COS221 - L28 - Query processing and optimisation (Part 1)

Linda Marshall

12 May 2023

High-level query processing

- Queries specified in a high-level language such as SQL must first be scanned, parsed and validated.
  - The scanner identifies the query tokens.
  - The parser checks the query syntax.
  - The validation step checks if relations and attributes are valid.
- An internal representation of the query is created using a query tree.
- The DBMS then chooses a query strategy for executing the query - known as query optimisation.
- This strategy may not be the most optimal, but reasonably efficient.
- Finding the optimal strategy is usually too time consuming.

High-level query processing

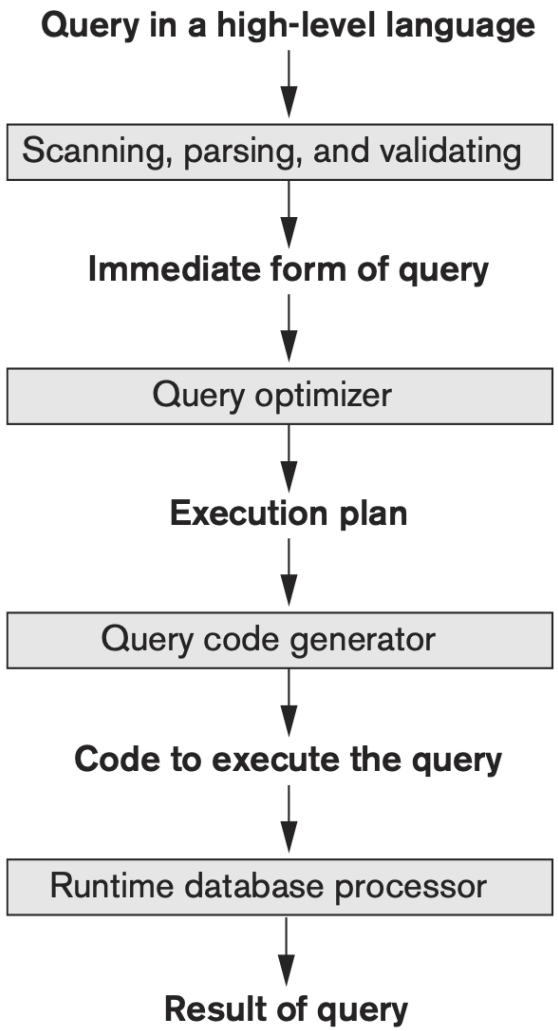

Figure 18.1
Typical steps when processing a high-level query.

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

High-level query processing

- Most high-level query languages specify what the intended results of the query are and not how it should be obtained.
- Two dominant techniques are used for query optimisation:
  - heuristic rules
  - systematic estimation of different execution strategies
- Most query optimisers combine these techniques.

Translating SQL queries into RA

- SQL queries are the dominant queries specified in RDBMSs. These queries are translated into an equivalent RA expression. This expression is represented in a query tree and then optimised.
- SQL queries are typically decomposed into query blocks. Each query block contains a single SELECT-FROM-WHERE expression - with its GROUP BY and HAVING clauses if it has one.
- It is more difficult to optimise *nested correlated queries* where the tuple variable of the outer block appears in the WHERE-clause of the inner block.

Translating SQL queries into RA

- Consider the example of a nested query without correlation with the outer query:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > (SELECT Max(Salary)
                FROM EMPLOYEE
                WHERE Dno=5);
```

- This query is composed into two blocks. The result of the second block feeds into the first block.
- Each block is translated to extended RA expressions.
  - Second block - $\mathcal{F}_{\text{MAX Salary}}(\sigma_{Dno=5}(EMPLOYEE))$
  - First block - $\pi_{Lname,Fname}(\sigma_{Salary>c}(EMPLOYEE))$ where c is the result of the second block

Algorithms for external sorting

- Sorting is one of the primary algorithms used in query processing.
  - It is used whenever ORDER BY-clause is specified and is key to JOINing, UNION, INTERSECTION, duplicate elimination on PROJECT (DISTINCT option in SQL).
- Sorting is avoided if either a primary or clustering index exists for the file attribute.

Algorithms for external sorting

- Typical external sorting algorithms use the **sort-merge** strategy because all records on disk do not fit into memory.
  - Sorting takes place by sorting subfiles - called runs.
  - Sorted subfiles are written back to disk for merging.
  - The actual sorting and merging takes place in buffer space in main memory - part of the DBMS cache.
  - Each buffer is the same size as a disk block.
- **sorting phase** - portions of the file (runs) to be sorted that fit into the buffer are loaded and sorted. Sorted runs are written back to a temporary file.
- **merging phase** - sorted runs are merged during one or more merge passes. The degree of merging is the number of sorted runs that can be merged in one step.

Algorithms for external sorting

```
set     i ← 1;
        j ← b;              {size of the file in blocks}
        k ← nB;             {size of buffer in blocks}
        m ← ⌈(j/k)⌉;        {number of subfiles- each fits in buffer}
{Sorting Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
           remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i ← i + 1;
}
{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log(k−1) m⌉ {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈j/(k−1)⌉;      {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
           read next k−1 subfiles or remaining subfiles (from previous pass)
              one block at a time;
           merge and write as new subfile one block at a time;
           n ← n + 1;
        }
        j ←     q;
        i ← i + 1;
}
```

Figure 18.2
Outline of the sort-merge algorithm for external sorting.

Algorithms for external sorting

- The performance of the soft-merge algorithm can be measured by the number of disk block read and writes it takes to sort the entire file. That is the sum of the reads and writes for the sorting phase and merging phase.
- The formula for the approximate cost is given by: $(2 * b) + (2 * b * (log_{d_M} n_R))$ where
  - b is the number of file blocks
  - d_M is the *degree of merging*
  - n_R is the number of initial runs

  - $log_{d_M} n_R$ is the number of merge passes
  - $(2 * b * (log_{d_M} n_R))$ is the merging phase
  - $(2 * b)$ is the sorting phase

Algorithms for SELECT and JOIN operations

SELECT

- SELECT operations are basically a search operation which complies to a certain condition.
- The following search operations can be used to implement the SELECT.
  - (S1) to (S6) are used on single attribute conditions - or *simple* condition
  - What if the condition is a *conjunctive*? That is, made up of several simple conditions connected with an AND? (S7) to (S9) make provision for conjunctive conditions.
  - What about *disjunctive* selection conditions? Compared to conjunctive, disjunctive are difficult to optimise. In many cases, the resultant records satisfying the condition are the union of the records which satisfy the simple conditions. The disjunction can only be optimised if an access path exists for each simple condition. If one simple condition does not have an access path defined, approach S1 needs to be followed.

Algorithms for SELECT and JOIN operations

SELECT: Simple conditions

- (S1) - **Linear search** (also referred to as brute-force search): retrieves every record and tests if it satisfies the search condition.
- (S2) - **Binary search**: requires the selection condition to involve an equality comparison on a key attribute used for ordering the file.
- (S3a) - **Using a primary index**: applied to an equality comparison of a key attribute with a primary index. Retrieves one record at most.
- (S3b) - **Using a hash key**: applied to an equality comparison on a key attribute with a hash key. Retrieves a single record at most.

Algorithms for SELECT and JOIN operations

SELECT: Simple conditions - continued

- (S4) - **Using a primary index to retrieve multiple records**: used if the comparison condition is $<$, $>$, $<=$ or $>=$ on a key field with a primary index. Find the equality record and proceed from there based on the comparison condition.
- (S5) - **Using a clustering index to retrieve multiple records**: used when the selection involves an equality comparison on a nonkey attribute with a clustering index.
- (S6) - **Using a secondary (B^+-tree) index on an equality comparison**: retrieves a single record if the indexing field is a key and multiple records if it is not a key. This also works for $>$, $<$, $>=$ and $<=$.

Algorithms for SELECT and JOIN operations

SELECT: Simple conditions - discussion

- S1 can be applied to any file, where the others depend on the access path of the selection condition
- S2 requires the file to be sorted on the search attribute
- S3a, S4, S5 and S6 make use of indexes and are referred to as *index searches*. The appropriate index must exist on the search attribute.
- S4 and S6 can be used to retrieve values in a range and are called *range queries*.

Algorithms for SELECT and JOIN operations

SELECT: Simple conditions - summary

When a single condition specifies the selection, the DBMS checks if an access path on the attribute exists. If an access path exists, the method corresponding to the access path (such index, hash key or sorted file) is used, otherwise a linear search is used.

Algorithms for SELECT and JOIN operations

SELECT: Conjunctive conditions

- (S7a) - **Using a bitmap index**: Apply the `OR` operation to a set of bitmaps to determine the set of record id's that apply.
- (S7b) - **Using a functional index**: Create a functional index to retrieve values that qualify. An example of a functional index is given by:

```
CREATE INDEX income_ix
ON EMPLOYEE (Salary + (Salary*Commission_pct));
```

- (S8) - **Conjunctive selection using an individual index**: used if the attribute involved in the simple condition of the conjunctive has an access path using methods S2 to S6. Use the condition to retrieve the records and then see which of the retrieved records satisfy the remaining simple conditions of the conjunctive condition.

Algorithms for SELECT and JOIN operations

SELECT: Conjunctive conditions - continued

- (S9) - **Conjunctive selection using a composite index**: when 2 or more attributes are involved in an equality condition in the conjunctive and a composite index exists on the combined fields, the index is used directly.
- (S10) - **Conjunctive selection by intersection of record pointers**: used if secondary indexes are available on more than one of the fields involved in the simple conditions of the conjunct and the index include records pointer rather than block pointers. The intersection of the resulting record pointers gives the set of record pointers which satisfy the conjunctive condition. Each record is then further tested to see if the remaining conditions are satisfied.

Algorithms for SELECT and JOIN operations

SELECT

- Query optimisation is mostly required for conjunctive conditions. The optimiser should try to retrieve the fewest records in the most efficient way.
- The DBMS must determine the **selectivity of a condition**.
  - Selectivity is defined as the ratio of the number of records that satisfy the condition to the total number of records in the file.
  - A ratio of 0 means there are no records that satisfy the condition and a ratio of 1 means all records satisfy the condition.
- Exact selectives of all conditions are not available, the DBMS does keep in its catalog estimates of selectives.

Algorithms for SELECT and JOIN operations

JOIN

- Join operations are the most time-consuming operations.
- Only the EQUIJOIN and NATURAL JOIN will be considered.
- There are numerous ways to implement a *two-way* join (a join between 2 files). As the number of files involved in a join increases, so does the number of ways the join can be done increase.
- For the discussion below, only join operations of the form $R \bowtie_{A=B} S$ will be considered.

Algorithms for SELECT and JOIN operations

JOIN

- (J1) **Nested-loop join** (or nested-block join) - The default brute-force algorithm. For each record r in R, retrieve every record s in S and test if r[A]=s[B]. Naturally, this is done on blocks on disk.
- (J2) **Index-based nested-loop join** (using an access structure to retrieve the matching records) - If an index exists for one of the join attributes (say B of S), retrieve each record in R (call it t) and then use the access structure (index or hash key on S) to retrieve the matching record s from S where s[B] = t[A]

Algorithms for SELECT and JOIN operations

JOIN - continued

- (J3) **Sort-merge join** - If the records of R and S are ordered on the join attributes A and B respectively, scan both files concurrently in order of the join attributes matching the records that have the same values for A and B. If the files are not sorted, they may be sorted using external sorting. If these are not key attributes, use a secondary index and sort the index. This method is efficient for previously sorted files for the join attributes and very inefficient for nonkey join attributes on unsorted files.

Algorithms for SELECT and JOIN operations

JOIN - continued

- (J4) **Partition-hash join** - Partition the records of R and S into smaller files using the same hash function on the join attributes of each file. This process has two phases, the partitioning phase where the smaller file is hashed and the values with the same hash value are partitioned into hash buckets. The second phase, the probing phase, iterates through the other file and hashes the attribute using the same hash function to determine the bucket from the first file which is then probed.

Algorithms for SELECT and JOIN operations

JOIN - discussion

- J1 to J4 are implemented using disk blocks rather than individual records.
- The file chosen for the outer loop of J1 impacts the performance of the join. The file with the least disk blocks makes for a more efficient algorithm.
- For J2, the *join selection factor* (fraction of records in one file to be joined with records on the other file) on an equijoin, impacts performance. A join selection factor closer to one is more efficient than closer to 0.
- J3 is most efficient when both files are already sorted on the join attributes.
- J4 is quite efficient for most cases. Both files only require 1 pass. A modified J4 - *Hybrid Hash-Join* - improves the efficiency of the algorithm. It includes a joining phase for one of the partitions in the partitioning phase.

Algorithms for PROJECT and Set operations

- The project operation, $\pi_{\text{attribute\_list}} R$, is straightforward to implement if the attribute list includes a key of R because the result will have the same number of tuples as R.
- If the attribute list does not include a key, duplicate tuples must be eliminated. This is done by:
  - sorting the result and removing the duplicates appearing consecutively after sorting; or
  - hashing the record and checking for a duplicate in the hash bucket. If it is a duplicate, it is not inserted.

Algorithms for PROJECT and Set operations

Set operations, UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT, are expensive to implement.

- CARTESIAN PRODUCT is notoriously expensive and is often substituted with other operations during query optimisation.
- UNION, INTERSECTION and SET DIFFERENCE (EXCEPT in SQL) apply to union-compatible relations.
  - That is relations with the same number of attributes and corresponding attributes from the same domain.
  - These operations are implemented with variations of the sort-merge technique.
  - Both files are sorted on the same attribute.
  - The sorted files are iterated through concurrently and for:
    - INTERSECTION - include records which appear in both files in the merged result.
    - UNION - records from both files are included in the merged result. When a record is found in both files, one is merged into the result.
    - SET DIFFERENCE - include records in the first file which are not in the second in the merged result.

Algorithms for PROJECT and Set operations

Hashing can also be used for UNION, INTERSECTION and SET DIFFERENCE.

- One file is scanned and hashed into hash buckets.
- The other file is then scanned and each record hashed to probe the appropriate partition.
- The probing is different for each operation:
  - INTERSECTION - add the record to the result file if an identical record is found in the hash bucket
  - UNION - only insert non-duplicate records from the second file into the first file buckets.
  - SET DIFFERENCE - remove a record from the hash bucket if an identical record is found in the bucket.

Note: SQL UNION, INTERSECTION and EXCEPT do not include duplicates in the result. UNION ALL, INTERSECTION ALL and EXCEPT ALL do.

Aggregate operations and OUTER JOINS

- Aggregates can be implemented by scanning the file or using an appropriate index if available.
- A B^+-tree index, if it exists, is useful for MIN and MAX operations.
- AVERAGE and SUM can only be applied when a dense index is available on the attribute.
- For non-dense indexes, the calculation is only correct if the total number of records associated with the index has been stored. If this is the case, operations such as COUNT an be done on the index rather than the actual file.
- When GROUPED BY is used, the operation must be applied to each group as partitioned by the grouping attribute. Either sorting or hashing can be applied on the grouping attributes. If a clustering index exists, then the records are already partitioned and the computation needs only to be applied to each group.

Aggregate operations and OUTER JOINS

- OUTER JOINS are implemented by modifying one of the JOIN algorithms
- LEFT OUTER JOIN - the left relation is used for the outer/single loop because every record in the left file must appear in the result. If there is a matching record in the right file, the records are joined. If not, the right 'values' are padded with NULL.

Aggregate operations and OUTER JOINS

- An OUTER JOIN can be written as a combination of relational algebra operations. For example:

```
SELECT Lname, Fname, Dname
FROM (EMPLOYEE
      LEFT OUTER JOIN DEPARTMENT ON Dno=DNumber);
```

Can be written as:

1. Compute the inner join of the two relations
2. Find EMPLOYEE tuples that do not appear in the inner join result
3. Pad the tuples of step 2 with NULL
4. Find the UNION of the results of steps 1 and 3.

Aggregate operations and OUTER JOINS

1. Compute the inner join of the two relations
   $TEMP1 \leftarrow \pi_{Lname,Fname,Dname}(EMPLOYEE \bowtie_{DNo=Dnumber} DEPARTMENT)$
2. Find EMPLOYEE tuples that do not appear in the inner join result
   $TEMP2 \leftarrow$
   $\pi_{Lname,Fname}(EMPLOYEE) - \pi_{Lname,Fname}(TEMP1)$
3. Pad the tuples of step 2 with NULL
   $TEMP2 \leftarrow TEMP2 \times NULL$
4. Find the UNION of the results of steps 1 and 3.
   $RESULT \leftarrow TEMP1 \cup TEMP2$

Combining operations using pipelining

- An SQL query is translated into a relational algebra expression - a sequence of relational operations. Executing a single operation at a time requires temporary results to be stored on disk - this is time consuming as the result may be required immediately by the next operation.
- In many cases, combining operations to limit temporary results on the disk improves performance.
- Heuristic relational algebra optimisation is used to determine which operations can be grouped together for execution. This is referred to as pipelining or stream-based processing.

What comes next?... Optimisation

- Using heuristics in query optimisation
- RA transformation rules
- Using selectivity and cost estimation for optimisation
- Semantic query optimisation