Data 101: Data Engineering Note 4 – September 22, 2022

Index Selection, Query Processing, and Optimization

Lecturer: Lakshya Jain *Scribe: Audrey Cheng*

1 Introduction

In this note, we'll cover index selection, query processing, and query optimization.

2 The Declarative Contract

So far, we have been imagining a declarative contract between the data system and the user, where the user specifies the relations, along with any constraints but details about how this data is managed or stored. The data system is figure out how to store data, process it, enforce constraints, etc. and do so in the "best" way it can. There are many reasons for which we would want to cede control to the system.

- Simpler for end-users to not need about details
- Hard for (most) users to reason about how to best execute a query
- System has more fine-grained awareness of the data

However, there are also reasons that users *should* worry about performance.

1. **Revisiting the specification:** for cases when even the "best" execution strategy is slow, users can rephrase the specification in less expensive ways
2. **Contract breakdown:** there are scenarios where data systems struggle to find good ways to execute queries and user have a better idea of data characteristics, so users can steer the system towards "better" execution strategies.
3. **Many heavyweight levers still under the control of users:** Materialized views, certain efficient data access structures, amount of resource employed, etc. can have significant impact on performance and are under user control.

3 A Mental Model for a Data System

The previous section shows that users can and should take actions improve performance. To do so, we need to first build a mental model of the data system.

Data is laid out on blocks/pages on stable storage (SSD or HDD), and each block may have 100s of tuples. Data is retrieved a block at a time for reading/writing by the data system. This data is then stored in frames in a main-memory buffer and written back to stable storage as needed. Usually, computation on data in main-memory is cheap relative to I/O, so the main cost in processing SQL queries is the cost of I/O, i.e., fetching the blocks into the buffer and writing them out. Also, reading/writing data blocks sequentially is typically cheaper than random reads and writes. This brings us to our first rule of what impacts performance:

4 Indexes

An important data structure for performance is the **index.** Indexes allow a user to perform data matching certain characteristics. They are essentially data structures storing "index key-location" pairs.

Indexes can be stored in a variety of formats. The most popular type of index in data systems are B+ trees, which are essentially applying and extending the binary search tree ideas. Unlike binary search trees, B+ trees have a high fanout. B+ trees are self-balancing and maintain certain fill-factor (usually half). They have logarithmic complexity for all operations.

One other prominent type of indexes are hash indexes, which are hash-based dictionaries. These have constant complexity but can only handle value lookups (not range lookups).

Indexes are especially useful for the following cases:

- When your relations are large
- When sequential scans are too time-consuming
- When joins are present

To declare an index, use the following syntax:

```
CREATE INDEX ageIndex ON Stops (age)
```

Similarly, to drop an index:

```
DROP INDEX ageIndex;
```

Indexes may seem magical, but there can be high the maintenance costs during updates, deletes, inserts. If an index is not maintained, it may lead to inaccurate query results. So, which indexes should we create? The answer to this question depends!

Many databases automatically create indexes for PRIMARY KEY or UNIQUE attributes. This helps enforce constraints and many queries tend to be based on these attributes.

One important consideration for indexes is **cardinality**, or the number of distinct values for a particular attribute. High cardinality attributes are great to index on because there are higher potential savings compared to having to search through many values. On the other hand, low cardinality attributes are not great for indexes. The exception is if data is clustered on that attribute; in this case, we can read many fewer blocks.

Indexes are also good on attributes used often in WHERE clauses, key attributes, and attributes that are "clustered". Indexes are not very good on attributes where there are many tuples per value of attribute and tables that are modified more often than queried.

5 Query Execution Plan

How are queries actually executed by the data system? The query execution plan lays out how to proceed. We can think of plans at two levels:

- The logical level: the logical operators describe "what" is done
- The physical level: the operator implementations describe "how" to do it

In code-centric data systems, the user can supply both the logical query plan,] and parts of the physical implementation or leave the physical implementation to the system.

Now, we'll cover different physical implementation of logical operators. A simple table scan would read all blocks on disk one-by-one. An index scan would first read the index and then use it to read relevant blocks.

On the other hand, joins (theta or natural) are expensive. At a high level, these operations match information across multiple relations, and there are many different way to do them. We'll cover three using example tables R and S.

Nested Loop Joins. The nested loop join is straightforward: for every tuple of R and S, check if it matches. If it does, add it to the output! There are variants of this method, including index-nested loop which uses an index in the "inner" loop to look up only blocks of S that can match the k blocks of R.

Sort-Merge Joins. This method involves two phases. The first phase is the sort phase: sort portions of R on the join attribute, and write out the sorted runs of blocks. Do the same thing for S. The second phase is the merge phase: merge and match tuples across the runs from the first phase by walking down the runs in sorted order. There are also variants that can use indexes and take advantage of relations that are already sorted.

Hash Joins. The hash join also involves two phases. In the first phase, both R and S should be hashed into buckets based on the join attribute. Then in the second phase, all the tuples hashed into each bucket should be read from both R and S at a time and the join should be performed. One variant is based on whether one of the relations can fit entirely in memory.

For our remaining operators, filters and projects can simply be "applied" to the results of an operation, so there are no specific physical operator choices. The exception is index-scan for which filtering done as part of scanning. For aggregation and grouping, hashing and sorting are key techniques, If an index is present, it can be used to retrieve in sorted order.

Overall, there are many variations in the physical design of operators, but most rely on a small set of techniques, including sorting, hashing, indexing, and nested-looping. While we've covered these techniques, we haven't talked about what might work best in a given setting. To do so, we need a cost model to take into account the sizes of the relations, the distributions of values, and buffer sizes.

5.1 Converting a Query Plan

We'll now cover how we covert a logical query plan into a physical one. There are four steps.

Step 1. First, convert the SQL query to a logical query plan. Sometimes, complex SQL queries get decomposed into multiple query plans (e.g., CTEs or subqueries).

Step 2. Apply rewriting to find other equivalent logical plans. To do so, we rewriting rules, or algebraic laws that allow us to manipulate relational algebra expressions, such as commutative, associative, and distributive laws. For selection, a special rule a called predicate pushdown is important to know. This rule states that the predicate can be applied before / after select without impacting the final result. The performance implication of this rule is that the earlier we process selections, the more we reduce data "at source", and the less we need to manipulate later for more exp operations (e.g. joins). There is also a similar rule projections called project pushdown.

Step 3. Use cost estimates pick among the logical plans and the corresponding physical plan. The query optimizer is responsible for this step. Cost estimates is often inaccurate, since it is done based on coarse-grained statistics. There are some heuristic rules that restrict the search space (e.g., predicate and projection pushdown). Joins are often the hardest part in the process since they are the only operators that "multiply". There are many different join orders and trees, leading to a very large search space. Furthermore, there are a lot of bad join orders that can lead to giant intermediate relations. A variety of approaches have been developed for optimization. One notable one is the "Selinger" algorithm, which only considers left deep trees. The top-down approach, exemplified by the Cascade query optimizer, is also a popular choice.

Step 4. Feed the corresponding physical plan to the query processor. Query optimization gives us a physical query plan that consists of a sequence or workflow of physical operators and specifies whether intermediate results are "pipelined" or materialized. Pipelining is an approach where tuples can "flow up" from the bottom operators to the top as soon as they are ready, allowing operators to do work in parallel.