Query Processing

- A lot happens between the time that you issue a *select* query and the arrival of that query's results: collectively known as *query processing*
- The overall steps are:
  - Parsing and translation
  - Optimization
  - Evaluation
- Hmmm…not unlike a typical programming language… (except maybe that last step)

Parsing

- No different from parsing and translation of programming languages
- For a relational database, the target representation is usually a relational algebra expression
- Semantic checks include:
  - Verification of relation and attribute names
  - Data type conversions
  - Expansion of views into their expression form

Relational Query Optimization

No single way to do things:

- Many ways to express the same query
- Many ways to translate into relational algebra
- Many ways to evaluate the relational algebra expression (e.g., order of operations)
- Many algorithms to execute an individual relational algebra operation

Each of these choices has a *cost* — the process of *query optimization* seeks to *minimize* this cost

Query Evaluation

- As optimal choices are made, the relational algebra expression in question is *annotated* with *evaluation primitives*, such as the algorithm or index to use
- The final set of evaluation primitives is called the *query-execution* or *query-evaluation plan*
- The query evaluation plan is executed by the *query-execution engine*, which returns the query's answers
- The whole process is internal to the database management system, so there is no "single best" way

Query Cost

- Many factors affect query cost:
  - Disk access
  - CPU time
  - Communication time
- It all adds up to the overall *query response time*
- Generally, the most significant factor is disk access: *block transfers* (data flow to/from disk) and *disk seeks* (finding data on the disk)

- If a disk has an average block transfer time of t_T and an average seek time of t_S, then reading a set of b blocks that requires S seeks would take:

$$bt_T + St_S \text{ seconds}$$

- Many databases run disk tests upon installation to determine t_T and t_S on their current server
- Block writes are usually twice as expensive as block reads, since they often do a read after the write in order to verify a successful write
- Cost estimation often errs on the conservative side, so actual time may be better than estimated
  - For example, a disk block may already be in buffer memory, but query processing always assumes that it isn't

Selection Operation

- Lowest-level algorithm: a *file scan*, which simply goes over an entire relation and picks out records that satisfy the current selection condition
  - *A1 (linear search)*: Grab all of the relation's blocks
    - Usually costs one seek and b_r block reads, assuming that the relation occupies b_r disk blocks; if blocks aren't contiguous, more seeks would be necessary
    - If selecting on a key attribute, we can stop on the first record, since there is only one
    - May be the slowest on average, but linear search is the most general — you can use it on any file, under any circumstances
  - *A2 (binary search)*: Search by block, requiring a seek and transfer of $\lceil log_2(b_r)\rceil(t_T + t_S)$ on average

- When available, we can perform an *index scan* — recall that a *primary index* matches the physical record order, while a *secondary index* does not
  - *A3 (primary index, equality on key)*: Use the primary index — for a B+-tree index of height h_i, cost would be $(h_i + 1)(t_R + t_S)$
  - *A4 (primary index, equality on nonkey)*: If the primary index is on a nonkey, meaning that multiple results may be returned, we incur additional cost proportional to the number of blocks containing matching records
  - *A5 (secondary index, equality)*: Same as A3 if indexed attribute is a key, but may be worse than linear search otherwise since we'll be reading *both* index node blocks *and* every block containing matching records

Selections with Comparisons

- Variations on linear, binary, and index search are used when the selection is of the form $\sigma_{A \leq v}(r)$
  - ◇ *A6 (primary index, comparison)*: For $A \geq v$, find the first record with v, then retrieve from there — note how linear scan is actually better for $A \leq v$, since we just scan from the start until we hit v !
  - ◇ *A7 (secondary index, comparison)*: Index nodes are sorted, but not the file blocks, so a seek has to be added for each block containing matching records — more costly than linear search for large result count

More Complex Selections

"Complex" means *conjunction* ($\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$), *disjunction* ($\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$), or *negation* ($\sigma_{\neg\theta}(r)$)

- *A8 (conjunctive selection using one index)*: Choose an individual θ_i and corresponding algorithm *A1* to *A7* with the lowest cost for $\sigma_{\theta_i}(r)$, then scan those results for records matching the other conditions
- *A9 (conjunctive selection using composite index)*: If a composite index is available for the attributes involved in the conditions and the comparison is for equality, then use that index directly

Complex Selections with Record Pointers/Identifiers

Additional algorithms are applicable if database records have internal unique identifiers or pointers

- *A10 (conjunctive selection by intersection of identifiers)*: Use individual indices to retrieve record identifiers that satisfy the corresponding conditions, then intersect before retrieving; for conditions without applicable indices, retrieve first then test
- *A11 (disjunctive selection by union of identifiers)*: Same as *A10*, except perform union; difference is, if *even one* condition has no index, then just go with linear scan

Sorting

- Sorting may be requested by the query (e.g., *order by*), or is an important preprocessing step for other queries, such as those that involve a *join*
- If records are completely in main memory, standard sorting algorithms such as quicksort apply
- Otherwise, some records are still on disk, resulting in what is called *external sorting*
- Common external sorting technique: *external sort-merge*, which cumulatively sorts multiple *runs* of the data based on amount that fits in memory at one time

Join Operation

- The next major query activity is the *join*, which is always involved the moment you use multiple relations in the query
- The first and simplest algorithm: *nested-loop join*
    - Nested iterations over the joined relations' tuples
    - In the innermost loop, test the current records; if they satisfy the join condition, add them to the result
    - Like linear scan, this is the most general algorithm, but is expensive on average

Block Nested-Loop Join

- Recall that the biggest cost in query processing is generally disk access — seek and transfer
- Instead of scanning by tuples, which may be scattered across disk blocks, we scan by *blocks*, then scan the tuples within each set of blocks
- The difference in iterating through blocks first is that, when going through just tuples, you read every block of the inner relation for every tuple of the outer one; with block nested-loop join, you read every block of the inner relation for every *block* of the outer one

More Nested Loop Tweaks

- If the join is equality with a key of the inner relation, then the inner loop can stop on the first match
- Modify block nested-loop so that instead of iterating by block in the outer loop, iterate by as many as will fit in memory (leaving enough for inner loop and results)
- Modify block nested-loop so that the inner loop alternates the scan direction — that way the block(s) used in the prior iteration can be re-used right away
- Use index on the inner relation if possible (see below)

Indexed Nested-Loop Join

- Since a join condition is essentially a select condition on the individual relations, an index can be used on the inner loop's relation if it is available
- Change in cost: instead of a linear scan for the inner "loop," you just get the cost of an index lookup (log for B+- or other tree, potentially less when hash is applicable to the join condition)
- This gives us some clear guidance on which relation to use for the outer loop (when there is a choice) — use the one with less tuples

Merge Join

- Also called the *sort-merge-join* algorithm, merge join is applicable to natural joins and equi-joins
- Prerequisite of the algorithm: tuples need to be sorted on the attributes over which they are being joined (i.e., if we are natural-joining $r(R)$ and $s(S)$, then merge join requires that r and s are sorted on $R \cap S$)
- Tuples may be unsorted if sorted indices are available for the joined attributes; in this case, the algorithm acts on the indices instead of directly on the data, resulting in additional disk block cost

- Overall sketch of the algorithm:
  - Gather up each tuple in s with the same values for $R \cap S$ (the text calls this set S_s)
  - Go through the tuples in r and place matching tuples in the cumulative join result
  - Move to the next value in $R \cap S$
  - Again, note that the sorting requirement ensures that we never have to re-read a tuple — we move linearly, once, through both R and S
- Final cost of the algorithm is the cost of the sort (order of log_M) plus the sort of reading every disk block of r and s once

Hash Join

- Another algorithm applicable to natural joins and equi-joins (good thing those are the most frequent ones)
- As you can tell from the name, this one involves a hash function *h*, specifically on the attributes being joined ($R \cap S$, as defined previously)
- *h* essentially *partitions* each relation according to the tuples with the same value for $h(t[R \cap S])$
- The main idea is that tuples from *r* and *s* that satisfy the join condition will also have the same hash value

- A second hash function is used to build an in-memory hash index *within* each partition
- Thus, the complete algorithm is:
  - ◇ Partition the tuples of *r* and *s* using *h*
  - ◇ For each partition of *s*, build a hash index, then iterate through the corresponding partition in *r* to locate matching tuples in *s* (called *probing*, where *s* serves as the *build input* and *r* is the *probe input*)
- The text covers various tweaks to the algorithm, particularly for when there isn't enough main memory to accommodate some element of the algorithm
- Cost is linear based on the number of blocks and the size of the hash function (i.e., number of buckets)

Complex Join Conditions

- As with selection, the specialized algorithms defined so far apply to a single equality comparison — otherwise, you'd have to use some kind of nested-loop join
- But some complex conditions can still take advantage of the faster approaches:
  - A join with a conjunction may have sub-conditions for which a faster join algorithm may be used; do those, then intersect the results
  - A join with a disjunction would do the same, but with a union instead of intersection

Eliminating Duplicates

- One way is to perform a sort: identical tuples end up being adjacent to each other, so a single linear scan will get rid of them easily
- Another way is through hashing: perform the same algorithm as hash join, but when building the in-partition hash index, throw out tuples that are already in their respective buckets
- Overall cost of duplicate elimination is relatively high, so SQL makes it explicit via *select distinct*; if you don't say *distinct*, duplicates are retained by default

Projection

- Projection is a matter of projecting each tuple
- For generalized projection, the per-tuple activity may involve additional calculations or function calls
- Since true relational projection removes duplicates, the final projected result will require duplicate elimination using the techniques described previously
- One optimization: a projection that includes a key is guaranteed to never have duplicates, so a quick check for this may eliminate the duplicate elimination step

Set Operations

- For all three set operations — union, intersection, and difference — sorting is again the key
- Sort the relations, then:
  - Accumulate them all for union, removing duplicates
  - Retain only common tuples for intersection (much like merge join but on all attributes)
  - Retain the *un*common tuples for difference
- In all cases, cost is sorting + one-time disk access
- An alternative approach based on hashing is also described in the text

Outer Join

- Recall that an outer join does *not* throw out unmatched tuples on the "outer" side — those still go to the result, with *null*s filling out the unmatched fields
- Two strategies for this:
  - Calculate the inner join first, saving the intermediate result as q_1, then perform $r - q_1$ where r is the relation on the "outer" side
  - Or, internally modify the join algorithms to support outer joins directly — for example, with merge join, also save non-matching "outer" tuples

Aggregation

- Recall that aggregation takes groups of tuples then performs a calculation over each group, such as sum, count, or average
- Strategy is similar to duplicate elimination: *sort-n-scan*, but instead of eliminating duplicates as you go, build the individual groups, then perform the aggregate function once the groups are formed
- Instead of building the *entire* group, you can calculate the aggregates as you go: reserve a tuple for each group, then modify the aggregate value while scanning

Expression Evaluation

- The algorithms shown so far pertain to *individual* relational algebra operations
- In reality, a typical query combines them — for example, *select customer_name from account natural inner join customer where balance* < 2500 consists of a selection, a natural join, and a projection
- Two general approaches for this: *materialization* and *pipelining* — as with the other algorithms, one is general but expensive, while the other is more efficient but doesn't apply to all cases

Materialization

- *Materialized evaluation* walks the parse or expression tree of the relational algebra operation, and performs the innermost or leaf-level operations first
- The intermediate result of each operation is *materialized* — an actual, but temporary, relation — and becomes input for subsequent operations
- The cost of materialization is the sum of the individual operations *plus* the cost of writing the intermediate results to disk — a function of the *blocking factor* (number of records per block) of the temporaries

Pipelining

- The problem with materialization is just that — lots of temporary files, lots of I/O
- With *pipelined evaluation*, operations form a queue, and results are passed from one operation to another *as they are calculated*, hence the technique's name
- Avoids write-outs of entire intermediate relations
- General approach: restructure the individual operation algorithms so that they take *streams* of tuples as both input and output

- Two "styles" of pipelines:
  - *Demand-driven* pipelines have operations at the top (or nearer the end, depending on your point of view) of the pipeline to request new tuples to process — can be modeled as an *iterator*, with *open/next/close* calls
  - *Producer-driven* pipelines have each operation churn out tuples "eagerly" — for as long as there is input, they produce output, sending them to a buffer; the succeeding operation drains that buffer as quickly as it can, putting *its* output in another buffer
- Related terms: demand-driven can be thought of as *pulling* from the top, and producer-driven can be viewed as *pushing* from the bottom; where producer-driven is "eager," demand-driven is "lazy"

Pipelining Implementation Limitations

- The main trick with pipelining is that the entire input — an intermediate relation — is not available to an operation (otherwise it would be the same as materialized evaluation)
- So for instance, algorithms that require sorting can only use pipelining if the input is already sorted beforehand, since sorting by nature cannot be performed until all tuples to be sorted are known
- Note how the most general (but expensive) algorithms (linear scan, nested-loop) can be pipelined easily

Pipeline Cost Considerations

- Ultimately, the query planner needs to choose between the cost of pipelining for the general algorithms (since those are the only algorithms that can *always* be pipelined) and the cost of materialization for the more efficient (but specialized) ones
- General reasoning:
    - ◇ Can pipelining *and* a specialized algorithm be used? (may depend on whether tuples are already sorted coming in)
    - ◇ If not, how does pipeline + generic compare against materialized + specialized? (typically, larger relations favor materialization)