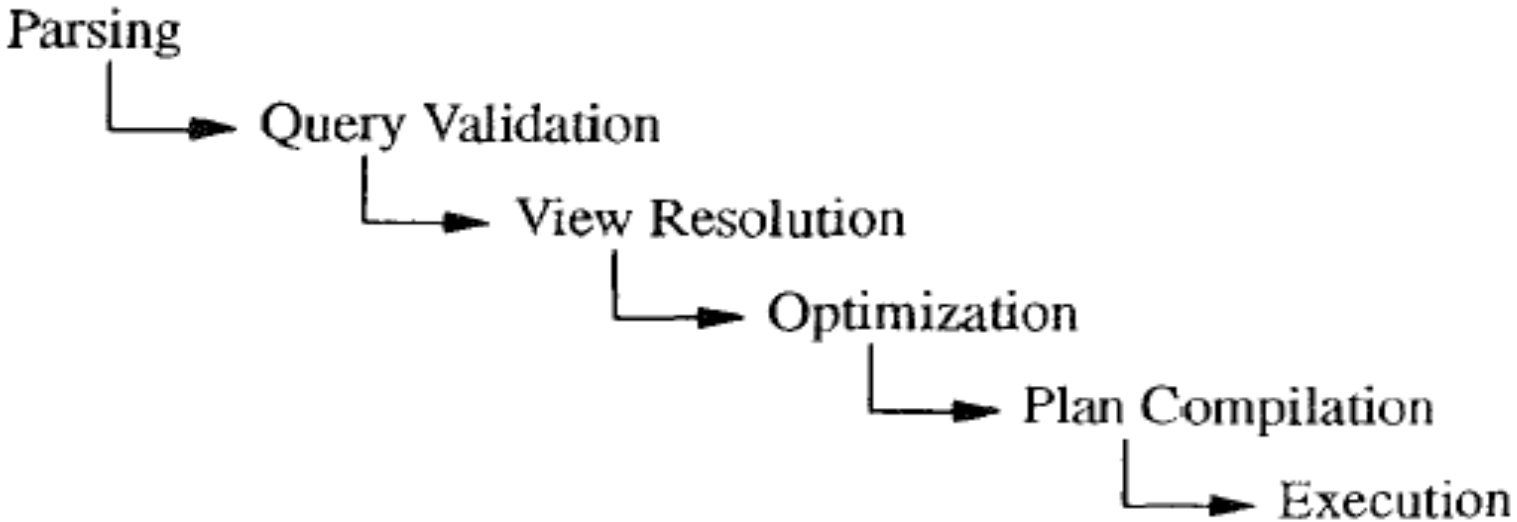



**Figure 2.** Query processing steps.

# Query Execution

- Parse into an internal form
- Validate query to ensure referenced objects exist
- Expand macros, views
- Map query to an optimized plan
- Convert execution plan to machine code
- Compile and execute query

# Techniques

Discussed:

- Query execution architectures
- Parallelism
- Hashing vs. sorting
- Algorithms, execution costs etc.

Not discussed:

- Recursive queries, optimization

# Techniques

Discussed:

- Query execution architectures
- Parallelism
- Hashing vs. sorting
- Algorithms, execution costs etc.

Not discussed:

- Recursive queries, optimization

# Query Execution Architecture

Focus on useful mechanisms for processing sets of items

- Records
- Tuples
- Entities
- Objects

# Physical Algebra

- Algorithms as algebra operators consuming input and and producing some output

- Physical Algebra - Query processing algorithms as a whole

# Physical vs. Logical Algebra

- Logical algebra: related to data model and defines what queries can be expressed in data model
  - Example: Relational algebra
- Physical algebra: system specific
  - Different systems may implement the same data model and the same logical algebra but may use different physical algebras
  - Example: Loops joins vs. hash joins

# Physical vs. Logical Algebra

- Cost functions are associated with physical operators only
  - Need to map logical operators to physical to determine cost
  - Query Optimization: mapping from logical to physical
  - Mapping process is guided by meta-data

# Issues While Mapping

- Binding
  - Whether to bind at start-up or compile time
- Synchronization and data transfer between operators
  - Temporary files vs. IPC
  - Rule-based translation programs
  - Schedule all operators in  a single operating system process => iterators

# Iterators

Prepare an operator for producing data

- Open

Produce an item

- Next

Perform final housekeeping

- Close

# Iterators

- Entire query plan executed in a single process
- Operators produce an item at a time on request
- Items never wait in a temporary file or buffer (pipelining)

# Iterators

- Efficient in time-space-product memory cost
- Can schedule any type of trees including bushy trees
  - Operators expressed as trees/DAGs
- No operator affected by the complexity of the whole plan

# Sorting vs. Hashing

- Purpose of many query-processing algorithms is to perform some kind of matching
  - Indexing, joins, aggregation, parallelization
- Two approaches
  - Sorting
  - Hashing
  - Both are memory-intensive

# Sorting: Design Issues

- Implemented as sorted runs
  - Merge sorted runs until all data is sorted
- Implement as iterator
  - Interfaces well with other operators
- Input is also an iterator
  - Can come from a scan or a complex query plan
- If data fits in memory, can use quicksort
  - Usually, exploit duality between mergesort & quicksort

# Sorting: Details

- Sorting large DBs
  - Sorting within main memory
  - Managing subsets of data on disk/tape
- Typically used - Physical dividing and logical combining
- Creating initial runs => *level 0* runs
  - In-memory sort algo like quicksort
  - Or, replacement selection

# Quicksort vs. Replacement Selection

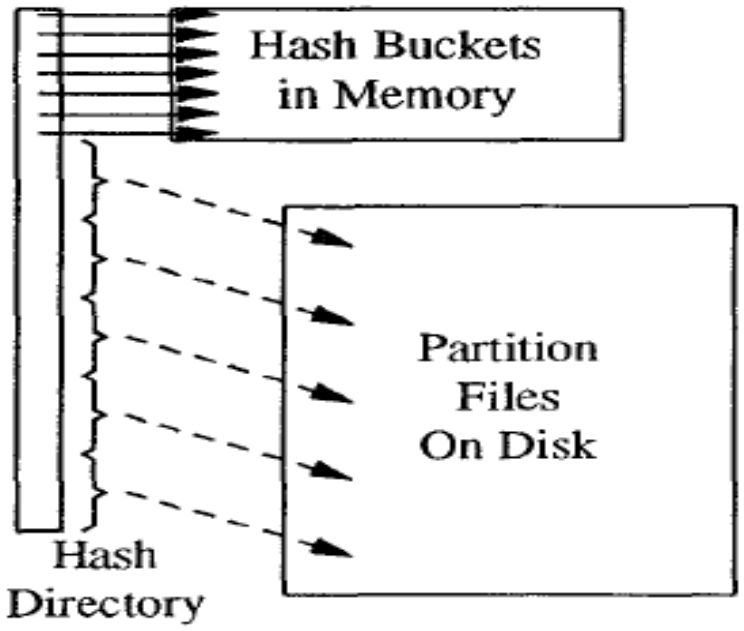
- Run files
  - Larger than memory in RS; size of the memory in QS
- Reads and writes:
  - RS alternates between both; QS does them in bursts
- Memory management
  - More complex in RS
  - Advantage of fewer runs must be balanced with the different I/0 pattern and the disadvantage of complex memory management

# Hashing: Design Issues

- Alternative to sorting
- Expected complexity of hashing algorithms is O(N) while for sorting, it is O(N log N)
- In-memory hash table
  - If entire table fits, hash-based algos are easy to design, understand, and implement
  - For binary operations, only one input needs to fit
  - If hash table is larger => hash overflow occurs

# Hashing: Hash Overflow

- Managing hash overflow
  - Avoidance
  - Resolution
  - Both involve partitioning
- Partitions are processed independently and concatenated to get final result



# Aggregation

- Important for summarizing data
- Aggregate functions: min, max, sum, etc.
- Duplicate removal is similar
  - Data needs to be compared before removal
- In many systems, aggregation and duplicate removal is based on sorting

# Nested Loops Join

- For each item in one input, scan entire other input to find matches
- Performance is poor; because inner input is scanned often
- Tricks to improve performance
  - Larger input should be the outer one
  - If possible, use an index on the attribute to be matched in the inner input
  - Scan inner input once for each ‘page’ of outer input

# Merge Join

- Requires both inputs sorted on the join attribute
- Requires keeping track of interesting orderings
- Hybrid join (used by IBM for DB2), uses elements from index nested-loop joins and merge join, and techniques joining sorted lists on index leaf entries

# Hash Join

- Based on in-memory hash table on one input (smaller one, called ‘build input’), and probing this table using items from the other input (called ‘probe input’)
- Very fast if build input fits into memory, regardless of size of probe input
- Overflow avoidance methods needed for larger build inputs

# Hash Join

- Both inputs partitioned using same partitioning function. Final join result formed by concatenating join results of pairs of partitioning files
- Recursive partitioning may be used for both inputs
- More effective when the two input sizes are very different (smaller being the build input)

# Universal Quantification

- Algorithms for relational division
- Can be easily replaced by aggregation
- Methods for universal quantification
  - Direct using sort
  - Direct using hash
  - Aggregation using sort
  - Aggregation using hash

# Summary

The choice of Hash based or Sort based should be based on relative sizes of inputs and the danger of performance loss due to skewed data or hash value distribution.