Chapter 15: Query Execution
Overview of Query Processing

- SQL query
  - parse query
  - query expression tree
  - select logical query plan
  - logical query plan tree
  - select physical query plan
  - physical query plan tree
  - execute physical query plan

- Query Compilation (Ch 16)
- Query Optimization
- Query Execution (Ch 15)
Overview of Query Compilation

- convert SQL query into a **parse tree**
- convert parse tree into a **logical query plan**
- convert logical query plan into a **physical query plan**:  
  - choose algorithms to implement each operator of the logical plan
  - choose order of execution of the operators
  - decide how data will be passed between operations
- **Choices depend on metadata**:  
  - size of the relations
  - approximate number and frequency of different values for attributes
  - existence of indexes
  - data layout on disk
Operations (steps) of query plan are represented using relational algebra (with bag semantics)

Describe efficient algorithms to implement the relational algebra operations

Major approaches are scanning, hashing, sorting and indexing

Algorithms differ depending on how much main memory is available
Remember!
Relational Algebra Summary

- Set operations: union $U$, intersection $\cap$, difference $-$
- projection, $\pi$ (choose columns/atts)
- selection, $\sigma$ (choose rows/tuples)
- Cartesian product $X$
- natural join (bowtie, $\bowtie$) : pair only those tuples that agree in the designated attributes
- renaming, $\rho$
- duplicate elimination, $\delta$
- grouping and aggregation, $\gamma$
- sorting, $\tau$
The Hive

Main memory

HDD
Measuring Costs

- Parameters:
  - $M$: number of main-memory buffers available (size of buffer = size of disk block). *Only count space needed for input and intermediate results, not output!*
  - For relation $R$:
    - $B(R)$ or just $B$: number of blocks to store $R$
    - $T(R)$ or just $T$: number of tuples in $R$
    - $V(R,a)$: number of distinct values for attribute $a$ appearing in $R$

- Quantity being measured: *number of disk I/Os.*
  - *Assume inputs are on disk but output is not written to disk.*
Scan Primitive

- Reads entire contents of relation R
- Needed for doing join, union, etc.
- To find all tuples of R:
  - *Table scan*: if addresses of blocks containing R are known and contiguous, easy to retrieve the tuples
  - *Index scan*: if there is an index on any attribute of R, use it to retrieve the tuples
Costs of Scan Operators

- **Table scan:**
  - if $R$ is clustered, then number of disk I/Os is approx. $B(R)$.
  - if $R$ is not clustered, number of disk I/Os could be as large as $T(R)$.

- **Index scan:** approx. same as for table scan, since the number of disk I/Os to examine entire index is usually much much smaller than $B(R)$. 
Sort-Scan Primitive

- Produces tuples of $R$ in sorted order w.r.t. attribute $a$
- Needed for sorting operator as well as helping in other algorithms
- Approaches:
  1. If there is an index on $a$ or if $R$ is stored in sorted order of $a$, then use index or table scan.
  2. If $R$ fits in main memory, retrieve all tuples with table or index scan and then sort
  3. Otherwise can use a secondary storage sorting algorithm (cf. Section 11.4.3)
Costs of Sort-Scan

- See earlier slide for costs of table and index scans in case of clustered and unclustered files.

- Cost of secondary sorting algorithm is:
  - approx. $3B$ disk I/Os if $R$ is clustered
  - approx. $T + 2B$ disk I/Os if $R$ is not
Categorizing Algorithms

- By general technique
  - sorting-based
  - hash-based
  - index-based

- By the number of times data is read from disk
  - one-pass
  - two-pass
  - multi-pass (more than 2)

- By what the operators work on
  - tuple-at-a-time, unary
  - full-relation, unary
  - full-relation, binary
One-Pass, Tuple-at-a-Time

- These are for SELECT and PROJECT
- Algorithm:
  - read the blocks of R sequentially into an input buffer
  - perform the operation
  - move the selected/projected tuples to an output buffer
- Requires only \( M \geq 1 \)
- I/O cost is that of a scan (either \( B \) or \( T \), depending on if \( R \) is clustered or not)
- **Exception!** Selecting tuples that satisfy some condition on an indexed attribute can be done faster!
One-Pass, Unary, Full-Relation

- duplicate elimination (DELTA)

Algorithm:

- keep a main memory search data structure $D$ (use search tree or hash table) to store one copy of each tuple
- read in each block of $R$ one at a time (use scan)
- for each tuple check if it appears in $D$
- if not then add it to $D$ and to the output buffer

- Requires 1 buffer to hold current block of $R$; remaining $M-1$ buffers must be able to hold $D$

- I/O cost is just that of the scan
One Pass, Unary, Full-Relation

- grouping (GAMMA)
- Algorithm:
  - keep a main memory search structure $D$ with one entry for each group containing
    - values of grouping attributes
    - accumulated values for the aggregations
  - scan tuples of $R$, one block at a time
  - for each tuple, update accumulated values
    - MIN/MAX: keep track of smallest/largest seen so far
    - COUNT: increment by 1
    - SUM: add value to accumulated sum
    - AVG: keep sum and count; at the end, divide
  - write result tuple for each group to output buffer
No generic bound on main memory required:
- group entries could be larger than tuples
- number of groups can be anything up to $T$ but typically
- group entries are not longer than tuples
- many fewer groups than tuples

Disk I/O cost is that of the scan
One Pass, Binary Operations

- **Bag union:**
  - copy every tuple of $R$ to the output, then copy every tuple of $S$ to the output
  - only needs $M \geq 1$
  - disk I/O cost is $B(R) + B(S)$

- For set union, set intersection, set difference, bag intersection, bag difference, product, and natural join:
  - read smaller relation into main memory
  - use main memory search structure $D$ to allow tuples to be inserted and found quickly
  - needs approx. $\min(B(R), B(S))$ buffers
  - disk I/O cost is $B(R) + B(S)$
One Pass, Binary Operations

- **Bag union:**
  - copy every tuple of $R$ to the output, then copy every tuple of $S$ to the output
  - only needs $M \geq 1$
  - disk I/O cost is $B(R) + B(S)$

- For set union, set intersection, set difference, bag intersection, bag difference, product, and natural join:
  - read smaller relation into main memory
  - use main memory search structure $D$ to allow tuples to be inserted and found quickly
  - needs approx. $\min(B(R), B(S))$ buffers
  - disk I/O cost is $B(R) + B(S)$
Set Union \((R \cup U \cup S)\)

- Assume \(S\) fits in \(M-1\) main memory buffers
- read \(S\) into main memory
- for each tuple of \(S\)
  - insert it into a search structure \(D\) (key is entire tuple)
  - copy it to output
- read each block of \(R\) into 1 buffer one at a time
- for each tuple of \(R\)
  - if it is not in \(D\) (i.e., not in \(S\)) then copy to output
Set Intersection ($R \cap S$)

- Assume $S$ fits in $M-1$ main memory buffers
- read $S$ into main memory
- for each tuple of $S$
  - insert it into a search structure $D$ (key is entire tuple)
- read each block of $R$ into 1 buffer one at a time
- for each tuple of $R$
  - if it is in $D$ (i.e., in $S$) then copy to output
Set Difference (R - S)

- Assume S fits in M-1 main memory buffers
- read S into main memory
- for each tuple of S
  - insert into a search structure D (key is entire tuple)
- read each block of R into 1 buffer one at a time
- for each tuple of R
  - if it is not in D (i.e., not in S) then copy to output
Additional Binary Operations

- See text for one-pass algorithms for
  - $S - R$
  - bag intersection
  - bag difference
  - product

- Similar to the previous algorithms
- Require that one of the operands fit in main memory
Assume $R(X,Y)$ is to be joined with $S(Y,Z)$ and $S$ fits in $M-1$ main memory buffers
- read $S$ into main memory
- for each tuple of $S$
  - insert into a search structure $D$ (key is atts in $Y$)
- read each block of $R$ into 1 buffer one at a time
- for each tuple $t$ of $R$
  - use $D$ to find all tuples of $S$ that agree with $t$ on atts $Y$
  - for each matching tuple $u$ of $S$, concatenate $t$ and $u$ and copy to output
What Ifs?

- What if data is not clustered?
  - Then it takes $T(R)$ disk I/Os instead of $B(R)$ to read all the tuples of $R$
  - But any relation that is the result of an operator will be stored clustered

- What if $M$ is unknown/wrongly estimated?
  - If over-estimated, then one-pass algorithm will be very slow due to thrashing between disk and main memory
  - If under-estimated and a two-pass algorithm is used when a one-pass would have sufficed, unnecessary disk I/Os are done
Query Processing

Two-Pass Algorithms
Nested-Loop Joins

Another helpful primitive.
Main idea to join relations $R$ and $S$:

for each tuple $s$ in $S$ do
  for each tuple $r$ in $R$ do
    if $r$ and $s$ match then
      output the join of $r$ and $s$

How to optimize this?
for each chunk of $M-1$ blocks of $S$ do
read the blocks into main memory
create search structure $D$ for the tuples with the common attributes of $R$ and $S$ as the key
for each block $b$ of $R$ do
read $b$ into $M$-th block of main memory
for each tuple $t$ of $b$ do
use $D$ to find each tuple $s$ of $S$ that joins with $t$
output the result of joining $t$ with $s$
Analysis of Nested-Loop Join

- Assume $S$ is the smaller relation
- number of disk I/O's:
  - number of disk I/O's per iteration of outer loop times number of iterations of outer loop
  - this is: $\frac{(M - 1 + B(R)) \times B(S)}{(M-1)}$
- If $B(S) \leq M-1$, then this is same as one-pass join algorithm.
Two-Pass Algorithms

- Used when relations are too large to fit in memory
- Pieces of the relation are read into memory, processed in some way, and then written back to disk
- The pieces are then reread from disk to complete the operation
- Why only two passes? Usually this is enough, but ideas can be extended to more passes.
Another lonely day.
Using Sorting as a Tool

- Suppose we have $M$ blocks of main memory available, but $B(R) > M$
- Repeatedly
  - read $M$ blocks of $R$ into main memory
  - sort these blocks (shouldn't take more time than one disk I/O)
  - write sorted blocks to disk; called sorted sublists
- Do a second pass to process the sorted sublists in some way
- Usually require there are at most $M$ sublists, i.e., $B(R) \leq M^2$. Gaining a factor of $M$ using preprocessing!
Duplicate Elimination Using Sorting

- Create the sorted sublists
- Read first block of each sublist into memory
- Look at first unconsidered tuple in each block and let $t$ be first one in sorted order
- Copy $t$ to the output and delete all other copies of $t$ in the blocks in memory
- If a block is emptied, then bring into its buffer the next block of that sublist and delete any copies of $t$ in it
Analysis of Duplicate Elimination

- $B(R)$ disk I/Os to create sorted sublists
- $B(R)$ disk I/Os to write sorted sublists to disk
- $B(R)$ disk I/Os to reread each block from the sublists
- Grand total is $3*B(R)$ disk I/Os.
- Remember we need $\sqrt{B(R)}$ blocks of main memory.
Grouping Using Sorting

- Create the sorted sublists on disk, using the grouping attributes as the sort key
- Read the first block of each sublist into memory
- Repeat until all blocks have been processed:
  - Start a new group for the smallest sort key among next available tuples in the buffers
  - Compute the aggregates using all tuples in this group -- they are either in memory or will be loaded into memory next
  - Output the tuple for this group

$3 \cdot B(R)$ disk I/Os, $\sqrt{B(R)}$ blocks of main memory
Union Using Sorting

- Create the sorted sublists for $R$ on disk, using the entire tuple as the sort key
- Create the sorted sublists for $S$ on disk, using the entire tuple as the sort key
- Use one main memory buffer for each sublist of $R$ and $S$
- Repeatedly find next tuple $t$ among all buffers, copy to output, and remove from the buffers all copies of $t$ (reloading any buffer that is emptied)

- $3(B(R) + B(S))$ disk I/Os; $B(R) + B(S) \leq M^2$
Intersection and Difference Using Sorting

- Very similar to Union.
- Different rules for deciding whether/how many times a tuple is output
- **Set Intersection**: output \( t \) if it appears in both \( R \) and \( S \)
- **Bag Intersection**: output \( t \) the minimum number of times it appears in \( R \) and \( S \)
- **Difference**: see text.
- \( 3(B(R) + B(S)) \) disk I/Os; \( B(R) + B(S) \leq M^2 \)
A Sort-Based Join

- **Goal:** make as many main memory buffers as possible available for joining tuples with a common value

- To join $R(X,Y)$ and $S(Y,Z)$:
  - Totally sort $R$ using $Y$ as the sort key
  - Totally sort $S$ using $Y$ as the sort key
  - Next pass reads in blocks of $R$ and $S$, primarily one at a time, i.e. using one buffer for $R$ and one for $S$. Familiar strategy is used to reload the buffer for a relation when all the current tuples have been processed.
Simple Sort-Based Join (cont'd)

- Let \( y \) be the smaller sort key at the front of the buffers for \( R \) and \( S \)
- If \( y \) appears in both relations then
  - load into memory all tuples from \( R \) and \( S \) with sort key \( y \); up to \( M \) buffers are available for this step (*)
  - output all tuples formed by combining tuples from \( R \) and tuples from \( S \) with sort key \( y \)
More on Sort-Based Join

(*) Suppose not all tuples with sort key $y$ fit in main memory.

If all the tuples in one relation with sort key $y$ do fit, then do one-pass join

If all the tuples in neither relation with sort key $y$ fit, then do basic nested-loop join
Analysis of Sort-Based Join

- Suppose all tuples with a given sort key fit in main memory.
- The dominant expense of the algorithm is the secondary storage sorting algorithm used to sort the relations:
  - $5(B(R) + B(S))$ disk I/Os
  - $B(R) \leq M^2$ and $B(S) \leq M^2$
Two-Pass Algorithms Using Hashing

- **General idea:**
  - Hash the tuples using an appropriate hash key
  - For the common operations, there is a way to choose the hash key so that all tuples that need to be considered together has the same hash value
  - Do the operation working on one bucket at a time
Partitioning by Hashing

initialize $M-1$ buckets with $M-1$ empty buffers
for each block $b$ of relation $R$ do
  read block $b$ into the $M$th buffer
  for each tuple $t$ in $b$ do
    if the buffer for bucket $h(t)$ is full then
      copy the buffer to disk
      initialize a new empty block in that buffer
    copy $t$ to the buffer for bucket $h(t)$
  for each bucket do
    if the buffer for this bucket is not empty then
      write the buffer to disk
Duplicate Elimination Using Hashing

- Hash the relation R to M-1 buckets, $R_1, R_2, \ldots, R_{M-1}$
- Note: all copies of the same tuple will hash to the same bucket!
- Do duplicate elimination on each bucket $R_i$ independently, using one-pass algorithm
- Return the union of the individual bucket results
Number of disk I/O's: $3*B(R)$

In order for this to work, we need:

- hash function $h$ evenly distributes the tuples among the buckets
- each bucket $R_i$ fits in main memory (to allow the one-pass algorithm)
- i.e., $B(R) \leq M^2$
Grouping Using Hashing

- Hash all the tuples of relation $R$ to $M-1$ buckets, using a hash function that depends only on the grouping attributes.
- Note: all tuples in the same group end up in the same bucket!
- Use the one-pass algorithm to process each bucket independently.
- Uses $3*B(R)$ disk I/O's, requires $B(R) \leq M^2$.


Use same hash function for both relations!

- Hash $R$ to $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
- Hash $S$ to $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
- Do one-pass \{set union, set intersection, bag intersection, set difference, bag difference\} algorithm on $R_i$ and $S_i$, for all $i$

- $3(B(R) + B(S))$ disk I/O's; $\min(B(R), B(S)) \leq M^2$
Join Using Hashing

- Use same hash function for both relations; hash function should depend only on the join attributes

- Hash $R$ to $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
- Hash $S$ to $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
- Do one-pass join of $R_i$ and $S_i$, for all $i$

- $3*(B(R) + B(S))$ disk I/O's; $\min(B(R),B(S)) \leq M^2$
Comparison of Sort-Based and Hash-Based

- For binary operations, hash-based only limits size of smaller relation, not sum
- Sort-based can produce output in sorted order, which can be helpful
- Hash-based depends on buckets being of equal size
- Sort-based algorithms can experience reduced rotational latency or seek time
Index based algorithms
The existence of an index is especially helpful for selection, and helps others

- **Clustered relation**: tuples are packed into the minimum number of blocks

- **Clustering index**: all tuples with the same value for the index's search key are packed into the minimum number of blocks
Index-Based Selection

- Without an index, selection takes $B(R)$, or even $T(R)$, disk I/O's.
- To select all tuples with attribute $a$ equal to value $v$, when there is an index on $a$:
  - search the index for value $v$ and get pointers to exactly the blocks containing the desired tuples
- If index is clustering, then number of disk I/O's is about $B(R)/V(R,a)$
Suppose $B(R) = 1000$, $T(R) = 20,000$, there is an index on $a$ and we want to select all tuples with $a = 0$.

- If $R$ is clustered and don't use index: 1000 disk I/O's
- If $R$ is not clustered and don't use index: 20,000 disk I/O's
- If $V(R,a) = 100$, index is clustering, and use index: $1000/100 = 10$ disk I/O's (on average)
- If $V(R,a) = 10$, $R$ is not clustered, index is non-clustering, and use index: $20,000/10 = 2000$ disk I/O's (on average)
- If $V(R,a) = 20,000$ ($a$ is a key) and use index: 1 disk I/O
Using Indexes in Other Operations

1. If the index is a B-tree, can efficiently select tuples with indexed attribute in a range.
2. If selection is on a complex condition such as "\(a = v \text{ AND } \ldots\)”, first do the index-based algorithm to get tuples satisfying "\(a = v\)".
   - Such splitting is part of the job of the query optimizer.
Consider natural join of $R(X, Y)$ and $S(Y, Z)$.
Suppose $S$ has an index on $Y$.

for each block of $R$
  for each tuple $t$ in the current block
    use index on $S$ to find tuples of $S$ that match $t$ in the attribute(s) $Y$
    output the join of these tuples
Analysis of Index-Based Join

- To get all the blocks of $R$, either $B(R)$ or $T(R)$ disk I/O's are needed.
- For each tuple of $R$, there are on average $T(S)/V(S, Y)$ matching tuples of $S$
  - $T(R)*T(S)/V(S, Y)$ disk I/O's if index is not clustering
  - $T(R)*B(S)/V(S, Y)$ disk I/O's if index is clustering
- This method is efficient if $R$ is much smaller than $S$ and $V(S, Y)$ is large (i.e., not many tuples of $S$ match)
Join Using a Sorted Index

- Suppose we want to join $R(X, Y)$ and $S(Y, Z)$.
- Suppose we have a sorted index (e.g., B-tree) on $Y$ for $R$ and $S$:
  - do sort-join but
  - no need to sort the indexed relations first
Buffer Management

- The availability of blocks (buffers) of main memory is controlled by buffer manager.
- When a new buffer is needed, a replacement policy is used to decide which existing buffer should be returned to disk.
- If the number of buffers available for an operation cannot be predicted in advance, then the algorithm chosen must degrade gracefully as the number of buffers shrinks.
- If the number of buffers available is not large enough for a two-pass algorithm, then there are generalizations to algorithms that use three or more passes.