Query Execution
Query processing

Query processing involves

- compilation
  - parsing to construct parse tree
  - optimization
    - Query rewrite to generate a logical query plan
    - Physical plan generation to make a physical query plan

Execution

Query plans are expression trees whose nodes are operators

- Extended relational algebra operators for logical query plans
- Physical operators for physical query plans
  - Implement extended relational algebra operators
  - Additional useful tasks
Types of operators

Physical operators classified according to

- #passes over the tuples of their input
  - One-pass
  - Pass-and-a-half
  - Two-pass
  - Multi-pass

- Their cardinality and #input tuples required at once
  - Unary
    - Tuple-at-a-time
      - Select, project
    - full-relation
      - Grouping/aggregates, duplicate elimination
  - Binary
    - Join, union, difference, intersection
Physical operators as iterators

Physical operators are viewed as iterators, with methods

- **Open()**
  - Initializes any structures for the operator

- **Next()**
  - Returns the next tuple in the stream of output tuples

- **Close()**
  - Destroys any structures created

The iterator model allows to non-materialize the output of operators, unless necessary or useful

- It allows for complete materialization by doing all the work in the Open() method

Some tasks fit naturally into the iterator model, some need tricks

- Sorting (multi-way merge-sort) does most of the work in the open method
Operator model

- Measure cost of operator in terms of block I/Os performed
- Assume that the output of each operator is immediately consumed by some other operator
- Cost of operator uses parameters
  - Available #memory buffers (blocks) \( M \)
  - For each argument (bag or set of tuples) \( R \)
    - The #blocks \( B(R) \)
    - The #tuples \( T(R) \)
    - The #distinct values \( V(R,X) \) of the tuples of \( R \) on the attribute-list \( X \)
- To estimate cost of an operator need to know
  - whether \( R \) is clustered, i.e. whether occupies about \( B(R) \) blocks or not
  - A used index on \( R \) is a clustering index, i.e. tuples with same value on the indexed attributes occupy as few blocks as possible
  - Safe to assume that output relations of operators are clustered!
    - often times input relations will also be assumed clustered as well!!
Operators for scanning relations

Read the contents of relation R

- **Table-scan**
  - reads the tuples of R from disk, using a system map of its blocks

- **Index-scan**
  - reads the tuples using an index to locate the blocks that contain its tuples

- **Sort-scan**
  - Return its tuples in a sorted order

What is the cost of each of these operators?

- Remember that we can
  - sort a relation with $B \leq M^2$ blocks with $3B$ block I/Os
One-pass tuple-at-a-time unary operators

- Obvious algorithms for
  - Project
  - Select
- Have cost of $B(R)$ regardless of $M$
One-pass full-relation unary operators

- Duplicate elimination

Assumes $B(\delta(R)) \leq M$

Similarly for grouping/aggregation

- Keep sufficient data for each distinct group and output aggregate value for each group at Close()

  - Assumes info for #groups fits M-1 buffers
**One-pass binary operators**

- Given relations R and S, with S being the smaller of the two
- Assume S fits in memory (M-1) buffers
- Bag union
  - Read each tuple from S, output it; repeat for R
- Set union
  - **Algorithm**
    1. Read each tuple of S and build an in-memory dictionary for S
    2. Output all of S
    3. Read each tuple t of R and output it if t is not in the dictionary for S
- Set intersection, difference are similar
  - Care is needed when computing set difference
    - R-S
    - S-R
One-pass binary operators

Bag intersection
- Each tuple appears in the result a number of times equal to the minimum number of its occurrences in either input relation

Algorithm
- Modify the algorithm for set union to
  - Build dictionary for the tuples of S
    - maintain the #occurrences of each tuple of S
  - Read each tuple t of R and
    - If t is in the dictionary, output t and decrement its count
    - Remove t from the dictionary if its count is 0

Bag difference, Cartesian product, Natural join are similar

The cost of all these operators is $B(S)+B(R)$

What happens if M is “wrong”?
- Thrashing or unnecessary passes
**Nested-Loop Join**

- Natural join $R(X,Y) \bowtie S(Y,Z)$
- Tuple-based nested-loop join
  - *for each tuple $t_s$ in $S$ do begin*
    - *for each tuple $t_R$ in $R$ do begin*
      - *if $t_s$ and $t_R$ join to make $t$*
      - output $t$
  - $S$ is called the **outer relation** and $R$ the **inner relation** of the join
- Cost is $T(R)T(S)$
  - Expensive since it examines every pair of tuples in the two relations
- Fits the iterator framework easily
  - reopen the scan of $R$ for each tuple of $S$!
- Requires no indices and can be used with any kind of join condition
Block-based nested-loop join

- Read a block instead of a single tuple at a time
  - Join all tuples in the pair of blocks at hand
- If S fits in M-1 buffers, make it the inner relation
  - Cost now is $B(R) + B(S)$
- If neither S nor R fit in memory and the S (the smallest) is the outer relation then the cost is

$$\frac{B(S)}{M - 1} (M - 1 + B(R)) = B(S) + \frac{B(S) \cdot B(R)}{M - 1}$$
Two-pass algorithms

- Input relations may be too large for the one-pass algorithms to handle

- Consider two-pass algorithms
  - Multi-pass algorithms can be obtained easily by induction/recursion from two-pass algorithms

- Focus on two-pass algorithms that are based on
  - Sorting
  - Hashing
  - Indexing
Two-pass duplicate elimination using sorting

Algorithm

- Partition $R$ into at most $M$ sublists each of size at most $M$, and write to disk each sorted sublist
- Read one block from each sorted sublist
- For each tuple $t$ still in memory do
  - Output $t$ and then delete all its occurrences from any blocks in memory
    - If a block becomes empty load it with another block from the same sorted sublist
    - Delete $t$ from each newly loaded block as well

- Cost is $2B(R) + B(R) = 3B(R)$

Requires that $B(R) \leq M^2$
Two-pass grouping and aggregation using sorting

Similar to the duplicate elimination algorithm

- Sort based on the grouping attributes $X$ only
- Compute aggregate value for each distinct value of $X$, in sorted order
  - Instead of deleting all other occurrences of a tuple $t$ as in duplicate elimination, update the aggregate information for the group $t[X]$
Two-pass set-union using sorting

- Two-pass set union algorithm using sorting
  - Make sorted sublists for both R and S
  - Use a buffer for each sorted sublist of either R or S
    - Load each buffer with the 1\textsuperscript{st} block of its sublist
  - Repeatedly, for each tuple t in memory
    - Output t
    - Delete t from the memory buffers
      - If a buffer empties reloaded with the next block from the same sublist
      - delete all occurrences of t from each newly loaded buffer as well

Cost is \(3(B(R) + B(S))\)

Requires that \(B(R) + B(S) \leq M^2\)
Two-pass intersection and difference using sorting

- Modify when t is output in the 2-pass sort-based set-union algorithm
- Set-intersection: output when t appears in both R and S
- Set-difference R-S: output when t appears in R but not in S
- Bag-intersection/difference
  - keep track of \( g(t,R) \) of t in each relation R while making the sorted sublists
  - For intersection, output t \( \min(g(t,S), g(t,R)) \) times
  - For difference R-S, output t \( \max(g(t,R)-g(t,S),0) \) times
Simple sort-based join

Algorithm

- Sort R and S
- Load two input buffers with the 1st block of R and S
- for each least value y of the join attributes Y in memory
  - Find all tuples from R and S with value y on Y, and output their join
  - use all remaining memory to read qualifying tuples from R and S
  - Delete all tuples with value y on attributes Y from memory
  - Reload any of the two input buffers with the next block from its corresponding relation

Cost is $5(B(R) + B(S))$

Requires that $B(R) \leq M^2$ and $B(S) \leq M^2$

Algorithm is a good choice when there are a lot of joinable tuples for each value y
More efficient sort-based join

- Previous algorithm can be modified by
  - Writing the sorted sublists to disk instead of the sorted relations
  - Allocating one buffer to each sorted sublist

- Advantage
  
  Cost is $3(B(R) + B(S))$

  Requires that $B(R) + B(S) \leq M^2$

- Disadvantage
  
  the algorithm is bad when there are very large number of tuples with a common value for the join attributes
Two-pass algorithms using hashing

- Data does not fit in memory

- Partition data into $M$ buckets using a hash function so that
  - A bucket or a pair of buckets with the same hash, one from each relation, fits in memory

- Perform the required operation in one-pass on
  - a single bucket for unary operators
  - On a pair of buckets for binary operators

- Duplicate elimination, set union, difference, intersection, and join can all be done this way

\[
\text{Cost is } 3(B(R) + B(S))
\]

\[
\text{Requires that } \min(B(R), B(S)) \leq M^2
\]
Index-based selection

- Equality-selection can be evaluated using an index (index-scan)

\[ \sigma_{v=a}(R) \]

- Cost for
  - Clustering index is \( h + \frac{B(R)}{V(R, v)} \)
  - Non-clustering index is \( h + \frac{T(R)}{V(R, v)} \)

- Where \( h \) is the \#block I/Os to access the index
  - Typically \( h \) is <5 for either B-tree or hash indexes

- Additional selection predicates can be evaluated using certain indexes

\[ \sigma_{a \leq v \leq b}(R) \quad \sigma_{v \neq a}(R) \]
**Index-based join**

- Use index to access tuples of the inner relation in tuple-based nested-loop join
  - Cost is approximately \( T(R)B(S)/V(S,Y) \) or \( T(R)T(S)/V(S,Y) \) depending on whether index is clustering or not

- If index allows for sorted access to the tuples of a relation, e.g., B-tree index
  - Use the 2\(^{nd}\) phase ("merge phase") of sorted-based join algorithms
    - Since relation(s) are already accessible in sorted order
    - Cost is \( B(R) + B(S) \) if both the indexes are clustering

- Other operations (duplicate elimination, set difference/union/intersection, grouping and aggregation) can be done similarly
Buffer management

- Physical operators use some #buffers M
- The DBMS buffer manager allocates available buffers to operators/queries from a buffer pool in
  - physical memory directly
  - Virtual memory managed by the Operating System
- Buffer pool is limited so careful management is needed to avoid
  - Thrashing
  - Unnecessary performance degradation
- Buffer replacement strategies
  - LRU
  - FIFO
  - MRU
  - Clock
  - System control – pinned blocks
Buffer manager & physical operators

Considering the algorithm of a physical operator

- How does it respond to changes in the number $M$ of available buffers?
  - when $M$ is not what it was assumed initially?
  - changes during the execution of the operator?
    - More buffers become available
    - A used buffer is taken away from the operator by the buffer manager

- How can the operator affect the decisions of the buffer manager?
  - Can it suggest
    - a replacement strategy?
    - a range of buffers needed for good performance?
    - A function to forecast its performance for any parameter value $M$?
  - Pin blocks?
Multi-pass algorithms

- The two-pass algorithms discussed generalize to $k$-passes using recursion/induction
  - Sort-based algorithms
    Cost is $(2k - 1)(B(R) + B(S))$ provided that $B(R) + B(S) \leq M^k$
  - Hashing-based algorithms
    Cost is $(2k - 1)(B(R) + B(S))$ provided that $\min(B(R), B(S)) \leq M^k$
Models of parallel machines

Parallel machines with p processors can be

- Shared memory model
- Shared disk (NUMA) model
- Shared nothing model
Parallelism for physical operators

Idea is to
- distribute the input relations to processors so that each one gets about $1/p$ of the input
- Each processor works on its given data
- Ensure the communication overhead and synchronization barriers are under control
  - sending a block over the network is typically faster than a block I/O
Parallel sort-based join

Partition and ship data to processors

- Total #blocks shipped $\frac{p-1}{p} (B(R) + B(S))$
- #block I/Os per processor $(B(R) + B(S))/p$

Each processor

- Stores the tuples it receives at cost $(B(R) + B(S))/p$
- Applies sort-based join on its fragment of the data

$$3 \left( B(R) + B(S) \right)/p \text{ provided that } B(R) + B(S) \leq pM^2$$

Total #block I/Os per processor

$$5 \left( B(R) + B(S) \right)/p \text{ provided that } B(R) + B(S) \leq pM^2$$

A speedup by a factor of about $p!!$