Query Execution
1. Parse query into a relational algebra expression tree.
2. Optimize relational algebra expression tree and select implementations for the relational algebra operators. (This step involves considering alternative but equivalent expression trees and selecting the best one.)
3. Execute the plan produced by step 2.

Statistics
- \( M \) – The number of blocks we can have in memory at the same time (not counting the input buffer)
- \( B(R) \) – The number of blocks needed to store the tuples (rows, records) of relation (table, file) \( R \)
- \( T(R) \) – The number of tuples (rows, records) in relation (table, file) \( R \)
- \( V(R, a) \) – The number of distinct values for attribute \( a \) in the rows of \( R \)
- \( V(R, [a_1, \ldots, a_n]) \) – The number of distinct value combinations for attributes \( a_1, \ldots, a_n \) found in the rows of \( R \)

Table Lookup Cost
- If the rows of a table \( R \) are stored sequentially on disc, \( B(R) \) disc accesses are needed to read all the rows.
- If the rows of a table \( R \) are scattered all over the disc, up to \( T(R) \) disc accesses are needed to read all the rows.

Iterators
Operators that produce a sequence of tuples are usually organized as iterators. Each has three methods.
- \( \text{Open}() \) - initialization
- \( \text{GetNext}() \) - each call returns a tuple or \( \text{NotFound} \)
- \( \text{Close}() \) - do any necessary cleanup

Duplicate Elimination
Read each row in. Compare to stored previous outputs. If row is not among stored previous outputs, add it to stored previous outputs and output it. If row is among stored previous outputs, read in the next row and repeat.
- 1 block needed to buffer rows for output to disc or application.
- \( M-1 \) blocks are available to store previous outputs.
- If more than \( M-1 \) blocks are needed to store previous outputs, some will have to be written back to disc and retrieved as needed: very costly.
- At best, # of disc accesses is \( O(B(R)) \), at worst, \( O(B(R)^2) \). The latter can be avoided if previous outputs are stored in a \( B^+ \) tree or hash table.

Grouping
Requires \( B(R) \) disc accesses. Temporary variables needed to store intermediate results. Final aggregate value computed after all tuples have been read.
- \( \text{MIN, MAX} \) – keep current best value as tuples are read
- \( \text{COUNT} \) – increment a counter as each tuple is read
- \( \text{SUM} \) – add value of tuple (if not \( \text{NULL} \)) to an accumulator
- \( \text{AVG} \) – compute \( \text{COUNT} \) and \( \text{SUM} \) at same time; afterwards, divide sum by count.
In general, to be able to compute a binary operation on tables R and S in one pass, one of the tables has to be small enough to fit into M-1 blocks of memory. Then only $B(R) + B(S)$ disc accesses are needed in the best case.

If both tables are too large, the smaller one can be read in and temporarily stored in a B+ tree or hash table instead of storing the smaller table entirely in M-1 blocks of memory. (See two pass algorithms.)

### Set Union, Intersection

Assuming both R and S are known to be sets, copy the smaller table into M-1 blocks of memory and output its tuples, then read in the larger table one block at a time; check each tuple against the stored table, outputting those that are not in the smaller table.

To compute the set intersection of R and S, read the smaller table into the M-1 blocks of memory. Read the larger table one block at a time; if a tuple in the larger table is also in the smaller table, output the tuple, otherwise go to the next tuple.

### Set Difference

To compute R – S.

If S is smaller, read S into memory. Read R one block at a time. If a tuple is in S, ignore it. If a tuple is not in S, output it.

If R is smaller, read R into memory. Read S one block at a time. If a tuple is in R, delete it from the copy of R in memory. After S is completely read, output what remains of R in memory.

### Bag Intersection

Read the smaller table into memory, but store only one copy of each distinct tuple. Have a counter for each tuple to record the number of copies of that tuple in the bag.

Read the larger table one block at a time. If a tuple is in the smaller table and the counter for that tuple is > 0, output the tuple and decrement the counter.

### Bag Difference

To compute R – S when R and S are bags:

If R is smaller, read R into memory but store only one copy of each distinct tuple. Have a counter for each tuple to record the number of copies of that tuple in the bag. Read in S one block at a time. If a tuple is in R and the counter for the tuple is > 0, decrement the counter. When all the tuples of S have been read, output the tuples in R with counters > 0, make as many copies as indicated by the counters.

If S is smaller, read S into memory but store only one copy of each distinct tuple. Have a counter for each tuple to record the number of copies of that tuple in the bag. Read in R one block at a time. If a tuple is in S and the counter is > 0, decrement the counter. Otherwise output the tuple.

### Bag Difference cont.

If S is smaller, read S into memory but store only one copy of each distinct tuple. Have a counter for each tuple to record the number of copies of that tuple in the bag. Read in R one block at a time. If a tuple is in S and the counter is > 0, decrement the counter. Otherwise output the tuple.
To compute $R \cdot S$, read the smaller table into memory. Read the larger table one block at a time. For each tuple in the block, concatenate it with each tuple in the smaller table to produce a tuple in $R \cdot S$ and output it.

To compute $R(X,Y)$ natural join $S(Y,Z)$:
Read the smaller table into memory as a data structure searchable by $Y$ as the search key. Read the larger table one block at a time. Use the $Y$ value of a tuple to find all the tuples in the smaller table that have the same $Y$ value, and join the tuple with each of them to form output tuples.

If both tables are too big to fit in $M-1$ blocks, their join can still be computed by means of nested loops. The simplest approach is the tuple based nested-loop join.

for each tuple $s$ in $S$
for each tuple $r$ in $R$
  if $r$ and $s$ join to make a tuple $t$ then output $t$

Open() { R.Open(); S.Open(); s = S.GetNext(); } 
GetNext() {
  repeat {
    r = R.GetNext();
    if (r == NotFound) {
      R.Close();
      s = S.GetNext();
      if (s == NotFound) { return NotFound; } 
      R.Open(); 
    }
    t = join of $r$ and $s$;
    until (t is a tuple);
    return t;
  }
Close() {
  Note: disc accesses can be up to
  R.Close();
  T(R) * T(S)
  S.Close();
}

To compute the join of $R(X,Y)$ and $S(Y,Z)$:
for each $M-1$ blocks of $S$
index these blocks on attribute(s) $Y$ (using index, B+ tree or hash table)
for each block $b$ of $R$
  for each tuple $r$ of $b$
    for each $s$ found by looking up $r.Y$ in the indexed blocks
      output the join of $r$ with $s$
Note: disc accesses are $B(S)/(M-1) \cdot (M-1 + B(R))$ or approximately $B(S) \cdot B(R)/M$. 

Two-Pass Algorithms

When tables are very large, they can be processed in some way with the result written out to disc, and then those results are read in again and processed to get the desired output.

Often the first process is to sort the tuples in a table or to put them in a hash table.

Example: if \( B(R) \leq M^2(M-1) \), \( R \) can be sorted by two-pass multiway merge sort.

Read \( M \) blocks of \( R \) into memory, sort, write out into a separate file.

Merge the up-to-\( M-1 \) files by reading them all one block at a time and merging the read-in blocks. Disc accesses are \( B[\text{read}] + B[\text{write}] + B[\text{read again}] = 3B \).

Duplicate Elimination, Grouping

Duplicate elimination can be done by doing a two pass multiway merge-sort on \( R \), modified to eliminate duplicates when discovered.

Grouping and aggregation can be done in two passes by sorting \( R \) on the grouping attributes \( L \), then reading the blocks back and stepping through the tuples. While the tuples that are read have the same values for \( L \), the aggregate function is computed. As soon as a tuple with different value for \( L \) is seen, the output for the previous value of \( L \) is produced and the next group is started.

Union, Intersection, Difference

For union: sort \( R \) and \( S \), merge (without duplicates) the two sorted files.

For intersection: sort \( R \) and \( S \), do merge but output a tuple \( t \) only if it is in both sorted \( R \) and sorted \( S \).

For difference: Sort \( R \) and \( S \), do merge but output a tuple \( t \) only if it is in sorted \( R \) but not in sorted \( S \).

These operations can be sped up if the merge phase of sorting is not done; instead, the sorted sublists of \( R \) and \( S \) are read back into memory one block at a time and the least remaining tuple in the sublists for \( R \) is combined with the least remaining tuple in the sublists for \( S \) according to the operation being performed.

Join

To compute join of \( R(X,Y) \) and \( S(Y,Z) \) (if not too large)

Sort \( R \) on \( Y \), sort \( S \) on \( Y \).

Using two buffers, read sorted \( R \) and \( S \) one block at a time.

Find the smallest \( y \) such that a tuple \( r \) from \( R \) and a tuple \( s \) from \( S \) have \( r.Y = s.Y = y \).

Get all the \( R \)-tuples \( r \) with \( r.Y = y \) and \( S \)-tuples \( s \) with \( s.Y = y \).

Form all possible joins of these tuples and output them.

Ignore all other tuples.

2 reads, 2 writes per block needed to sort. One more read per block to join; total cost = \( 5-5(B(R) + B(S)) \).

Sort-Join

Read in \( R \) and \( S \) \( M \) blocks at a time, sort on \( Y \) and write them out to disc. If no more than \( M-1 \) files were written to,

Read in the first block of each file.

Find least \( y \) such that there is an \( R \)-tuple \( r \) and an \( S \)-tuple \( s \) such that \( r.Y = s.Y = y \).

Find all the \( R \)-tuples \( r \) and \( S \)-tuples \( s \) with \( r.Y = s.Y = y \), form all possible joins and output them.

Continue reading, finding the next least \( y \) as above and repeat.

Read in more blocks when all tuples in a block have been examined/used.

Requires \( 3-5(B(R) + B(S)) \) disc accesses.

Using Hash Tables

Read in a relation \( R \) one block at a time and insert the tuples into a chained hash table having \( M-1 \) buckets. Write a block to disc only when extending a chain.

(When two such tables are used, they the buckets are also written to disc after the table is finished; the buffers will be read back in one block at a time.)

Requires about \( B(R) \) block reads to construct hash table; about \( 2B(R) \) if hash table is written out to disc.

Whatever key was used, only tuples within one bucket have to be compared to find tuples with matching keys.
**One-Pass Algorithms**

Duplicate elimination:
Use the whole tuple as the key when making hash table.
Remove duplicates within each bucket, output remaining tuples.

Grouping, aggregation:
Use grouping attribute(s) as key when making hash table.
Compute aggregate of each group; each group fits in one bucket. Output the group and aggregate value tuples.

**Two-Pass Algorithms**

When working with two relations R and S, each is put in a chained hash table and then written to disk. The binary operation being computed just needs to be done between corresponding buffers in the two hash tables.
Only $3 \times (B(R) + B(S))$ disk accesses needed (one read to make hash table, one write to store hash table, one read to get buckets back one block at a time).

**Union, Intersection, Difference**

Use the whole tuple as the key.

**Union:**
Form union of corresponding buckets, output tuples.

**Intersection:**
Compute intersection of corresponding buckets, output tuples.

**Difference:**
Compute difference of corresponding buckets, output tuples.

**Hash-Join**

To compute join of R(X,Y) and S(Y,Z):
Make chained hash tables for R and S using Y as the key.
Make all possible join tuples from tuples in corresponding buckets, output join tuples.

**Hybrid Hash-Join**

To compute join of R(X,Y) and S(Y,Z), assuming S is smaller than R (the reverse case is similar). Use Y as key.
Let k be a little less than $B(S)/M$. Construct chained hash table for S, but keep entire chain for bucket 0 in memory and only one block each for the other k-1 buckets as is usually done. When done, write the other k-1 buckets out to disk.
Construct chained hash table for R, but if tuple hashes into bucket 0, immediately form joins with tuples in bucket 0 for S (still in memory) and output the join tuples. Other tuples go into the other k-1 buckets for R (one block per bucket).

**Hybrid Hash-Join cont.**

After making chained hash table for R, write the other k-1 buckets of R to disc, and clear memory of bucket 0 for S.
Read into memory the chain of each of the remaining k-1 buckets of S. Read in the corresponding bucket of R one block at a time. Form all possible join tuples from the tuples in that block and the tuples in the chain from S. When bucket from R is finished, read in the chain for the next bucket of S and repeat.
Bucket 0 contains about $1/k$ of the blocks in R or S, so number of disc accesses is about $(3 - 2 \times M/B(S)) \times (B(S) + B(R))$.
(Bucket 0 isn't written to or read from disc.)
Selection

If relation R is not stored in order based on attribute a, but is stored in consecutive blocks, computing $\sigma_{a=v}(R)$ requires $B(R)$ disc accesses.

If relation R is stored in order based on attribute a, about $B(R)/V(R,a)$ disc accesses are needed.

If relation R is not stored in consecutive blocks, a dense index is needed, and about $T(R)/V(R,a)$ disc accesses are needed.

Join Using Indexes

To compute join of $R(X,Y)$ with $S(Y,Z)$:

Suppose that S is already indexed on Y.

If tuples of S are stored in order of Y on consecutive blocks, computing joins takes $T(R) \times (B(S)/V(S,Y))$ accesses to S, plus $B(R)$ or $T(R)$ accesses to R.

If tuples of S are not stored in order of Y, computing joins takes $T(R)/(T(S)/V(S,Y))$ accesses to S, plus $B(R)$ or $T(R)$ accesses to R.

R can be sorted first, then the matching tuples from S can be read only once for each distinct value of Y in R.

Zig-Zag Join

If both R and S are already indexed on Y and the indexes are dense, the indexes can be compared to find values of Y that are common to both indexes. Only tuples for those values of Y have to be read from disc.