Building a composable Database Management System

Elias Strauss
elias.strauss@campus.tu-berlin.de
TU Berlin
Berlin, Germany

Jonas Frey
jonasfrey@win.tu-berlin.de
TU Berlin
Berlin, Germany

Mateusz Musiał
mateusz.a.musial@tuberlin
TU Berlin
Berlin, Germany

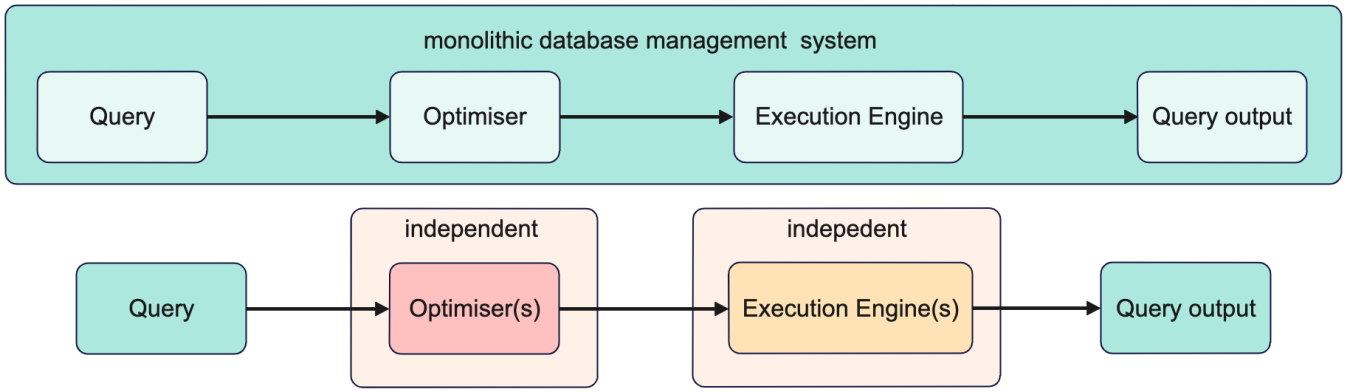

Figure 1. Splitting the monolith structure of a traditional DBMS

Abstract

When it comes to data in computer science, very few things are as ubiquitous as database management systems. Since a database management system (DBMS) allows for an easy and ready made solution to store data in any way imaginable, they are the go to software when someone is in need of a data storage system. Currently the major DBMS's are not trying to be more than a ready made solution. It has become standard procedure to keep the inner workings of a DBMS so tightly coupled, that it has become increasingly difficult to understand if all the individual components of a DBMS are the right tool for the job. To address this challenge we have build a novel system that allows users to operate a selection of those components interchangeably. This approach shows a new level of understanding on how individual DBMS components interact with each other and clearly demonstrates the performance and implementation differences of each component.

Artifact Availability: The source code has been made available at https://github.com/Elias-Strauss/dbpro

Keywords: DBMS, optimiser, execution engine, composable components

DBPRO, July 18th, 2022, Berlin, Germany

1 Introduction

Traditional DBMSes are monolithic software artifacts that tightly couple the work of the query optimisers together with the query execution engine. This creates a large black-box that is responsible for the entire query processing pipeline without the possibility of customisation. This greatly limits the growth opportunities for legacy systems where only part of the black box is required as well as the knowledge about the potential speedup arising from breaking the monolithic architecture remains uncharted territory. The goal of this project is to allow broad cross-comparison of the performance of the various combinations of query optimisers with query execution engines.

Beginning with the SQL input into the DBMS - a parser is responsible for transforming user queries into abstract syntax trees (logical plans). A query optimiser is a DBMS component that analyzes SQL queries and is responsible for the efficiency of the execution. A query optimiser generates a number of logical plans, each capable of running the query, and then chooses a one "that is good enough". An execution engine performs operations by fetching data from the database. It does that by taking the logical plan and then tranforming it to a physical pla, that is executable. The problems arise from the compatibility between the query optimiser output and the execution engine input. The logical plan corresponds roughly to the relational algebra, whereas the execution engine requires direct calls of its functions that perform concrete operations in relation to the given data source. The implementation of these two concepts has by no means a single solution and this gives rise to a challenge -

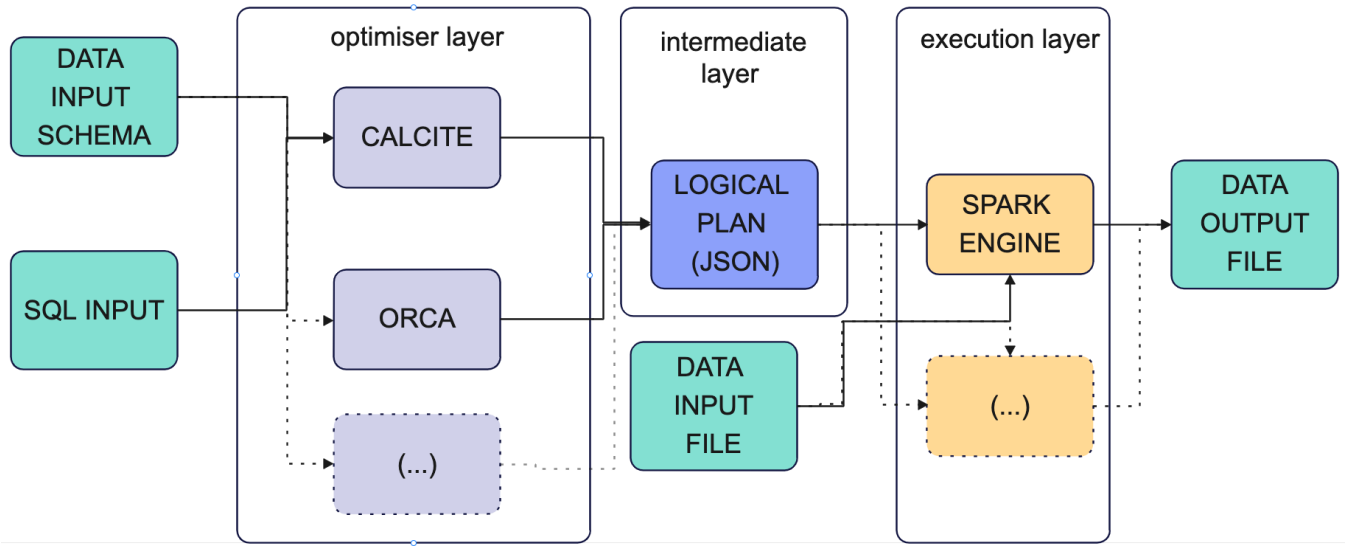

Figure 2. System overview

connect multiple, often not meant for each other, optimisers with execution engines despite different implementations, whereas a monolithic architecture only needs to worry about a 1 to 1 compatibility.

2 Problem Definition

Therefore the problem can be defined as follows:

- Agree on the intermediary format that is general enough so that its required arguments are general enough so that any execution engine can run on them, yet specific enough so that all/most of the optimizers provide enough information to create the JSON
- Extract the information (for each operation, e.g. which column ids should the join operation be performed on) from the query optimiser output and save it to the intermediate representation
- Be able to use any JSON that we create to run the execution engine and create a final output
- Collect information about the query execution speed and compare it to the other results

3 System Overview

So far we have talked about the goal of the project and how it should look in its end-stage. Now we will begin our journey over the implemented prototype that strives to fulfill that goal, for now it is however limited in its capabilities - it only runs a single tpc-h query and the pipeline is not complete. The state should be better understandable once we are done explaining each component. In this section we will introduce each component and give general information about its implementation, whereas the exact details of the implementation will be explained in Section 4.

The prototype in the broader sense consists of 4 parts - the Calcite part, the Orca part, the Spark part and the introduced JSON-middleman. Calcite and Orca are our optimisers whereas Spark is our execution engine. The calcite part can be further broken down into the SQL-parser and Logical-plan-parser. The Orca part consists only of the SQL-parser and the Spark part is the aforementioned calling of the low-level Spark RDD functions. Let us begin with the Calcite part.

3.1 Calcite (Optimiser Layer)

The goal of this part of the solution was: given a SQL query input return a parsed logical plan in the defined intermediary format. Additionally, in order to be able to optimise anything, Calcite (or any optimiser basically) requires at least the schema for the query. Complex optimisations require statistics about the data to be provided. This allows for example join order optimisations. With the schema and statistics Calcite is then able to validate and optimise SQL queries. The simplest solution for Calcite to acquire a schema is to create a connection to a DBMS such as Postgres. Our prototype allows the creation of any schema to which then rules for rule-based optimisation can be added.

Having that, we can input the SQL-query into the prototype that parses the query and optimises it using Calcite methods. It outputs a logical plan which is in the form of a tree which consists of multiple nodes, each being a relational algebra operation. Thanks to the Calcite API this output is a Java object and it is possible to automate the extraction of its arguments. These need to be in turn saved to the Intermediate representation.

3.2 Orca (Optimiser Layer)

Orca is an optimiser developed from Greenplum and is developed for Greenplum's database GPDB. Orca is very tightly coupled to its database, this means that it has no API and needs a proper wrapper to be a standalone optimiser. Regardless we have managed to understand a vital part of its output and created a parser that processes a logical plan for a tpc-h query. What lacks however is the way to input a SQL-query into the Orca, the logical plans for the tpc-h query are pre-available in the Orca project. Once parsed, the output logical plan is converted into the inetrmediate representation, analogically to the Calcite plan.

3.3 Intermediate Layer

The motivation behind this JSON intermediary is to have one format, universal for all optimisers outputs and for all execution engines inputs. It is a chain of operations that an execution engine has to run, containing all the variables needed for the proper execution of the specified operation. This format has been created by comparing the logical plans of Calcite and Orca with the requirements of Spark RDD therefore it might not be universal for any component insofar as it has not been tested with any other component. In such cases it might have to be adapted. The exact specification of this component will be provided in Section 4.

3.4 Spark (Execution Layer)

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. From our perspective Spark is a fast execution engine that can be accessed low-level by creating RDDs and interacting with the RDD functions. On the other hand, it is also possible to interact with the high-level Spark-SQL that takes a SQL-query and executes it using its own optimiser, Catalyst as well as, of course, the RDD. This is used in our project as a benchmark since Spark in its default form (SQL) is widely used. The execution engine part is the first part of the project where the actual data source (database) is required. In the prototype which we adapt for the tpc-h queries it is important that we use their data source. This is being done by running a generator that outputs a database of desired size (be it 1GB or 1TB). The output is in a .tbl format, which is basically a .csv that is separated by '|' instead of ','. Once done with processing all the operations from the JSON file, the final output RDD can be returned as such or saved as, for example, a .csv file.

4 Solution details

4.1 Data sources

Our data consists of files in .tbl format. Currently this is the only format we support as Spark reads files in that specific format.

For the files we use the TPC-H dataset. The TPC-H dataset provides a selection of functions that we used. It provides a data generator that can create its dataset in any size. It also provides a selection of 22 queries that are build around benchmarking thus providing a great baseline for comparisons. In addition the queries can be build in different SQL variants with an included query generator. We are using the standard SQL syntax for our project. TPC-H also provides the output of the queries as a known quantity allowing validation on whether a query output is correct.

4.2 SQL to Calcite

Transforming SQL to Calcite in our project consists of 2 primary stages. Defining the schema of the data that our query runs against and transforming said query into a valid logical plan.
Each of our components is as independent as possible to allow for maximum flexibility when adding new components. This means that for Calcite we did not use its already included systems to connect it to other components. First for the schema we are using our own class for the tables and statistics. This way the schema can be build for any dataset regardless of its data. In our implementation Calcite never looks at the actual data. This has the benefit of also allowing any statistics, which are a customisable component in our implementation. We are only utilising a row-count statistic but any type could be implementet to ensure ideal compatibility with any execution engine.
For further flexibility Calcite uses a customisable understanding of the SQL syntax. This allows Calcite to work with any query. For example in some SQL variants you use GETDATE() to retrieve the current date and time while in others you use NOW(), but with our implementation both variations can be validated assuming the customisable SQL syntax Calcite uses has both SQL operators included.
To produce the logical plan, calcite provides that plan in the RelNode data format, we have to assign an optimiser to Calcite. There are 2 optimisers Calcite provides: VolcanoPlanner and HepPlanner. The VolcanoPlanner is a cost based optimiser while the HepPlanner is a simpler rule based optimiser. We have implemented a HepPlanner with a customisable ruleset that allows a certain selection of logical operators in the logical plan. This allows us to ensure our parser from Calcites logical plan to the intermediary format does not run into an operation it does not know.

4.3 Intermediate format

The intermediary is the universal connector between our optimisers and our execution engines, that allows our components to be independent from each other.
The data structure is in JSON format, that means we are using key-value-pairs. We decided to use JSON because it's schemaless and this allows our structure to be flexible and to fit any query within our supported operators.

The intermediary is basically a chain of logical operators, that the execution engine can run without the need to check the ordering of the operations. This is achieved by the optimiser parsers through a topological order of the logical operator tree (the output of the optimisers).

{ "id" : Int, "type" : String, "id_parent" : Int, ... }

Figure 3. beginning of each operator in intermediary

Each logical operation has the same basic structure, it consists of an ID of the current statement and the type of the operation. Almost every operation takes as an input the output of another operation that's why most statements also have their predecessor ID.
In Listing 1, the instantiation of our intermediate representation for a projection, you can see the advantages of the schemaless, self-describing key-value structure, because every operation has the same basic structure but depending on the type of the structure we can store additional, necessary information about the operation as well. Additionally, we are able to represent nested structures as well.

```
{"id": "3", "type": "Projection",
"id_parent": "4",
"colIDs": ["4","5","6","7","8","9",
  {"operator": "*",
   "left":  {"colID": "5"},
   "right": {"operator": "-",
             "left": {"value": "1.000"},
             "right":{"colID": "6"}}}]}
```

Listing 1. Instantion of a projection in intermediary

4.4 Orca to Intermediate Format

The Orca's output is a file in DXl format. It is quite different from a general logical plan because it is tailored perfectly for the GP database. That means the DXL file has many Orca specific operators that need to be converted to a more general representation that the other optimiser also supports. For example the Aggregation in Orca consists of 3 smaller steps, in Calcite it is just one step.
So we had to make a tradeoff between keeping the intermediate format universal and still having special features for each optimiser, so that we don't generalise too much, so that both plans are not indistinguishable in the end.

4.5 Calcite to Intermediate Format

Calcite provides its logical plan in the RelNode data format. This data structure includes everything calcite knows about the optimised plan it produced.
A RelNode can be iterated over. If done in the correct order we traverse each logical operation after the next. This allows us to work on each operation independently and transform its information into its corresponding representation in the intermediary Format.

4.6 Intermediate Format to Spark

This component translates each operator to the corresponding RDD function in Spark. It also keeps track of the output of each operation, so that the component can map the input for each operation according to the parent ID in the intermediary. It is to mention that Spark evaluates the operation in a lazy way. That means Spark doesn't actually run the operation directly but builds up a chain of operations that is run at the end of the intermediary.

Intermediary	Spark
Selection	filter(row -> True / False)
Projection	map(row -> row_new)
Aggregation	aggregateByKey (row, aggregator -> aggregator)
Sort	sortByKey(key_columns, ascending = True/False)
TableScan	textFile(file).map(row -> row.split("|"))

5 Validation

In order to understand if our project works as intended, we first have to understand what we are expecting.
First we want to verify our optimisers. For this we make sure they take a random SQL input and produce a valid logical plan. We only make sure it is in fact a logical plan with correct syntax that allows parsing, we do not make sure it is also equivalent to its SQL Input. This is a requirement in validating an optimiser, but we have implemented the optimisers without changing their logic, as such going forward we will be assuming our optimisers to utilise a valid internal optimiser logic. We also want to see a difference in the logical plans that are produced by our optimisers. This difference is especially important because we build our project on the idea that we can find more optimal database solutions if we allow comparisons of singular components. Should we see a difference in the optimised plans between optimisers we will take a look at execution times in conjunction with the execution engine looking to see a comparison between the optimisers.
Second we look to validating our implementation of the execution engine. Here we are testing two stages. One that translation and retranslation to and from our intermediary format does not change the logical meaning of the optimised plan and by transition the SQL query. And two that the actual data output is also equivalent to the beginning of the chain.
We have implemented a single execution engine as such we will not be looking for comparisons between execution engines, instead only make sure that our one execution engine functions correctly.
The Optimiser Orca did not require a significant validation

effort, as we are not using it to translate SQL input to a logical plan. Instead we are translating the output provided by queries that are prerun on Orca into our intermediary. Calcite does take an SQL input and produces a logical plan. To validate that it is parsable, we are making sure the output always conforms to operators we can parse. Our implementation of calcite gives us this for free, as we have limited the rules calcite uses during optimisation to just those that produce logical operations we can parse.
Confirming that the intermediary format produces logically equivalent transformations in all situations is a lot harder. We can run queries in SQL on our TPC-H database that have a known valid output. Matching our output with the known one allows us to verify that our query stayed logically equivalent through the chain. We tested this and found our implementation to produce the correct results. With this we will be assuming that our intermediary format does not cause inconsistency and that our execution engine runs the operations equivalent to the logical plan.
We do not believe this to be exhaustive. A future project centred around the logical equivalency of our intermediary format and our implementation of Spark could proof valuable.
Next we will be looking to find a difference between the two optimisers Orca and Calcite. More specifically we will be looking at the intermediary format representation of their logical plan, as that is the Information that will reach the execution engine and might thus produce a difference between the optimisers. Representative we will be looking at the TPC-H Query1 where we already find a difference. While Calcite executes a projection immediately after the tablescan and the filter afterwards, Orca starts with the filter and executes the projection afterwards. We consider any difference to be a success as we are not trying to understand what the differences are but rather that there is a difference to begin with and thus a possibility of different optimisers having different performance characteristics and execution engines dealing differently well with different optimised plans.

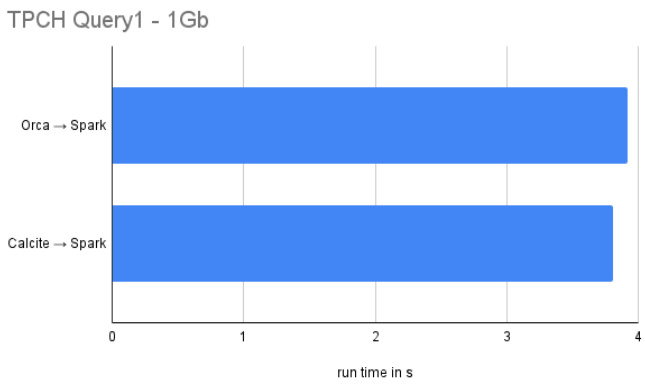

Figure 4. Runtime of TPC-H Query1 on our implementation with a 1GB sized Data set

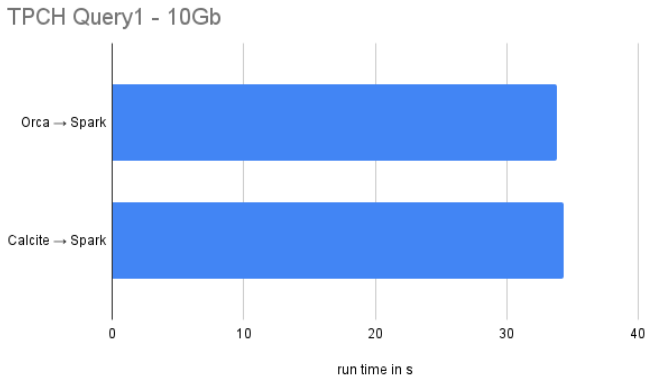

Figure 5. Runtime of TPC-H Query1 on our implementation with a 10GB sized Data set

Furthermore we tested our theory that a different optimised plan produces a different execution time on Spark. We executed the aforementioned TPC-H Query1 on our project and measured the time it took to finish. The runtime is the average over 15 runs on the same computer and with two datasets. 1 that is 1GB in size and another that is 10GB in size. There is no massive difference suggesting that both optimised plans are quite fast. But we can show that on the smaller dataset calcite performed slightly better while on the larger dataset Orca did, suggesting different strengths and weaknesses.

6 Conclusion

We have shown that there are differences between different optimisers that also translate into different performance characteristics. In our tests the differences were quite small, but present. It is highly probable that in the right type of workload those differences can be larger, potentially significantly so.
Our project thus lays a foundation towards more extensive comparisons between database components, but we believe our project is to small in scope to really provide a framework that can be used to test and compare many execution engines and optimisers to find the most ideal combinations for the right workloads.
We are translating an optimiser output into an intermediary format and back. Unfortunately this comes with a tradeof between being general and being more optimised. It currently functions under the assumption that a logical plan is a physical plan. That is not the case. Our intermediary format provides for all logical operators only one representation for the execution engine, but looking at the logical operator join there are many different ways it may be implemented on a physical level. The most prominent types being nested loop joins, hash match joins, and merge joins. Currently our intermediary format is unaware of these differences. To optimise the connection between the optimisers and execution engines many more operators could be implemented but with every one that is implemented the execution engines also need the capability to understand said operators. The more we specify the less we generalise.
The project itself also does not provide extensive options. Only two optimisers, with one being unable to receive SQL input, and a single execution engine. Including more optimisers and execution engines would be vital. A second execution engine in particular would provide significant more options.

In conclusion we provide a baseline for a composable database. Our intermediary format proving that the optimiser output can be transformed into an intermediary that allows any number of optimisers and execution engines to be compared and combined with each other. To be practically viable it only requires a larger scope than what we were able to achieve in this project.