Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Automatic big data SQL execution engine selection based on machine learning

Scientific Students' Association Report

Author:

Barnabas Maidics

Advisor:

dr. Akos Dudas

2019

Contents

Kivonat

A nagyméretű adathalmazok feldolgozása mindig is fontos szerepköre volt az informatikai rendszereknek. Jelenleg amennyiben a felhasználó egy lekérdezést szeretne futtatni egy ilyen ún. big data adathalmazon, akkor több eszköz közül is választhat. Ezek közül kettő legelterjedtebb, nyílt forráskódú megoldás az Apache Hive és Apache Impala. A két megoldást nagyon gyakran párhuzamosan használják egy valós környezetben. Bizonyos esetekben a Hive, másokban az Impala tud gyorsabb végrehajtási időt elérni. Jelen pillanatban a felhasználó feladata eldönteni, hogy melyiken fogja futtatni az adott lekérdezését. Ez a feladat viszont meglehetősen nehéz, összetett. Rengeteg paramétertől függ a végrehajtási idő: lekérdezésben szereplő táblák mérete, azok partícionálása, az SQL lekérdezésben lévő illesztések, csoportosítások száma, az egymásba ágyazott allekérdezések száma stb. Ezeken kívül feltételezhető, hogy sok más, ismeretlen, manuálisan nehezen felderíthető tényezőtől. Projektem, kutatásom célja, hogy megkönnyítsük a felhasználók feladatát azzal, hogy automatikusan kiválasztjuk az optimális végrehajtó egységet az adott lekérdezéshez. Ezzel összességében a felhasználói élményt, és a rendszerünk áteresztőképességét javíthatjuk.

A projektem neve QuDi (Query Distributor), amely egy proxy rétegként működük a felhasználói réteg és az Apache Hive, Impala között. A bejövő lekérdezést elemzi, azokból különböző változókat gyűjt (pl.: partíciók, illesztések, csoportosítások száma, a lekérdezésben szereplő táblák mérete stb.), majd ezek alapján egy machine learning modellt felhasználva eldönti hogy melyik végrehajtó egységen érdemes az adott lekérdezést futtatni a lehető leggyorsabb válaszidő érdekében. Ezen kívül képes fordítás nélkül eldönteni, hogy egy lekérdezés hosszú vagy rövid ideig fog futni, melyet felhasználhatunk a lekérdezéseink megfelelő ütemezésére.

Abstract

Processing big data has always been an essential task in the IT industry. Currently, if the users would like to run a query on a big dataset, they can choose from various solutions. Among these, two of the most popular, open-source technologies are the Apache Hive and Apache Impala. These two solutions are often used simultaneously in a real-world environment. In some cases, Hive in some, Impala can provide a faster execution time. Currently, it is the user's task to decide where to run a given SQL query. This task is very challenging and complex. The execution time depends on a lot of factors: number of tables in the given query, sizes of those, and how they are partitioned, the number of joins, group by-s, number of subqueries in the SQL query, etc. Besides that, probably many more unknown and hardly detectable factors. The goal of my project and research is to ease the work of the users by choosing the optimal execution engine automatically for a given query. Using this, we can improve the user experience and the throughput of our system.

The project is called QuDi (Query Distributor), which is a proxy layer between the UI and Apache Hive and Impala. It analyses the incoming query, gathering different variables from that (number of partitions, joins, group by-s, sizes of tables, etc.). Based on that, using a machine learning model, we can predict where to run the given query to reach the fastest response time. Besides that, it can predict if a query runs for a short or long time without compilation, which we could use to schedule the queries accordingly.

Chapter 1

Introduction

The digital era has led to large amounts of data being amassed by companies every day. Data comes from multiple sources: sensors, sales data, communication systems, logging of system events, etc.. Larger corporations can easily create hundreds of Terabytes daily. We need a new solution to process this amount of data. The traditional relational databases (RDBMS) can deal only with Gigabytes. Hadoop provides a software framework to scale up our system for storing, processing, and analyzing big data.

Hadoop is the most popular implementation of the map-reduce programming paradigm and widely used to process and store massive datasets in a distributed way. However, a map-reduce program represents a low level of abstraction and is challenging to maintain or reuse. Data scientists come from a world where SQL is the standard of querying data. Nowadays, we can choose from several solutions to query from large datasets: Apache Hive, Apache Impala, Apache Spark (SparkSQL), Presto, and many more projects that people built on top of the Hadoop ecosystem. The most commonly used open-source SQL engines are Hive and Impala, which companies often use in parallel.

Currently, if the users want to execute an SQL query, they need to decide on which engine they want to run it to get their results as quickly as possible. The decision is a hard and challenging task since the execution time can depend on various factors of the data and the query itself: number of joins, aggregations, subqueries, and also on table metadata on which the query is operating: number of partitions, rows, data size etc. An incorrect decision can result in unexpectedly long execution times that can last as long as hours.

This work introduces QuDi, a Query Distributor that aims to provide a solution for predicting the optimal execution engine for a given SQL query. Currently, it focuses on the two most frequently used big data SQL software: Apache Hive and Impala. QuDi provides a framework to analyze and gather feature variables from a query, and measure its execution times on both engines. This way, we can collect data that can be later used to build multiple machine learning models. Using this defined model, it can select the optimal execution engine for any given query automatically.

Until today, in the Hadoop ecosystem, there has not been any project to ease the job of big data analysts. With QuDi my goal is to introduce a machine learning model that we can use to predict the optimal execution engine and if a query will be short- or long-running. The predicted values can be used in multiple scenarios. I.e., the needs mentioned above should be satisfied by taking over the difficulty of choosing the optimal execution engine for the query. As an additional benefit, it would not only ease the work of our users but increase the throughput of our data warehouse system.

Having a prediction without the actual execution of the query can be valuable in multiple aspects. We can even use that for scheduling our workload. Suppose we have multiple queries hitting our execution engine in a short amount of time, and the number of queries is higher than what our engine is capable of executing in parallel. In this scenario, the traditional first-come-first-served scheduling is not beneficial. Consider the following: if our first queries are long-running ones (let us say daily, or weekly reporting queries have started), and after those several short ”interactive” queries arrive, those will wait until the long-running ones execute. This causes user frustration due to suboptimal scheduling. We should instead schedule the queries in a way that the short queries would run first. In a real-world customer use-case, it is usually not a problem to delay long-running queries - in the worst case the report will be finished a few minutes/hours after the original time -. However, with short running queries, our users most often are doing some exploratory analytics, and it is not acceptable to make them wait for several minutes or even hours until some resources will free up. Previously, we could not do such scheduling, since execution time could not be predicted before it begins to compile on the given execution engine, but having a predicted runtime will introduce the possibility of this improvement as well.

1.1 Project timeline

This section summarizes the different phases of the research so the reader can have a basic concept about the project and the wide subtopics that I covered within the area of computer science.

Understanding the query engines Since the project aims to decide the optimal executor for queries, I had to gain more in-depth knowledge about how the different solutions work and what use-cases they are optimized for in the world of big data analytics. For the prototype, I restricted the supported engines to Apache Hive and Impala. The reason behind this decision is straightforward: these are the most commonly used big data query processors, and both projects are open source, so the available documentation and knowledge are sufficient for a deeper understanding.

Creating QuDi - first phase To build a machine learning model that can be used to predict runtimes, we need a lot of data about queries. This requires an environment that can analyze a query without executing or compiling it. As a solution for this requirement, I created the specification and architecture of QuDi and, based on those, developed and later integrated the software that communicates with Hive and Impala and connects them with the users. Given a query, QuDi analyzes it saving the collected features and executes them on both engines while measuring the response time.

Gathering data At this phase, I had an infrastructure that can analyze a query. However, to have enough data for building different machine learning models, I need gigabytes/terabytes of data on a distributed filesystem (in my case HDFS - Hadoop Distributed File System) and thousands of queries. No doubt, the hardest part of the project was to obtain queries and data that can simulate a real-world use-case. Customers are not willing to make their data public, therefore to solve this, I had to come up with a different method, that I can use for collecting features from thousands (or any number of) queries.

Building the machine learning model Having data about thousands of queries made it reasonable to start the data science work, the first purpose of which was to predict if a query is a short or long-running one. At this phase of the project, I began with exploratory data analysis (EDA) to summarize the main characteristics of my dataset. Afterward, I developed multiple machine learning models and validated the performance of these and chose the best suited for my use-case.

Evaluation of the project The evaluation I did was running the same queries with and without QuDi (using the machine learning model) and checked what effects it has on the overall response time and throughput. I also considered scalability, robustness, and the security of the system.

1.2 Overview

In Chapter 2, I briefly present the used frameworks, Hive and Impala, and expound the desire to have a solution to the problems mentioned above. Chapter 3 describes the specification and architecture of QuDi and how it analyses a query. Chapter 4 demonstrates the method and model behind the solution that was used to generate thousands of queries on a given dataset. The developed machine learning models and data science workflow is introduced in Chapter 5. Chapter 6 contains the evaluation of the project, including a simulated workload. Finally, Chapter 7 summarizes the work and describes the afterlife of QuDi and the planned future improvements.

Chapter 2

Background

This chapter briefly introduces the background technologies that are necessary to understand the motivation behind QuDi and how it is working. I start with the idea behind large data processing focusing on Hadoop, continuing how SQL engines like Hive and Impala work in a high-level overview and compare the basic use-cases that the two engines were created for and how they developed in the meantime.

2.1 Hadoop [15]

Apache Hadoop is an open-source distributed framework for managing, processing, and storing large amounts of data in clustered systems built from commodity hardware. All modules in Hadoop were designed with an assumption that hardware failures are frequent and the framework should automatically handle those. One of the most important characteristics of Hadoop is that it partitions data and computation across many hosts and executes in parallel close to the data it uses. This section describes the basics of the Hadoop ecosystem that is essential to understand the two projects that build upon it.

The base of the Hadoop framework contains the following modules:

- HDFS - Hadoop Distributed File System: designed to store large data sets reliably and stream those at high bandwidth to user applications. It is divided into two main components: a Namenode, which stores data location, and many Datanodes that store data and stream it to their clients.
- Hadoop MapReduce: an implementation of the MapReduce programming model for large data processing
- YARN - Yet Another Resource Negotiator: resource management and job scheduling technology

The basic principle of Hadoop A computation is much more efficient if we execute near the data it operates on. This is especially true for big data.

2.1.1 MapReduce [8]

Understanding the MapReduce paradigm is a must to understand how Hive works (and in the latest Hive, LLAP).

An input to a MapReduce algorithm is a list of key-value pairs $\{\langle k_1, v_1\rangle, \dots, \langle k_i, v_i\rangle, \dots, \langle k_N, v_N\rangle\}$

Definition A mapper μ takes an input $\langle k, v\rangle$ and produces a list of key-value pairs $\langle k'_1, v'_1\rangle, \dots, \langle k'_s, v'_s\rangle$

Definition A reducer ρ takes an input key k and a list of values $v_1, \dots, v_m$ and produces the same key and a new list of values $v'_1, \dots, v'_M$.

Definiton A MapReduce program M is an alternating list of mappers and reducers $M = (\mu_1\rho_1, \dots, \mu_K\rho_K)$. It executes for each $k = 1 \dots K$:

1. U_{r-1} is a list of key-value pairs from the output of the previous round or the input of the MapReduce program if $r = 1$. We will apply μ_r for each key-value pair of the list. The output will be $\bigcup_{\langle k,v\rangle \in U_{r-1}} \mu_r(k, v)$
2. Shuffle and sort phase: group the values by key. $V_{k,r} = \{k, (v_{k,1}, \dots, v_{k,s})\}$
3. Assign a ρ_r for each $V_{k,r}$ that can be run parallel and the output will be $U_r = \bigcup_k \rho_r(V_{k,r})$

2.1.2 Advantages of MapReduce

Parallel Processing In MapReduce, we divide the job among multiple nodes (Datanodes), so they can work on their part of the data parallelly. This way, data processing is done by multiple machines instead of one, so the time is significantly reduced.

Data Locality In Hadoop MapReduce, instead of moving data into the processing unit, we move the processing unit to the data. The traditional approach has its limitations when it comes to processing big data. Moving large amounts of data is costly: network issues can occur, and the master node (where data is stored) can get overloaded and may fail.

Using Hadoop, we need to provide the map and reduce functions, and the framework handles the rest: parallel, distributed execution.

2.1.3 The need for SQL engines

The traditional way to execute MapReduce operations is that the users specify the Map and Reduce functions in Java. However, this approach has some problems:

- It is not a high-level language for data processing
- Data scientists do not understand Java. They came from the world of traditional databases, where SQL is used.
- Even a simple problem can result in hundreds of lines of code.

Hive gave a solution, and later multiple other projects were created to fulfill this need.

2.2 Hive

Apache Hive gives us a data warehouse solution built on top of Hadoop to write SQL-like queries so we can utilize the advantage of a declarative language. Its language - similar to SQL - is called HiveQL. Hive is a great choice if we want to analyze large, relatively static data sets, and easy, low-cost scalability is required.

2.2.1 Architecture

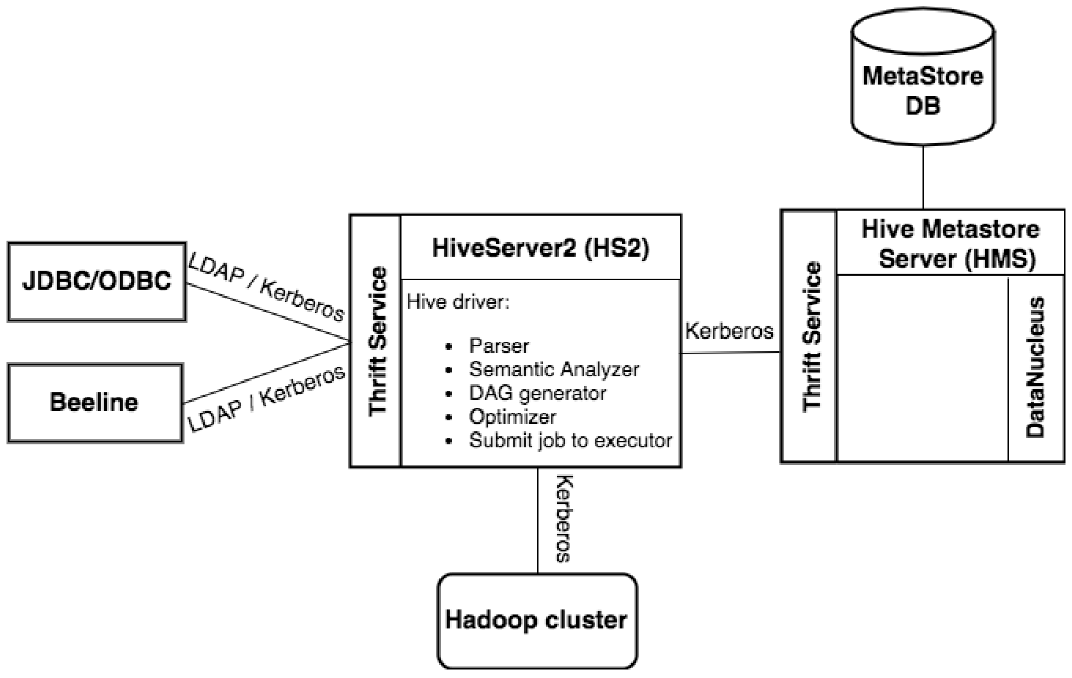


Figure 2.1: Hive architecture

The main components of Hive are the following [16]:

- **Driver**: manages the lifecycle of a HiveQL query. The Driver also collects the result after the execution phase.
- **MetaStore**: stores metadata about tables, columns or partitions. For example, it stores the table schema and location of the data.
- **Compiler**: compiles the HiveQL statement and generates the execution plan using the partition and table metadata obtained from the MetaStore. First, it parses the query and does semantic analysis. The compiler then converts it into AST (Abstract Syntax Tree), and after compatibility checking, to a DAG (Directed Acyclic Graph) of Map and Reduce tasks (if Hadoop MapReduce is the execution engine).
- **Optimizer**: transformations are performed to optimize the DAG for better performance. It is an evolving component. Previously, only rule-based optimization was available, which performed column pruning and predicate pushdown. Later, map-side join was introduced, and several other join optimizations, as well as cost-based optimization, were added.
- **Execution Engine**: executes the plan created by the compiler. The plan is a DAG of stages. The engine manages the dependencies between these stages and executes them in the corresponding component. Hive is compatible with three execution engines: MapReduce, Apache Tez, and Spark, which can run on Hadoop YARN. In the case of MapReduce, the Map and Reduce jobs will be distributed between Datanodes in a way that the previously mentioned data locality is maintained.
- **HiveServer2**: a service that provides a Thrift interface so clients can execute queries against Hive. Thrift is an RPC (Remote Call Procedure) framework for defining services for multiple languages. HS2 is also the process that contains the Driver.
- **Clients**: multiple clients are available to interact with Hive. Examples include Beeline (CLI), JDBC/ODBC etc..

2.2.2 Data Storage

Hive structures data in the following units [16]:

- **Databases**: namespaces to avoid conflicts of table, partition, or bucket names.
- **Tables**: a storage unit for data with the same schema. Tables map to directories in HDFS.
- **Partitions**: Tables can have multiple partition keys. These will determine how data is stored. In HDFS, partitions map to subdirectories in the table's directory, which speeds up analysis. Instead of running the query in the whole table, Hive will only run our query in the relevant partitions.

2.2.3 Limitation of Hive on MapReduce

Hive was created to work on top of Hadoop, creating a MapReduce program that can be executed by the framework. But Hadoop MapReduce is batch-oriented, so it is not suitable for interactive queries, so developers had to choose between two solutions. Create an entirely new framework from scratch for interactive querying (Impala), or improve the already existing Hive until it is capable of serving such use-cases (Hive Tez and Hive LLAP).

2.2.4 Hive on Tez [7]

Tez is a framework that was built on top of Hadoop Yarn to execute DAGs of data processing tasks and can be used as an execution engine in Hive, apart from MapReduce. Tez improves Hadoop MapReduce in multiple aspects:

Multiple reduce stages With Tez, several reducers can be linked directly, without the need of mappers writing temporary files to HDFS. Using this feature, Hive can create a more sophisticated DAG that can drastically speed up the execution time.

Pipelining With Tez, Hive can send the entire query plan at once, allowing the framework to allocate the resources more intelligently and pipeline data through the stages.

Less disk access When executing a MapReduce job, it typically looks like the following: Mappers read data from the file, they finish their task and write out the result to the disk. Followed by running the shuffle and sort phase, which rereads the data, and when finished, writes it. Afterward, reducers will read the sorted data and execute their task, finally save the final results to disk. That means 6 disk access overall. With Tez, it first executes the plan without accessing data; once everything is ready, it will get the data from disk and produce the output. Furthermore, Tez allows smaller datasets to be handled entirely in memory. It is beneficial for queries that are aggregating or sorting a smaller dataset after the filtering is done.

Summarizing, Tez is the successor of MapReduce, which uses Hadoop containers (YARN) efficiently, can link multiple reduce phases without maps, and effectively uses HDFS.

2.2.5 Hive LLAP [1]

Hive introduced the Long Live and Process (LLAP) functionality to make the engine more interactive and competitive with its challengers, like Impala (introduced in the next chapter).

LLAP is not an execution engine (like MapReduce or Tez). The execution is scheduled and monitored by Tez over LLAP nodes, and an LLAP daemon handles I/O, caching, and query fragment execution. The daemons have in-memory data cache, which can accelerate queries if common data is accessed multiple times. To avoid long startup times, LLAP uses persistent query servers.

The following diagram summarizes the differences - that I presented in this chapter - between Hive on MapReduce, Hive on Tez, and Hive LLAP.

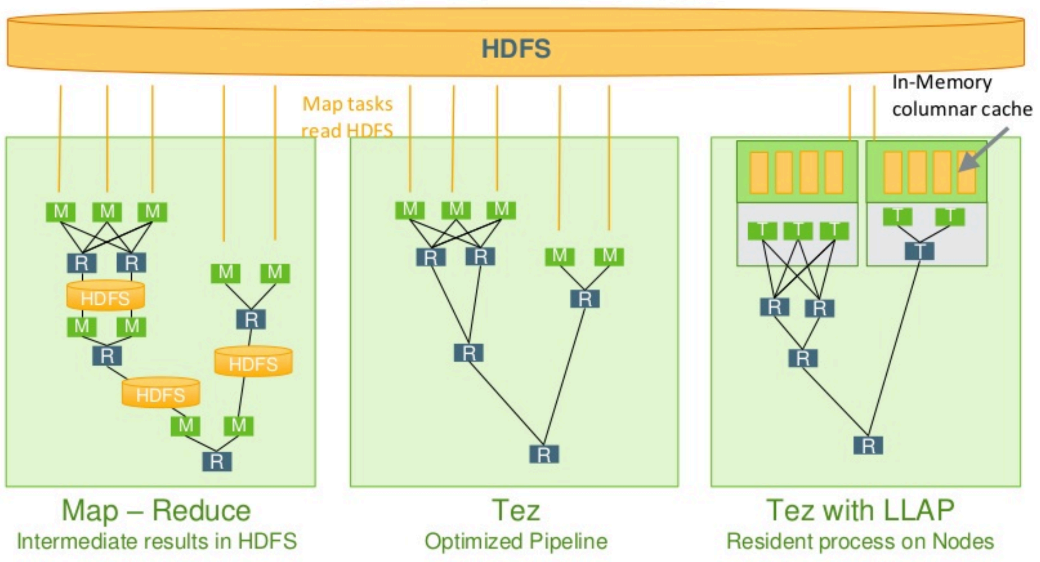


Figure 2.2: Hive modes [14]

2.3 Impala [5]

Originally, Impala gave a solution for a use-case Hive could not cover at that time (because only Hive on MapReduce existed). Impala is an open-source massively parallel processing SQL framework that builds upon Hadoop. It provides an engine for interactive, real-time querying of big data.

Impala took over many existing solutions from Apache Hive, including its Thrift interface, and Hive MetaStore, where table and column metadata are stored. The core Impala component is the daemon that runs on each slave node of the cluster, accepts queries, reads, writes data files, and distributes the work across the other daemons running in the cluster.

Another component of Impala is the Statestore, that periodically checks the health of all the Impala Daemons. If one goes down due to any reason (hardware, software failures etc.), it informs other nodes to avoid sending query parts to the unavailable daemons.

The Catalog Service is a component of Impala, that relays the metadata changes done in Impala (e.g. creating new tables, repartitioning etc.) to all Impala daemons so that they can be used for querying data from the new tables. It is essential to mention that Impala can also query data from tables that we created through Hive (and vice versa).

2.4 Problem statement

Hadoop gave us a scalable, distributed framework to process big data, in not a very user-friendly manner. First, Hive provided a SQL engine that was able to execute declarative queries over Hadoop MapReduce, but it was designed for batch processing rather than interactive querying. Impala made it possible to run real-time analytic queries over Hadoop. However, in the meantime, Hive improved its interactive querying capability as well, with Apache Tez and later with LLAP.

Currently, users need to decide which one to use for every workload or query, and the decision is difficult to make. We cannot draw a strict border between the use-cases when to use Hive or Impala. My question for this issue was why the users have to choose themselves? Their goal is simple: get the results as quickly as possible, without the challenge to analyze, categorize, then execute the query on the engine that they believe is more capable of giving results in a reasonable time. In the following chapters, I introduce my project, QuDi, that aims to answer the question automatically: should I use Hive or Impala for my query?

Chapter 3

QuDi - Query Distributor

In the previous chapter, I presented the background technologies and the reason behind starting this project. In the following, I introduce the architecture of QuDi and how I managed to collect data from a given query without even compiling it.

3.1 Architecture

QuDi introduces a new layer between the users and the supported SQL engines. It works like a proxy between them. The following figure illustrates the architecture of QuDi.

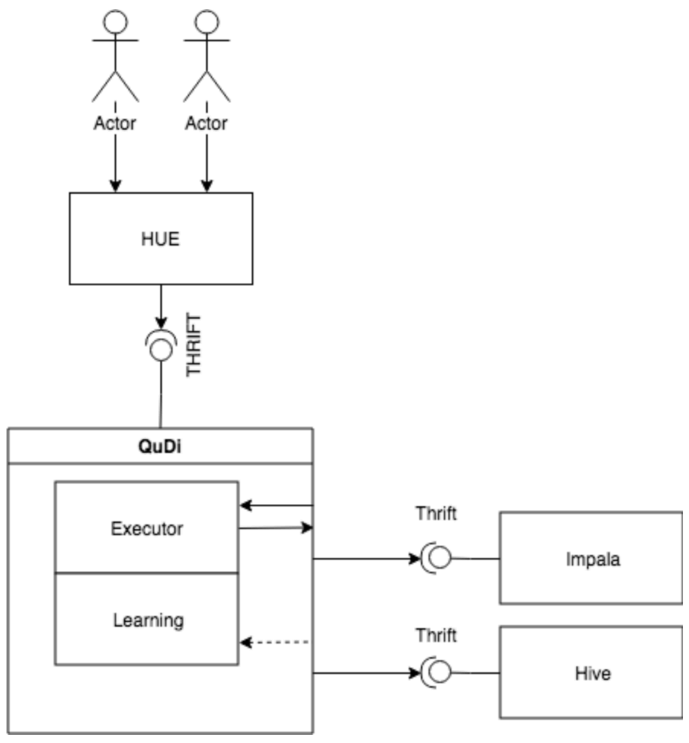


Figure 3.1: QuDi architecture

Nowadays, most of the users work with HUE (Hadoop User Experience) to launch their query, but QuDi can work with any client (i.e. Beeline, JDBC clients etc.). HUE connects to Hive or Impala using their Thrift interface, which is the interface of HiveServer2. QuDi also implements the same interface, so users do not recognize the presence of the new layer, it is transparent to them and works as a server for the users, and as a client for Impala and Hive.

The plan is that QuDi will have three basic modes: turned-off, learn, and execute mode.

- When QuDi is in turned-off mode, it just routes the query to the engine that was originally selected by the user.
- During learning mode, it will run the same query on all the engines, and collect metrics from it. When QuDi gathered enough data, it builds a model that will we can use for prediction.
- During execution mode, QuDi will use the model to select the best engine and executes the query on it. At certain queries (e.g. periodically), QuDi can adapt to the changes made on the data by executing some queries on all the engines and improve its model.

3.2 Collecting metrics from a query

After implementing the communication between clients, QuDi, Hive, and Impala, my first step was to analyze a query when it reaches QuDi. It is a must to collect data that we can use later for building several machine learning models.

The right place to analyze a query is before it is executing. The Thrift method that starts the execution is called *ExecuteStatement*, so in the implementation of this function in QuDi, I inserted the analyzer calls. I collect two types of metrics during the analyzation: query features and table metadata.

3.2.1 Query features

Query feature means the metrics that I can collect just by looking at the query string itself. The full list of these features:

- number of select
- number of where
- number of different tables
- number of columns
- number of orderby
- number of groupby
- number of unionall
- number of function
- number of casts
- is disdinct
- number of if
- number of join
- number of rightouterjoin
- number of leftouterjoin
- number of subquery
- number of having
- number of in
- number of ifexists
- number of ifnotexists
- number of aggregates

Getting these features from only using the string representation of the query can be challenging in certain cases: how can we identify the number of joins or the number of subqueries?

Suppose we have a θ-join ($\bowtie_\theta$) between R and S tables:

$$\underset{a\ \theta\ b}{R \bowtie S}, \text{ where } \theta \in \{<, >, \leq, \geq, =, \neq\}$$

This type of join can be expressed in two different way in SQL, since

$$R \underset{\theta}{\bowtie} S = \sigma_\theta(R \times S)$$

. Translating relational algebra to SQL where θ is '=':

select ... from R join S on R.a = S.b $\Leftrightarrow$ ***select ... from R, S where R.a=S.b***

As a result of these equivalencies, identifying joins that can be expressed by multiple syntaxes would be challenging and time-consuming. An even more difficult task would be to count the number of subqueries in the statement. Of course, writing a complicated parser that recognizes when a new subquery begins and when it ends, and that can recognize different join syntaxes can be an approach. However, looking back to how Hive works, we can find a simple solution to this: Hive has its parser, which creates an AST (Abstract Syntax Tree) of the given statement. This tree contains the tokens, like "tok_subquery" or "tok_join" etc., and transforms joins which are represented by where statements into join tokens.

Take the following query as example: ***select a, b, c from A, B where A.a = b.b group by c***

The Hive Parser represents this in a tree format:

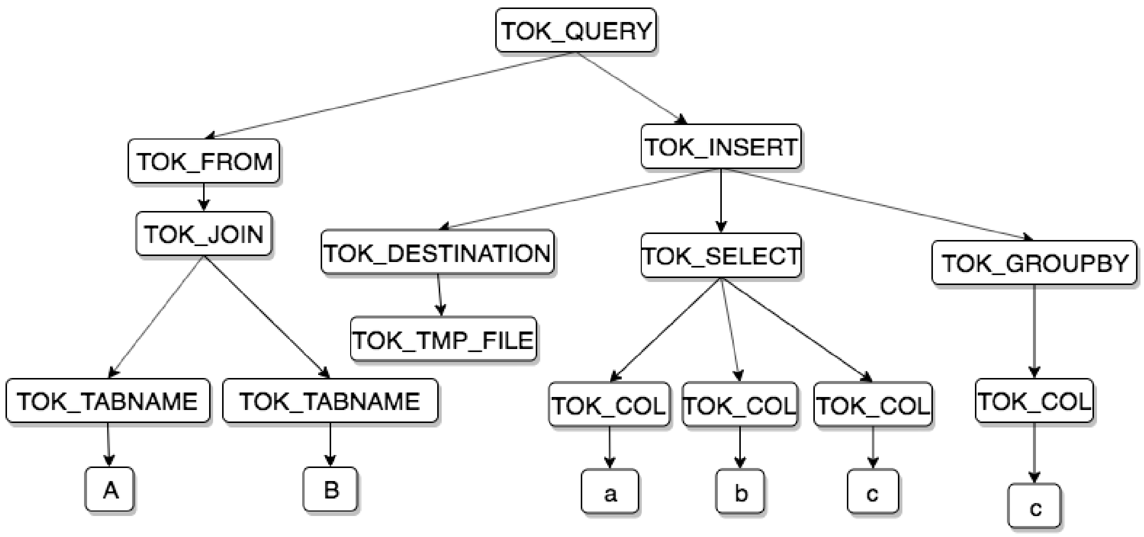


Figure 3.2: Example of an AST

Walking through this AST gives us the correct number for every feature we want to collect from the query without writing a completely new parser for my need.

3.2.2 Table metadata

We can gather more data from a query in milliseconds without compiling it. As previously mentioned, Hive and Impala stores metadata in a separate component called the Hive MetaStore (HMS). It provides a Thrift interface for clients, which I can use to get the metadata from tables that are present in the query. Using the parser, I can identify the tables that the query contains and get information about them from the Metastore.

For building a prediction model, it is ideal to have a fixed number of features. The values that we can collect from HMS are table-level metadata. Thus for getting an accurate picture of the presented tables, we should collect those for every table. Therefore, the number of features would be proportional to the number of tables. As a restriction, instead of storing the values for each table, QuDi will aggregate them: storing the average and maximum of those. E.g., rather than collecting the number of partitions for all the tables in the query, QuDi will calculate the average and maximum partition numbers.

The full list of metadata features:

- average raw data size
- max raw data size
- average number of files
- max number of files
- number of casts
- average buckets
- max buckets
- average number of rows
- max number of rows
- average total size
- max total size
- average partitions
- max partitions
- number of partition keys
- number of text tables
- number of compressed tables
- number of parquet tables
- number of orc tables

Most of the above features are self-explaining however, some of them needs an explanation:

Number of text/orc/parquet tables: Both Hive and Impala can work with different file formats. The simplest among these is the text format, which stores the text representation of the data. ORC and Parquet formats are quite similar: to simplify them, both are column-oriented formats and store metadata of the data itself.

3.2.3 Measuring execution times

Now that we have our predictors (features), we still need the target variables, which in this case will be two numbers: the execution time of our query with Impala and execution time with Hive. Right before passing the request from QuDi to Hive or Impala, we start a timer that will measure the execution time of the given query. Query execution on Hive and Impala begins in parallel. When the original execution engine finishes (the one that the user wanted to connect to), QuDi immediately returns the resultset. After the second engine finished, the execution times, as well as the feature variables, are saved in a CSV file that I will use for creating the prediction model. The following sequence diagram helps to understand the introduced process of analyzing and executing a query.

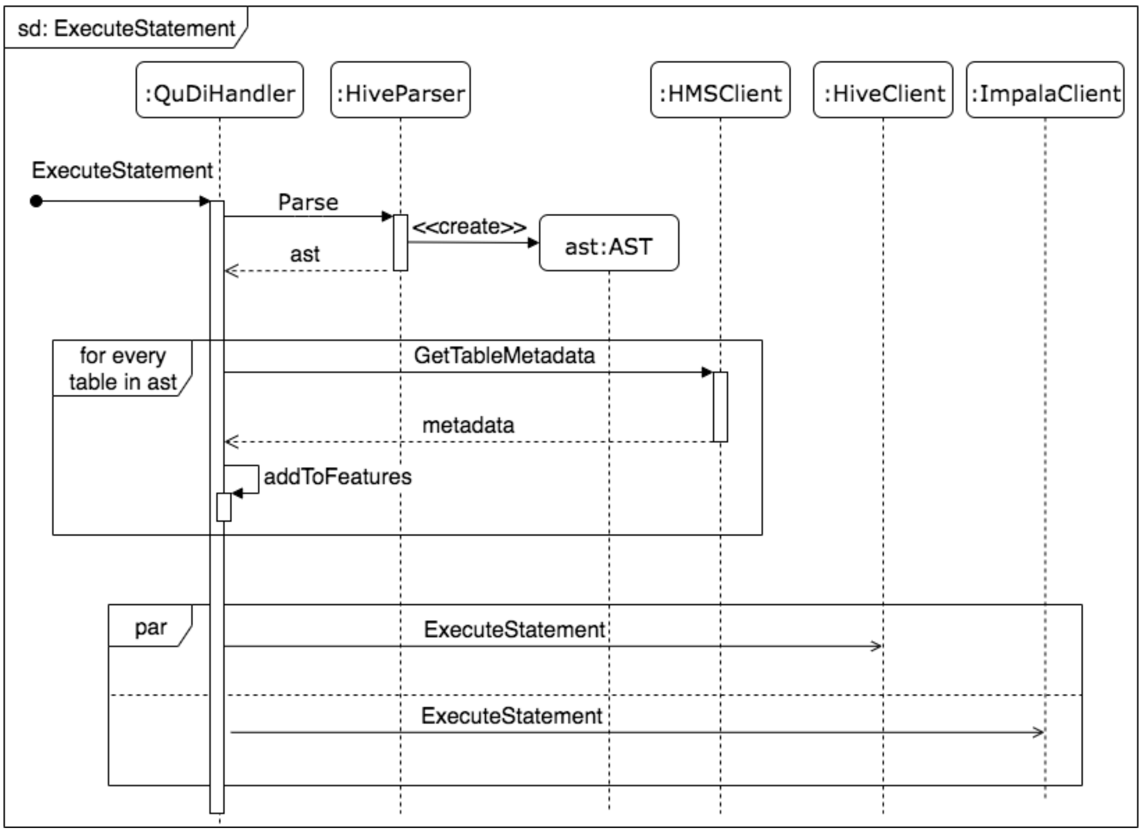


Figure 3.3: Execution of a statement

Chapter 4

Data collection

QuDi can analyze a query, collects 40 different features, executes the statement on both engines, measures the execution times, and finally saves these to a file. The only thing that is missing from building a machine learning model is thousand of lines of data. This chapter presents the two ideas I investigated to collect data and queries to simulate a workload.

4.1 First approach - Real workload

The obvious approach would be to use a real workload with thousands of different queries and analyze those using QuDi. As obvious, it seems as difficult it is to accomplish. The three main problems with this method:

- **Data volume**: It means around hundreds of Terabytes or even Petabytes when we speak about a real workload. Even storing this volume would require a huge cluster, not mentioning the computation resources, which are much more expensive. Running a single Impala or Hive LLAP daemon on such a volume would require a minimum of 60 Gigabytes of memory, and it would not be enough to run only one daemon. Taking into account that this cluster would run for weeks to collect enough data, it would be a costly approach.
- **Privacy**: It is not easy to get terabytes of meaningful data and schema information for them. Companies would never publish this amount of data since it harms their privacy or business.
- **Queries**: Even if we had a meaningful amount of data and a schema above them (e.g. star schema), another issue would be to get queries that can run on the defined dataset.

To sum it up, collecting data using this approach is extremely difficult. Once QuDi is production-ready, it would run on such environments, but until then, a basic model is essential to launch the project. Eventually, QuDi would change and adapt its internal prediction model to the current data and workload that the customer has.

4.2 Second approach - Data and query generation

Synthetic data generation can be used to generate a dataset that we can query later and build a model based on it. A pure synthetic method would create a dataset that is far away from an actual analytic dataset. Therefore, we should take into account the data domains and schemas that are usually used in the industry.

4.2.1 TPC-DS dataset [10]

Transaction Processing Performance Council (TPC) is an organization founded to define database and transaction processing benchmarks. Their Decision Support standard is a benchmark to measure the performance of decision support systems. This includes a schema definition that simulates a business database. TPC-DS provides the possibility to scale the volume of data to our needs. In this section, I will briefly introduce the schema of the TPC-DS dataset that I used to simulate a real-world use-case.

TPC-DS schema TPC-DS implements a so-called snowflake schema, which is a hybrid solution between 3NF and pure star schema. It includes large fact tables and several small dimension tables. Fact tables are frequently growing, while dimensions store data that do not change regularly. In addition to the pure star schema, dimension tables can not only have relations with fact tables but with other dimension tables as well.

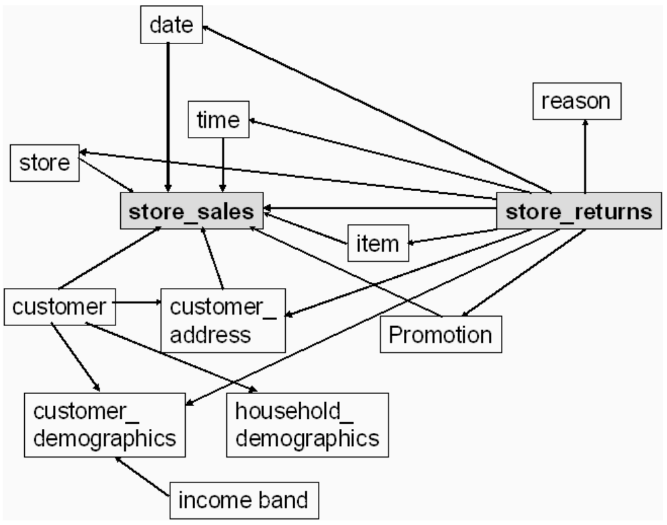


Figure 4.1: TPC-DS example of snowflake schema

4.2.2 Generating random queries

TPC-DS also provides queries that simulate a real workload. I intended to use these queries to collect data for the machine learning models, but the defined 100 queries which TPC-DS provides is inadequate for that purpose. To build a proper model, I need thousands of queries. The only solution that left is to generate queries randomly. In this section, I introduce two already existing solutions that I investigated and explain which one I chose and how it generates queries that cover a wide section of the SQL language.

4.2.2.1 SQLSmith [9]

SQLSmith is a domain-aware fuzz tester, which means it provides random data as input to a program. It targets database systems while generating random queries and aims to find bugs in the database under test (DUT). The basic idea behind SQLSmith also applies to the other framework I investigated.

The query generation begins with investigating the given database for available tables, columns, and types. After analyzing the database, it starts producing SQL statements. It works as a probabilistic recursive model. SQLSmith selects the statement type and fills its clauses randomly (FROM, WHERE, GROUP BY, ORDER BY). When it needs an expression, it chooses one from the following: function, the result of unary, binary operators, table, or column expressions etc.. If an expression needs some arguments - i.e. the '+' operator takes two arguments - they might also require an expression which can continue to select one from the presented list. This will recurse until a given point. As it is a recursive algorithm, it has to stop at some point. SQLSmith defines a boundary that limits the generation of new expressions so that the system can execute the query in a reasonable time.

Consider the statement

```
SELECT email_address,
        Concat(first_name, last_name)
FROM customer
WHERE birth_year >= 1996
        AND birth_country = 'Hungary';
```

The below figure shows a simplified parse tree of the query.

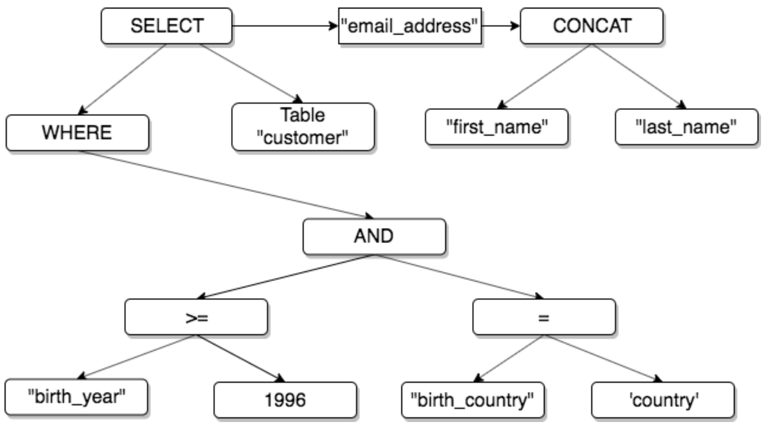


Figure 4.2: Parse tree for the above statement

Walking through this parse tree, we can read out the query manually. SQLSmith and the other project introduced in the next section - Discrepancy Searcher - does the same, except that it builds the tree stochastically as it walks it.

Drawbacks of the project SQLSmith is a great tool for fuzz testing of our system, that aims to break our database. I spent two weeks connecting the C++ project to Apache Hive and managed to make it work so it can analyze a given Hive database and collect schema information about those. However, it was not aware of the HiveQL language, which mostly follows the SQL standard but has some limitations and differences. It resulted that only around half of the queries could run on Hive and Impala as well. Therefore I decided to search for another option.

4.2.2.2 Discrepancy Searcher [2]

Apache Impala already had a similar solution to SQLSmith. The community used it to find discrepancies between Impala and a second reference system, like PostgreSQL, Oracle, or Apache Hive. It generates a query, using the same idea as presented above and runs the query on Impala and the reference engine. Afterward, analyzes the resultset of the statement and reports if it finds any discrepancy. The great feature of the Discrepancy Searcher that it is already aware of Hive's and Impala's SQL grammar and can generate syntactically and semantically correct statements.

The stochastic model it uses relies on a QueryProfile object. This contains probabilities of different clauses, functions, operators, etc.. The QueryProfile also contains the boundaries of the query being generated. This allows the user to customize the type of queries.

```
self._bounds = {
  'MAX_NESTED_QUERY_COUNT': (0, 2),
  'MAX_NESTED_EXPR_COUNT': (0, 3),
  'SELECT_ITEM_COUNT': (1, 8),
  'WITH_TABLE_COUNT': (1, 3),
  'TABLE_COUNT': (1, 5),
  'ANALYTIC_LEAD_LAG_OFFSET': (1, 100),
  'ANALYTIC_WINDOW_OFFSET': (1, 100),
  'INSERT_VALUES_ROWS': (1, 10)}
```

Also, the user can change the weights of different query features, that the tool will use to determine probabilities. The probabilities of some clauses can be independently set by the user, as shown in the example below.

```
self._weights = {
  'SELECT_ITEM_CATEGORY': {
  'AGG': 7,
  'ANALYTIC': 5,
  'BASIC': 5},
  'TYPES': {
  Boolean: 1,
  Char: 1,
  Decimal: 1,
  Float: 1,
  Int: 10,
  Timestamp: 1},
  'JOIN': {
  'INNER': 90,
  'LEFT': 30,
  'RIGHT': 10,
  'FULL_OUTER': 3,
  'CROSS': 1}, ...
}
self._probabilities = {
  'OPTIONAL_QUERY_CLAUSES': {
  'WITH': 0.1,
  'FROM': 1,
  'WHERE': 0.5,
  'GROUP_BY': 0.2,
  'HAVING': 0.25,
  'UNION': 0.1,
  'ORDER_BY': 0.2},
  'OPTIONAL_ANALYTIC_CLAUSES': {
  'PARTITION_BY': 0.5,
  'ORDER_BY': 0.5,
  'WINDOW': 0.5} ...}
}
```

After setting up the environment and upgraded some deprecated dependencies, I made it working and turned off the execution of the queries since I planned to execute them differently. With the Discrepancy Searcher, I am now able to generate thousands of queries for the TPC-DS database in minutes.

```
SELECT a2.cc_employees,
       a1.hd_income_band_sk,
       COALESCE(Cast (a2.cc_company AS BIGINT) - Cast (a2.cc_division AS
BIGINT),
       a2.cc_closed_date_sk, Cast (a1.hd_demo_sk AS BIGINT) - Cast (
                             a2.cc_open_date_sk AS BIGINT)) AS int_col,
       Sum(a2.cc_employees)                                 AS int_col_1
FROM   household_demographics a1
       INNER JOIN call_center a2
               ON ( Cast (a1.hd_demo_sk AS BIGINT) - Cast (
                    a2.cc_company AS BIGINT) ) =
                  ( a2.cc_company )
GROUP  BY a2.cc_employees,
          a1.hd_income_band_sk,
          COALESCE(Cast (a2.cc_company AS BIGINT) - Cast (
                   a2.cc_division AS BIGINT),
          a2.cc_closed_date_sk, Cast (a1.hd_demo_sk AS BIGINT) - Cast (
                                a2.cc_open_date_sk AS BIGINT))
```

Figure 4.3: An example of the generated queries

Using this tool, I generated thousands of different queries and executed them using Beeline that is a console JDBC client for Hive and can also work with Impala. When more than a thousand queries have executed through QuDi, on the TPC-DS database of a scale of 100 GB, I had enough data to start the Exploratory Data Analysis (EDA) on the collected results.

Chapter 5

Building Machine Learning models

Predicting where a query will run faster can be approached from multiple directions. We can define the task as a binary classification: the target variable would be the execution engine for the query. The other solution would be to consider this as a regression: predict the runtime for both Hive and Impala - using different models - and choose the engine with the smaller number. The latter would require a massive amount of data gathered by QuDi to provide an accurate prediction. Since the queries for larger dataset - 100 GB in this case - vary in runtime from milliseconds to hours, it takes weeks or even months to collect a meaningful amount of data.

The original goal is to extract patterns and predict the optimal execution engine. However, this requires weeks to collect enough data. In the meantime, during the collection, a logical step was to start with a simpler task, what we can use for another use-case and the data preparation part can be reused in the execution engine prediction. This task is to predict if the query is a short- or long-running one. For this classification, fewer data proved to be enough. This chapter introduces the models and results of the two different goals. The first part focuses on the classification of the query (short or long-running), the second one will introduce a prototype model developed for predicting the optimal SQL engine and the issues that it has and needs a solution in the future. The following diagram summarizes the data science workflow I used during the research.

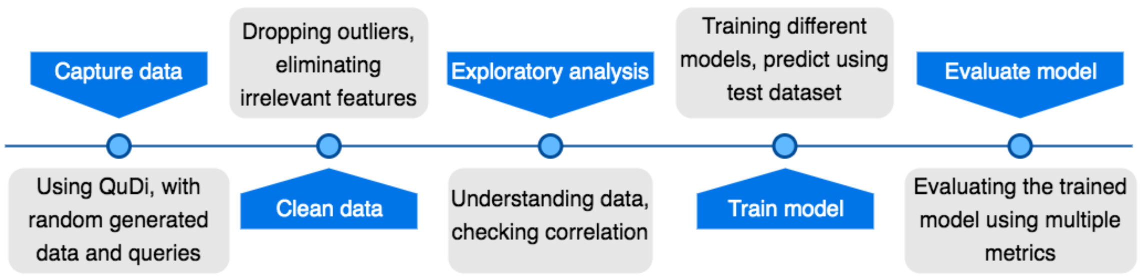

Capture data
Using QuDi, with random generated data and queries
Dropping outliers, eliminating irrelevant features
Clean data
Exploratory analysis
Understanding data, checking correlation
Training different models, predict using test dataset
Train model
Evaluate model
Evaluating the trained model using multiple metrics

Figure 5.1: Data Science Workflow

5.1 Cleaning the data

For both prediction models, I used the same data cleaning techniques. This included finding and dropping the outliers, but replacing nulls was not necessary since the dataset does not contain any rows that have empty values.

5.1.1 Dropping outliers

First, it is necessary to clarify what we consider as an outlier. Generating random queries has some serious drawbacks. It can generate statements that can run for an extremely long time hence preventing the collection of ML data. To overcome this issue, I maximized the execution time for both engines in 2 hours. Hive and Impala both ran on two nodes with 45 GB of available memory. On a scale of 100 GB, queries running over 2 hours are probably bad queries that can run even for days. These should be stopped, and QuDi should continue executing the other queries. This will result in rows, where the execution time is close to the maximum allowed. We should drop these rows not to confuse our model. However, before that, we should look at the data to identify more outliers that should also be deleted.

	hiveExecTime	impalaExecTime
mean	8.781037e+04	5.010635e+04
std	5.310370e+05	4.956381e+05
min	1.200000e+01	1.100000e+01
25%	3.040000e+02	7.525000e+01
50%	1.558500e+03	3.785000e+02
75%	1.580100e+04	1.692500e+03
90%	6.959900e+04	8.326700e+03
98%	7.708386e+05	2.024716e+05
max	7.112459e+06	7.200016e+06

Table 5.1: Basic statistics of the execution times

The above table describes the basic statistical measures of the two execution times in milliseconds. In addition to the quartiles, I calculated the 90th and 98th percentile to determine what I should consider as an outlier. The standard deviation is an order of magnitude bigger than the mean of the execution times, which is not surprising because we used random queries that cover a vast area of the SQL language and vary greatly. Based on the performance measures discussed in the following section, I decided to drop the rows where the execution time on Hive or Impala is below the 2nd percentile and above the 98th percentile helping the model to learn frequent patterns.

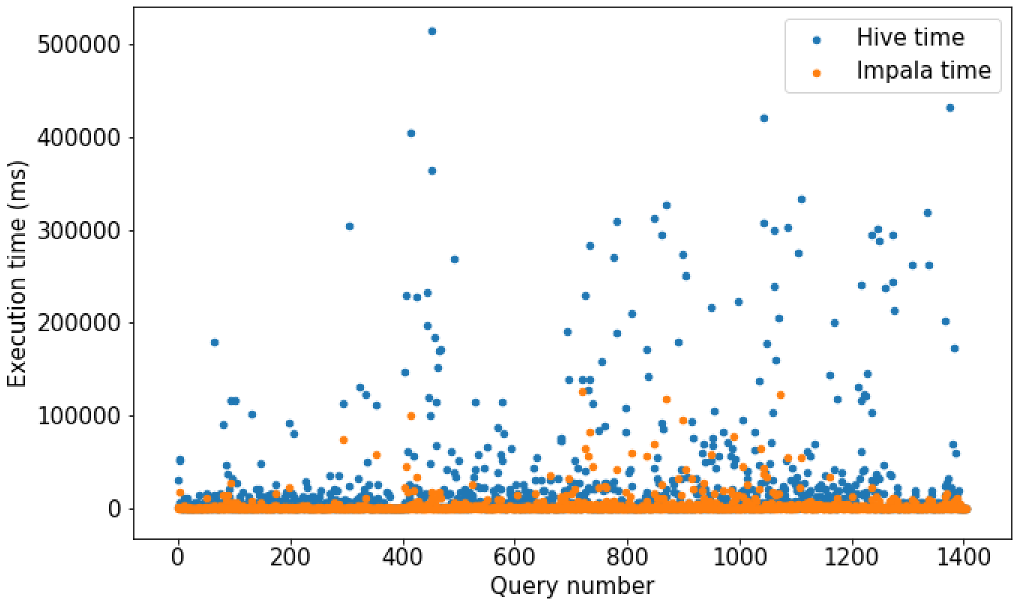


Figure 5.2: Execution times of both engines

If we look at the above scatterplot representing both execution times, we can identify another type of outliers. Some queries have a huge difference in the runtime of the two engines. Using random queries can cause this issue. E.g., for some queries that do not have a resultset, either Hive or Impala could optimize based on the statistics, or just simply on one of the engines, a query can stuck executing, causing a significant delay for the other queries. Our first intention would be to keep this type of rows since it can contain useful information. However, the sample size of these type queries are very small, so these values would only confuse our model because they are distinct from the rest of our data.

To identify the queries to be dropped, I introduced a new column,

$$ratio = \frac{\max(hiveExecTime, impalaExecTime)}{\min(hiveExecTime, impalaExecTime)}$$

Using this variable, I was able to identify the rows that differ significantly in execution times. Figure 5.3 shows the original ratio of the queries. We can see that some of them have a huge difference in runtime, causing the y-axis in the plot to cover a wide interval. Figure 5.4 represents the filtered dataset, where I dropped the rows with a ratio greater than 25 (red dots in Figure 5.3). This way, I kept 90% of the rows, which caused a great improvement in our classification models. In the future, when real queries (not random) will run through QuDi, and we have enough samples of queries with a bigger ratio in execution times, I will have to investigate these separately. However, for now, I will let the machine learning model recognize patterns in the majority of the queries.

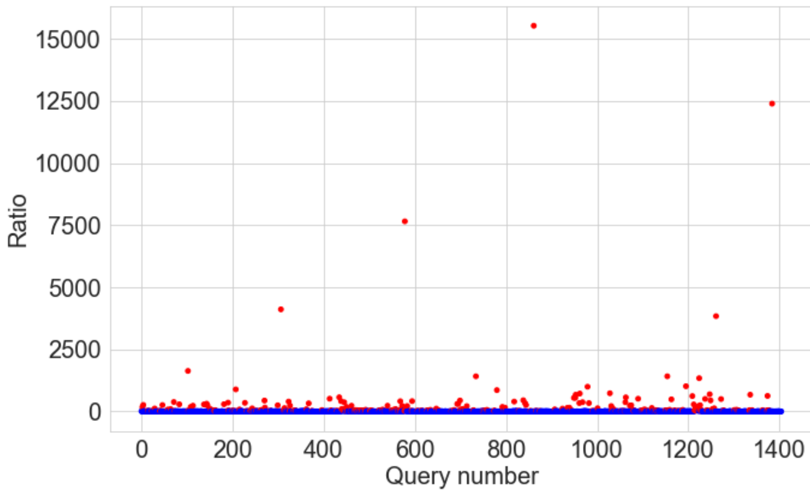


Figure 5.3: Ratios without filtering

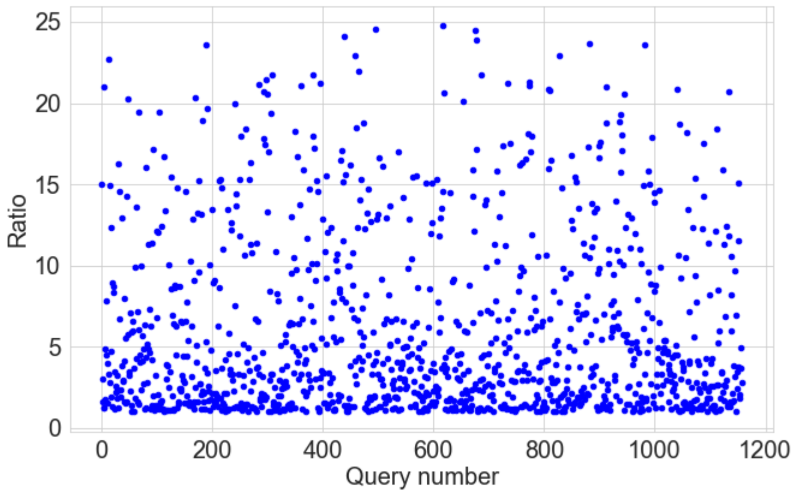


Figure 5.4: Ratios after filtering

In the upcoming sections, I present the iterative work on the machine learning models. First, I created several without any feature elimination, measured how they perform, and selected the best suited for the task. After that, I tried several feature elimination techniques and retrained the model, checking if we can identify improvements. This part focuses only on the simplified classification: predict if a query is short- or long-running.

5.2 Measuring the performance of a model [13]

Identifying how the different classification algorithms perform is a must to select the best model for the task. This section presents the metrics and the sampling method I used during the model creation.

5.2.1 Jaccard index (accuracy)

This metric is the easiest to interpret. Given two arrays, X and Y

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Assume that X is an array of the actual values of the target variable, and Y is an array of the predicted values. In this case, the Jaccard Index is the rate of the correctly classified rows. This score can be a poor metric if there are no positive values for some classes so, besides this value, other metrics are needed to get a clear picture of the models.

5.2.2 ROC-AUC [11]

One of the most commonly used evaluation metrics in binary classification is the AUC score. Before presenting the ROC curve and AUC score, the concept of confusion matrix must be understood.

The confusion matrix is also an evaluation metric used in binary classification. It is usually represented in the following table format:

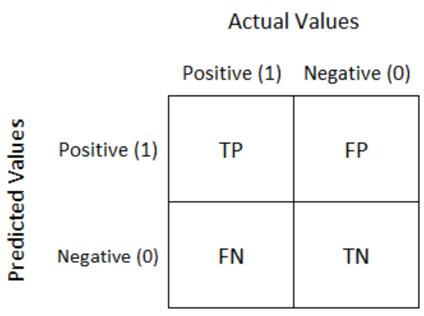


Figure 5.5: Confusion matrix

- TP: We predicted positive correctly
- FP: We predicted positive, but it was false
- TN: We predicted negative correctly
- FN: We predicted negative, but it was true

The ROC curve uses the following terms based on the confusion matrix:

TPR: how much we predicted correctly out of the positive classes

$$TPR = recall = \frac{TP}{TP+FN}$$

FPR: how much we predicted correctly out of the negative classes

$$FPR = \frac{FP}{TN+FP}$$

The **ROC** (Receiver Operating Characteristic) combines the FPR and TPR into one plot where TPR is on the y-axis, and FPR is on the x-axis. It first computes the TPR and FPR metrics with many different classification thresholds and then plots them on a graph (see, for example, Figure 5.6).

The **AUC** (Area Under Curve) is the area under the ROC curve. This score shows how much a model can distinguish between classes. High AUC means that the model is better at predicting positive values as positives and negatives as negatives. The perfect model has an AUC score of 1, and a random prediction would have an AUC score of 0.5.

5.2.3 F1 score

Defining the F1 score requires another metric that is based on the confusion matrix.

Precision or Positive Predictive Value (PPV): shows the percentage of relevant results.

$$PPV = precision = \frac{TP}{TP+FP}$$

The F1 score is a weighted average of precision and recall. A higher F1 score means that our model performs better.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

5.2.4 Cross Validation

The previously mentioned metrics are not useful on their own. We need a way to validate our model. I decided to use cross-validation (CV) to estimate how the different models would perform on an unknown dataset in practice. Different cross-validation methods exist: train-test split, KFold etc.. In this research, I used the train-test split method, where we randomly split our dataset into training and test sets. We train our models in the training dataset and use the test set to validate the performance. The split ratio that I used is 70:30, where 70% is the training dataset. As a metric for cross-validation, I used the three previously discussed methods: accuracy, AUC, and F1 score.

5.3 Creating classifiers

I tried many different classifiers to find what fits best to the problem of classifying SQL queries as long or short-running. In this section, I present only those that performed the best or are needed to understand the concept of a more complex algorithm. First, I start with shortly introducing the algorithm, continuing with the evaluation of each model. This part is not intended to describe the models in detail, it only gives a high-level overview to the reader. At the end of the section, I summarize the models and choose the best-performing for future improvements.

However, one thing is still not clarified: what do we consider as a slow query? It is a fully workload-dependent question. To get a model that can classify correctly, we need enough positive (in our case, slow means positive) samples. Thus, I decided to label the slowest 30% of our queries as slow. In the collected dataset, this resulted in every query executed for more than 8 seconds to be slow. As a simplification, I only considered one of the execution times of a query: execution time on Hive.

5.3.1 K-Nearest Neighbour (KNN)

This algorithm assumes that similar things are close to each other. It first calculates the distance (in case of numeric variables most commonly Euclidean Distance) of each object and defines the k nearest ones. The class of a given object will be the most frequent class among its neighbors. In this example, I used five as a value for k.

Executing the evaluation method for each metrics, I got the following results using the KNN algorithm:

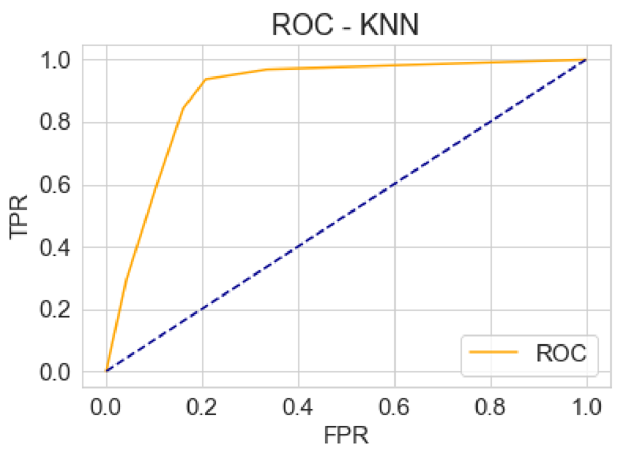


Figure 5.6: ROC curve of the KNN model

Accuracy	**AUC**	**F1**
0.835396	0.812717	0.737993

5.3.2 Bernoulli Bayes [12]

To understand the Bernoulli Bayes model, first we need the basic concept of the Naive Bayes algorithm. This model is based on the Bayes' theorem. Given a target variable y and the feature variables $\boldsymbol{X} = (x_1, x_2, ..., x_n)$

$$P(y \mid \boldsymbol{X}) = \frac{P(y)P(\boldsymbol{X}|y)}{P(\boldsymbol{X})}$$

Here, given a that $(x_1, x_2, ..., x_n)$ occured we can find the probability of y occured $(y = 1)$. If we substitute into $\boldsymbol{X}$ and apply the probability chain rule

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1|y)P(x_2|y)...P(x_n|y)}{P(x_1)P(x_2)...P(x_n)}$$

Based on our training dataset, we can substitute every variable on the right side. Now we have the probability for all possible (in our case 1 or 0) y if $(x_1, x_2, ..., x_n)$ occured. We just have to get the value with the maximum probability and assign the object to that class.

The Bernoulli Bayes model implements the Naive Bayes model, with a restriction that each feature is a binary variable. This requires the samples to be in binary format. In our case, this requirement is not met, so before applying this algorithm, the model will binarize the input. Let us see how this model performs with our dataset.

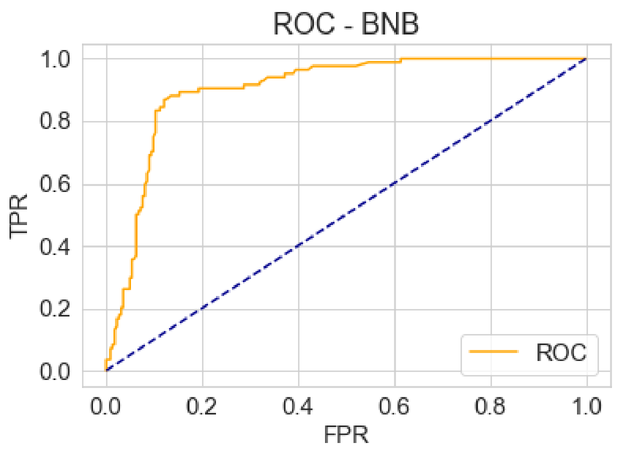


Figure 5.7: ROC curve of the Bernoulli Bayes model

Accuracy	**AUC**	**F1**
0.851466	0.862649	0.779389

5.3.3 Decision Tree

A Decision Tree classifier predicts the target variable based on simple rules. These rules are deduced from the feature variables in the dataset. This type of model can be visualized, and people can easily understand and interpret it.

The main steps of the algorithm: Select an attribute that splits the records. This attribute will be a decision node that breaks the dataset into smaller subsets. Continue with building the tree by repeating this process recursively until either the chosen condition does not split the records further or there is no more remaining attribute, or there are no more instances to split.

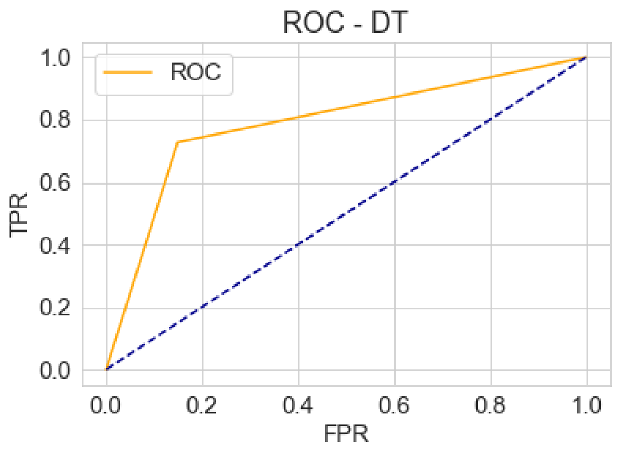


Figure 5.8: ROC curve of the Decision Tree model

Accuracy	**AUC**	**F1**
0.824864	0.788915	0.709625

As we can see, this simple Decision Tree model does not perform as well as the previous ones. However, many more complex algorithms use Decision Trees inside, so understanding the concept behind them is necessary.

5.3.4 Random Forest

The Random Forest is an ensemble algorithm, meaning that it uses multiple algorithms to obtain a better performance. To be more specific, Random Forest is a bagging method, so every model in the ensemble is trained with a random training dataset and votes with equal weight. This model contains a large number of individual decision trees that are choosing a class individually and independently for the sample. The class with the most votes will be the prediction of the model.

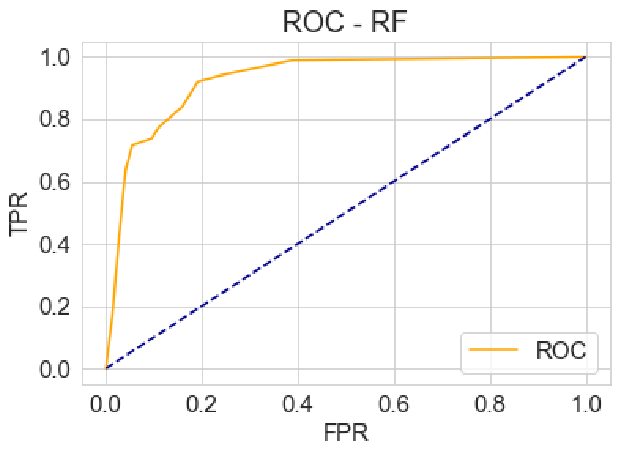


Figure 5.9: ROC curve of the Random Forest model

Accuracy	AUC	F1
0.864061	0.839039	0.774908

5.3.5 Gradient Boost [3]

Boosting is also an ensemble technique but opposed to bagging, the models inside are made sequentially, improving the previous model by correcting its error. The basic concept of the algorithm:

1. The algorithm fits a model to the data (in our case Decision Tree)
2. Calculates the classification error (how good the current model is at classifying slow queries)
3. Fit a new model to the error as the target variable
4. Create a new model where we add the predicted errors to our original model
5. Repeat step 2-4 until our model is not overfitting: we can check it by cross-validation

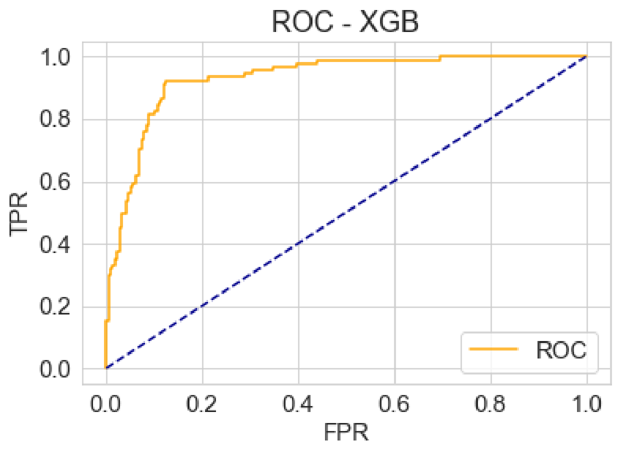


Figure 5.10: ROC curve of the Gradient Boost model

Accuracy	**AUC**	**F1**
0.878610	0.863896	0.807982

5.3.6 Summary

This section presented the top 5 performing classification algorithms I used. Now it is time to select the best model. The metrics that I used are depending on how we split our data to train and test sets. Thus, I repeated the model training and prediction several times to get a clearer picture of the performances. The following table contains the means of each metrics collected for 30 different train-test split. Means are rarely enough to represent a dataset, so the boxplot intends to describe the different classification models accurately.

model	auc	accuracy	f1
KNN	0.815070	0.836808	0.740039
BNB	0.866732	0.856352	0.786683
DT	0.794773	0.829642	0.711240
RF	0.832578	0.862324	0.770319
GBDT	0.867305	0.882302	0.809130

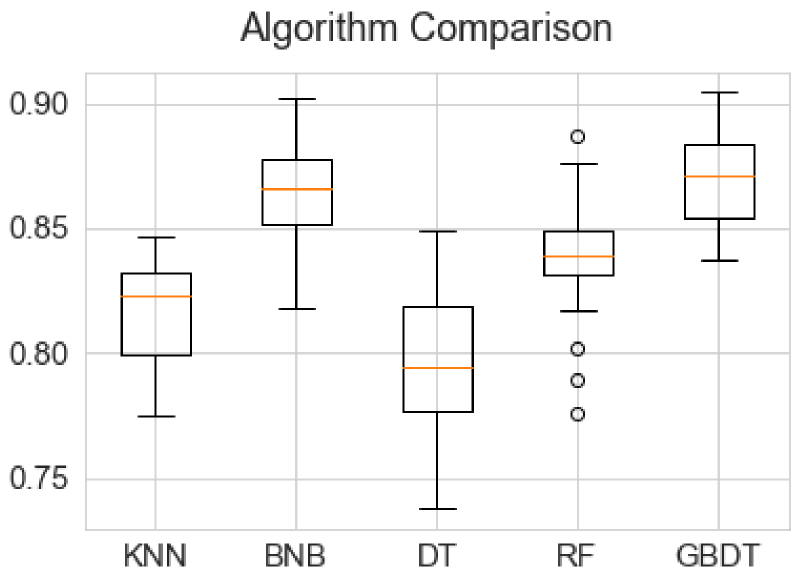


Figure 5.11: Model performance plot

We can identify that the two most accurate models are the Bernoulli Naive Bayes and the Gradient Boost Decision Tree models. They perform similarly in the AUC score. However, inspecting the other metrics, we can see that in terms of accuracy and f1 score, the Gradient Boosting could reach a higher value. Based on the metrics, I decided to continue using the Gradient Boosting Decision Tree algorithm since it seems the best fit for predicting short and long-running queries.

5.4 Feature elimination technices

The Gradient Boost Decision Tree model proved to be the best performing in all the metrics. Our dataset has many features; however, most probably, not all of them have an impact on the output variable. These irrelevant features can make our model less accurate and can cause overfitting, so selecting a subset of relevant features can improve our classifier. In the data science world, different feature selection techniques exist. In this section, I describe the best performing and show how it impacts the accuracy of predicting short and long-running queries.

5.4.1 Recursive feature elimination - RFE

Recursive feature elimination requires the number of features as an input. It fits the given model and removes the weakest feature recursively until we reach the specified number of features. The first question that comes to mind: what is the weakest feature? It depends on the model. The Gradient Boosting Decision Tree in the XGBoost package uses the F-score metric that sums up how many times a feature is split on. I experimented with keeping a different number of features.

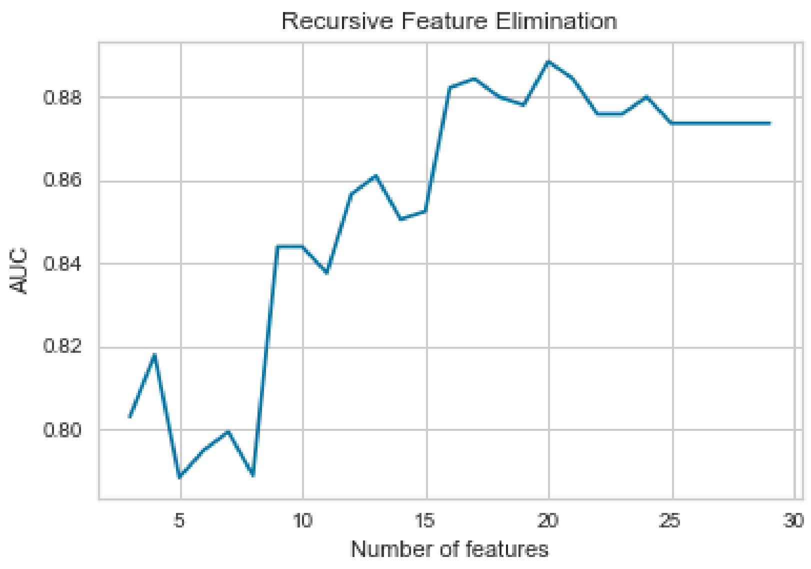


Figure 5.12: Recursive Feature Elimination

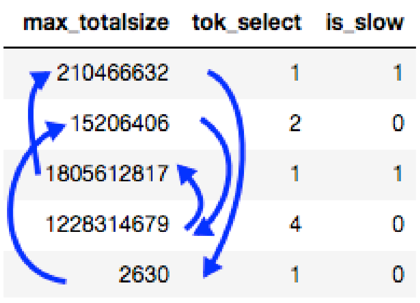
The graph illustrates how changing the number of features in the input of the RFE algorithm affects the AUC score. We can see that the model performed the best with 20 features, resulted in a 0.8884 AUC score. Increasing the number of features up to 20 does not improve the model instead, making it more complicated and even less accurate. It is worth looking into the importance of each selected feature.

5.4.2 Permutation Importances [6]

After selecting the best features using RFE, I wanted to find out what features have the most impact on our prediction. To accomplish this, I used a different approach, called Permutation Importance.

It first gets the trained model and selects a single column in it. The algorithm shuffles the values in the selected columns and tries to make a prediction using the changed dataset. Using the new predictions and the true classes, it calculates the loss that the shuffling caused compared to the original model. This performance loss will measure the importance of a feature.

For a better understanding, let us examine the following simplified example:



max_totalsize	tok_select	is_slow
210466632	1	1
15206406	2	0
1805612817	1	1
1228314679	4	0
2630	1	0

Figure 5.13: Permutation Importance Example

The chosen column is the *max_totalsize*, which is the size of the biggest table in the query in bytes. The algorithm shuffles the values in the *max_totalsize*, and the other features remain untouched. Then the given GBDT model will be used to predict if the query is slow or fast. The performance of the model is calculated (e.g. using AUC score) and compared to the performance of the original model (without shuffling *max_totalsize*). The loss will be the difference between these two values. Since *max_totalsize* is an important feature, the loss will be high, so this feature is important in our model => the feature importance of *max_totalsize* will be high.

The following bar chart shows the feature importances (weights) of the 20 features selected by the RFE algorithm.

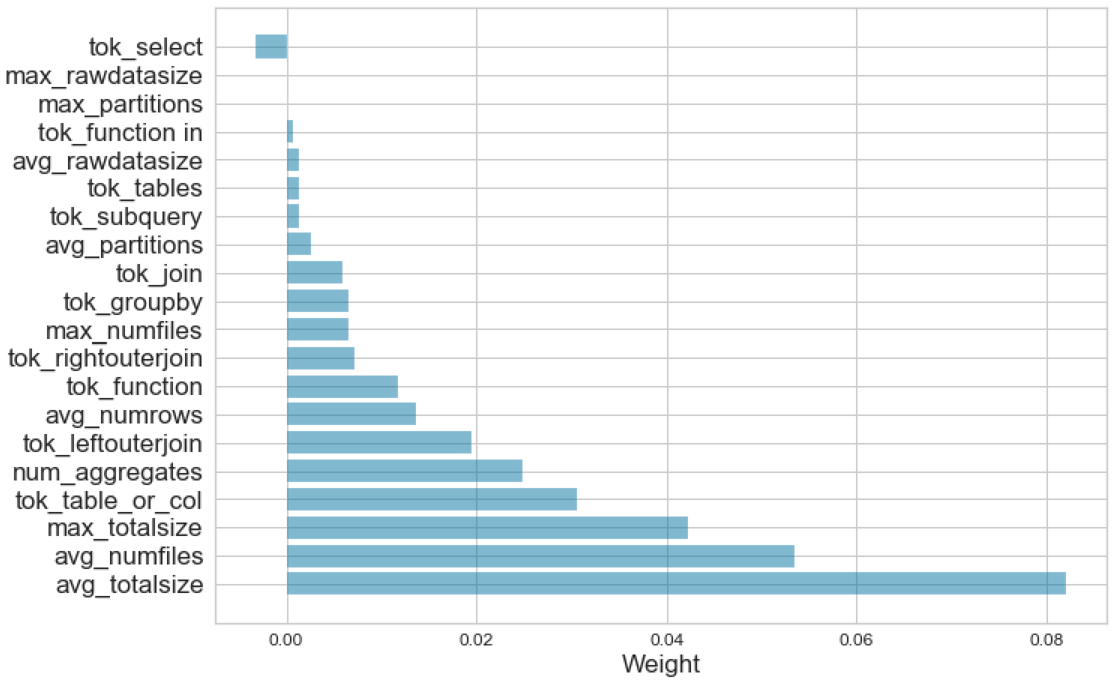


Figure 5.14: Permutation Feature Importances

We can see features with 0 or negative permutation importance scores. A negative score means that after shuffling, the model gave a more accurate prediction. Zero permutation importance means that it does not matter if we shuffle these features or not; the accuracy remains the same. Considering these, we can easily remove the features with 0 or less importance value without any decrease in accuracy. Dropping *tok_select*, *max_rawdatasize*, *max_partitions* did not influence our model, we got the same AUC score with 17 features as with 20.

5.5 Understanding the model

Besides improving the AUC score of our model, having less feature also benefits in interpreting it. Visualization is not an option since the decision trees are deep and challenging to understand manually. However, the domain is well known, so looking into the previous feature importance plot helps us understand what causes a query to be slow.

The most important feature is the *totalsize*. Both the maximum and average of the table total sizes in a query highly increase the execution time in Hive. The number of rows and files also correlates significantly with the execution time. As expected, the number of joins and aggregate functions affect runtime. Nothing surprising so far. However, the number of subqueries does not seem to influence the execution time. This does not illustrate reality. However, it can be explained easily: our machine learning model is as good as our data. Since the collection of execution times is difficult, restricting the number of subqueries was necessary to prevent the generation of too complex queries resulting in a huge execution time and less data to be collected in a given period. This results that there is not enough sample that has multiple subqueries inside.

To sum it up, we can predict if a query will run for a short or long time with high accuracy, without compiling it. We only need to inspect the query using the parser of Hive and getting some metadata from Hive MetaStore. However, is it useful to predict this in milliseconds? The evaluation chapter will answer this, but first, I will try to fit a model to our original question: run our query on Hive or Impala?

5.6 Predicting where to run a query

The data preparation task for predicting if a query will run faster on Hive or Impala was almost the same as predicting if a query will be long- or short-running. But instead of removing queries that differ highly in execution time in Hive and Impala, I keep them since these contain useful information for the prediction.

The Gradient Boosting Decision Tree as a classifier proved to be the best performing in this scenario, as well. The following graph shows the result of recursive feature elimination with a different number of features.

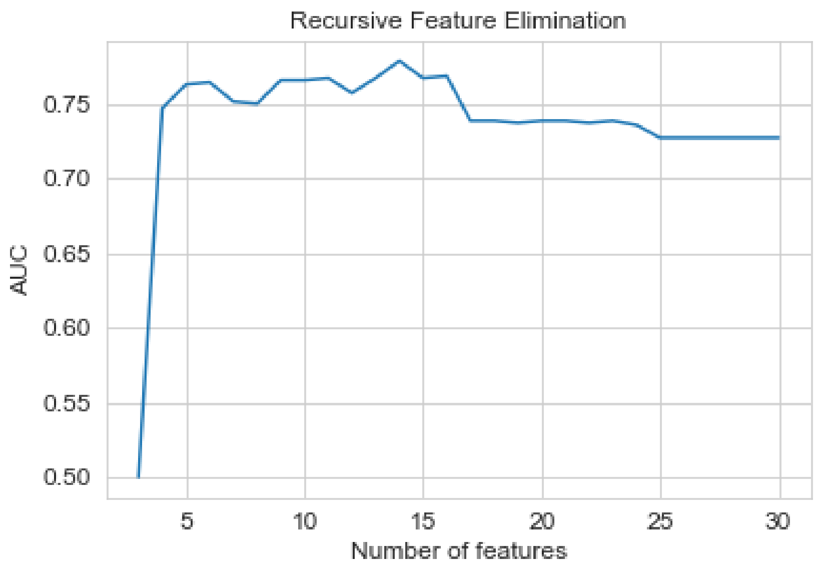


Figure 5.15: Recursive Feature Elimination results

As shown, we can reach an AUC score of 0.78 using the best 14 features. Although this looks promising, I looked into how much we gain if we route the queries to the predicted execution engine. The amount of seconds we would win with the routing is comparable with the increase of complexity if we introduce a new layer. What could be the explanation for this?

Even though we can predict where a query will run faster with acceptable accuracy, many of them are running for only milliseconds. The randomness of the queries causes that many of them do not have a resultset; hence, an optimization algorithm in Hive or Impala can recognize and return immediately.

Our machine learning model is as good as the data that we used for training it. While randomly generated queries are enough to recognize patterns when a query is slow, it is not enough for a more sophisticated classification. To create a meaningful model, I need meaningful queries. I will introduce a solution in the final Future Work chapter that we can use to overcome this issue and have data and queries that simulate a real environment. The rest of the evaluation chapter focuses on the scheduling of the queries based on prediction given by QuDi. The result accomplished with random queries proved that the model could recognize patterns for the given random use-case, so continuing with a real-world example can also be beneficial.

Chapter 6

Evaluation

We have a model that can classify if a query is slow- or long-running in milliseconds without compiling it. However, is it beneficial in a real industrial environment? In this chapter, I simulated a workload, where I used a Shortest Job First (SJF) scheduling algorithm based on the prediction given by the model. Followed by examining QuDi in the aspect of scalability and robustness.

6.1 Scheduling queries

6.1.1 Problem with current scheduling

Currently, both execution engine only supports First Come First Served (FIFO) scheduling algorithm. That means that the query that came first will execute first. In many scenarios, this results in a huge waiting and response time. Scheduling queries according to their execution time would result in an improvement of the throughput of our data warehouse system.

To understand the issue, consider the following example:

The number of concurrent queries in our system is set to 2. What if the following five queries come in this order having only milliseconds of delay between them. The unit these queries run can be anything between seconds to hours; this depends on the workload.

- Q1: query running for 6 unit
- Q2: query running for 7 unit
- Q3: query running for 2 unit
- Q4: query running for 1 unit
- Q5: query running for 1 unit

Let us say that queries below an execution time of 3 units are short queries. This means Q3, Q4, and Q5 are considered short. What happens currently if these queries hit Hive or Impala?

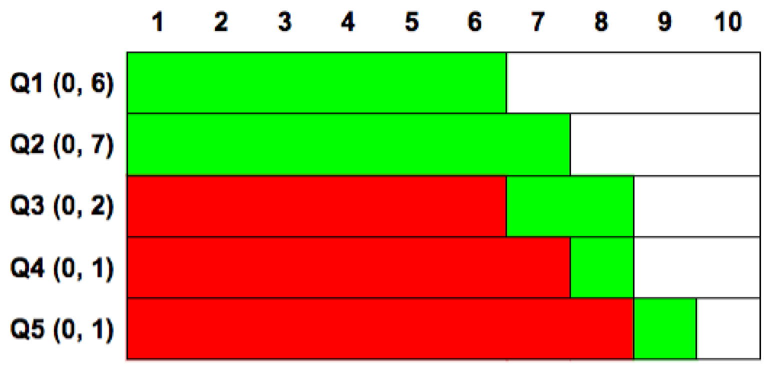


Figure 6.1: Result with FIFO scheduling

The above Gantt-chart represents the queries, showing the waiting time with red, and the runtime with green. Since these queries arrived around the same time and the delay between them is negligible to the execution time, we can consider them arriving in the 0th unit. The first two queries can run concurrently, while the last arriving three will wait for execution in the queue.

Calculating the waiting time (WT) for the queries:

$$WT(Q_1) = 0, WT(Q_2) = 0, WT(Q_3) = 6, WT(Q_4) = 7, WT(Q_5) = 8$$

The Total Waiting Time:

$$TWT = \sum_{i=1}^{5} WT(Q_i) = 21$$

The short queries had to wait for a long time to start executing. In a real scenario, these queries can be analytic queries, that were launched by data scientists, while the long-running ones can be daily reporting queries started automatically. Our users will wait for their simple, short queries to execute, but first, the reporting has to be done, causing a significant loss in productivity. Most of the time, it is expensive to increase the number of concurrent queries, since it requires a linear scaling of our resources. Increasing it without improving the hardware underneath can cause our data warehouse to crash. However, we could schedule the queries using a Shortest Job First (SJF) scheduler. The below chart shows the result of this scheduling.

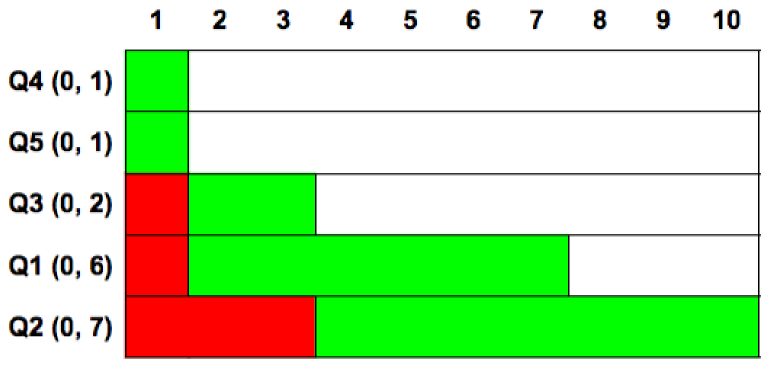


Figure 6.2: Result with SJF scheduling

$$WT(Q_1) = 1, WT(Q_2) = 3, WT(Q_3) = 1, WT(Q_4) = 0, WT(Q_5) = 0$$

$$TWT = \sum_{i=1}^{5} WT(Q_i) = 5$$

The gain in Total Waiting Time is 16 units, and we could increase the throughput of our system easily: e.g. in 1 unit, we could execute two queries with SJF, while the throughput with using FIFO scheduling was 0.

Query scheduling is a fundamental problem in data warehouse systems. Hence many algorithms were developed until now (etc. Distribution-Based Query Scheduling [4]). The efficient utilization of our computing resources can save millions of dollars to customers with a large data warehouse system. The ideal would be to know the actual runtime for every query in advance of its execution. However, estimating the execution time of queries is a difficult problem, as query running time is a complex function of the query itself, the data that it operates on, and the environment in which it executes. However, simply knowing that the query will be short- or long-running can result in a significant improvement in waiting and average query completion time. Fortunately, I have a model that can predict with 90% of accuracy that a given query will be short or long-running.

6.1.2 Scheduling queries using the prediction

In the following simulation, I selected ten random queries from the dataset collected by QuDi and used Apache Hive to simulate a concurrent workload, with the number of allowed concurrent queries set to 2. First, I ran the queries serially to identify the response time individually.

Query	Response time (ms)
Q1	27020
Q2	7608
Q3	6266
Q4	6200
Q5	6460
Q6	6200
Q7	22463
Q8	30042
Q9	40988
Q10	16435

Table 6.1: Response times of the queries if ran individually

We can identify a couple of long queries: Q9, Q8, Q7, Q1. Examine the worst scenario, when these queries start executing in ascending order of their execution time. I ran this simulation through QuDi and measured the response times of each query using the environment mentioned above (2 concurrent queries allowed in Hive). After that, I used the previously built prediction model to decide whether a query is short- or long-running. Using this prediction, I sorted the queries in the following way: run the short queries first, in any order, and then run the long-running ones last. The following chart presents the response time for each query.

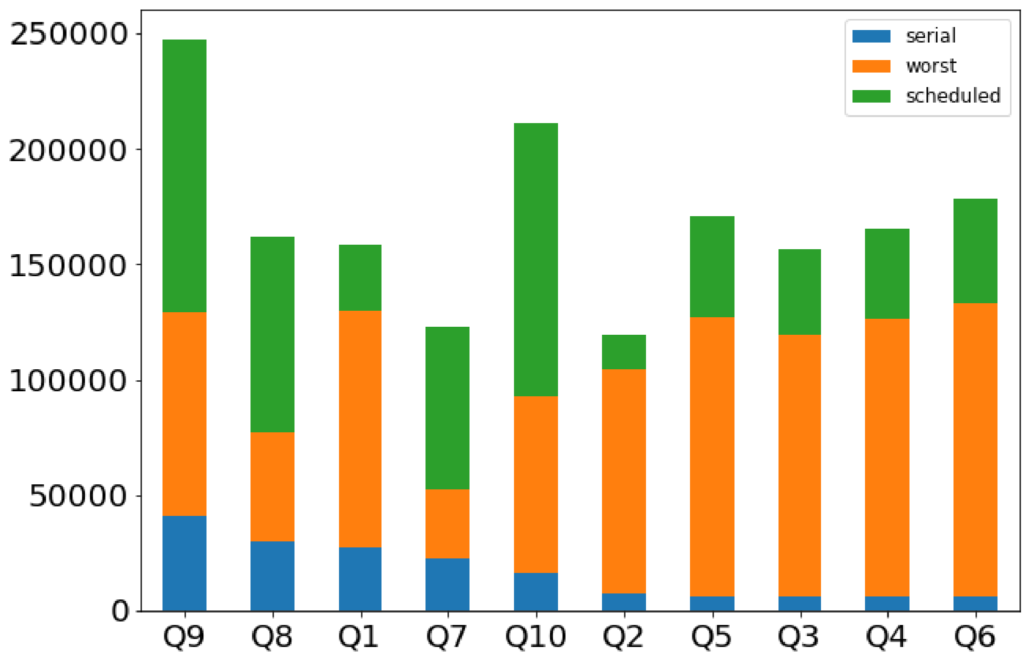


Figure 6.3: Response times in milliseconds using different scheduling

In this diagram, we can see the gain for every query individually. For example, looking at query 2, we can identify that the response time without scheduling is six times longer than with using the prediction and schedule based on that. However, it is hard to identify the overall result based on this diagram.

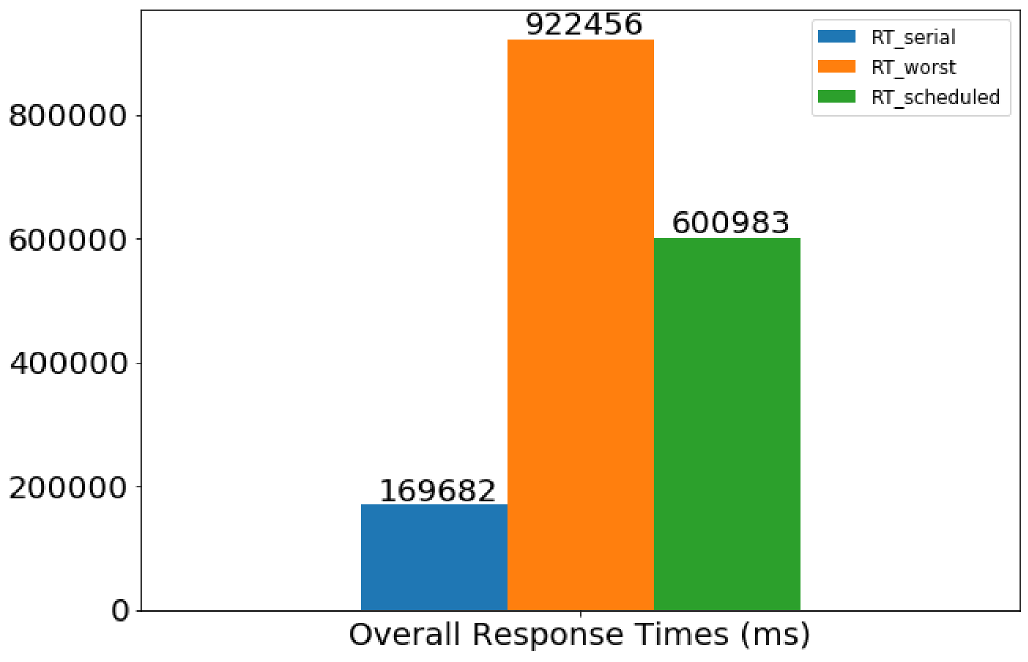


Figure 6.4: Overall response times in milliseconds using different scheduling

Figure 6.4 shows the total response time for the 10 queries. Even in this simple workload, the total response time decreased with 320 seconds (35%), which is notable. Looking at the throughput of our system: without scheduling, Hive could run 5 queries in 100 seconds. Using the prediction based SJF scheduling, Hive could execute 8 queries in 100 ms. This means 60% increase in throughput for the sample workload. Even though for some queries QuDi predicted incorrectly, we can say that the overall result is promising using the prediction. I ran several other workloads, including random scheduling of the queries, but in all cases, we could observe a significant improvement in overall response time and throughput.

In a real workload, the simulated scenario can happen easily. However, what about the starvation of long-running queries? The main drawback of the demonstrated simple scheduling (SJF) is that, in some cases, long queries could never execute. However, we can handle this issue quickly. Given that a long query is waiting for a certain amount of time, we can increase its priority and execute it immediately. Another solution could be to allocate some bandwidth for only short running queries in Hive: e.g. if we can run 10 concurrent queries, always keep 2 for fast running ones. The ratio of the two queues can be configured based on the workload since we can identify how much of our queries are short- and long-running.

6.2 Scalability

Until now, I did not write about the scalability of QuDi. Since both Hive and Impala are scalable up to petabytes of data and hundreds of users running concurrent queries, a proxy layer above them cannot be a bottleneck. I had to keep in mind from the beginning that in production, QuDi will be used in a heavily concurrent environment, probably running on a different machine than Hive or Impala. For communicating with clients, I used a thread pool server, which spawns a worker thread for every connection made to QuDi. However, creating a new thread for every query is not acceptable, so I used sessions for solving that. For every user, QuDi will allocate one thread from the pool, creating a session for that connection. This session will be kept and used for running several queries until the connection is closed. This will allow QuDi to scale vertically. Horizontal scalability is not needed since QuDi is not doing any computation that would require huge resources.

We can consider the data collection ran through QuDi a scalability test since it contained thousands of random queries often run in a parallel manner. QuDi was able to serve multiple clients concurrently, routing their queries to Hive and Impala. However, in the future, before putting QuDi in production, more scalability tests and benchmarks are needed to prove that QuDi does not introduce a performance bottleneck and can scale with increasing the workload.

6.3 Robustness

QuDi is working with SQL systems that slightly differ in SQL grammar. However, it is intended to route the queries according to the predicted optimal engine. But what if the given query can only run in one of the engines? At the learning phase, QuDi can identify queries that failed during the execution in one of the engines, and mark those with a huge execution time on the engine that it is failed. If we have a significant amount of samples that failed in one engine, the model can identify those in advance so that it could distribute the query accordingly.

QuDi can also handle erroneous inputs. It is working like a transparent proxy, so the users cannot identify that a new layer was introduced. If a user sends a syntactically or semantically incorrect queries, QuDi returns the exact error that the execution engine gave.

6.4 Security

QuDi is working with distributed systems that communicate with Hive MetaStore, Hive, Impala, and the clients using a network. It is important to work in a secure environment, as well. To achieve this, I implemented Kerberos authentication methods that Hive and Impala support to identify the users of our system. QuDi also supports SSL encrypted connections in order to improve the security of the system.

Chapter 7

Conclusion

In this paper, I introduced the two most commonly used open-source big data SQL engines and presented the problem that the users have to face: where to run a query to get the results as soon as possible? Usually, Hive and Impala are both running on the cluster, and users need to decide themselves. However, this is extremely difficult since query execution time is a complex function of the query itself, the data that it operates on, the environment in which it executes. Until now, there was no existing solution to help users with this complex problem. With QuDi, my original goal was that using a machine learning model, predict where a given query will execute faster.

During my research, I built a system that can analyze queries and collects more than 40 different features from them, including query and metadata features. I examined the possibilities to collect data that can be used for building a machine learning model. I decided to use random queries to build a prototype model. I found two existing solutions for generating random queries: SQLSmith and Discrepancy Searcher. I used the latter, generated thousands of queries, and ran through QuDi, analyzing each query, and measuring their execution time in both engines.

Having thousands of samples allowed me to start building a model that can predict where to run a query. However, I decided to start with a simpler task: predict if a query is short or long-running. This was beneficial in two aspects: the data preparation part of the machine learning was almost the same as for the original problem, and knowing if a query will run for a longer time before compilation can be used for scheduling queries in a heavily concurrent environment.

The data science work started with cleaning the data, dropping the outliers, and elimination features that do not influence the class of a given query. I examined multiple classifiers and decided to use a Gradient Boosting Decision Tree as the model for predicting if a query is short or long-running. For determining which model to use, I calculated several metrics that describe how a particular model performs. With the selected model, I reached an **AUC score of 0.89**.

Achieving this result, I continued with the original problem: predict where a query will execute faster. This classification problem is more complex, and even though I could achieve an AUC score of 0.77, the gain in execution time is not significant since some of the queries made the model overfitting and identifying some patterns that do not exist in reality. The randomness of the queries caused this, so improving the solution will require to use a real-world workload.

The results with predicting if a query is slow or fast were promising. I did a simulation where I restricted the number of concurrent queries and ran ten randomly selected at the same time. I implemented a simple scheduling based on the prediction given by QuDi: classify the queries if they are slow or fast, and using this, run the faster queries first. Comparing the overall response times in the worst case (when the queries arrived in decreasing order of their execution time) and in the scheduled workload, I reached **35% decrease of overall response time** and **increased the throughput of Hive with 60%** on the simulated workload.

7.1 Future work

In the near future, I planned several improvements. First, to continue with predicting where a query will run faster, we need a real workload with data and queries. Although customers will not provide their data, they are eager to publish queries and metadata about their data helping developers to run scale tests and benchmarks. This includes information like table schema, how they are partitioned etc.. Tools already exist to generate databases and tables using their metadata. This will enable us to run queries that are used in the industry and simulate a real workload.

The great results of a simple scheduling algorithm influenced to continue the research in this area. I am currently progressing with the implementation of a type of Shortest Query First scheduler inside QuDi. The following sequence diagram illustrates how it will work. The users write their queries in their favorite client and send them to Hive or Impala (the solution can work with both since their interface is identical). QuDi analyzes the query to get the most important features discussed in the Machine Learning chapter. Using the model, QuDi will predict if it is short- or long-running and place it in a queue to be executed. The scheduler will prefer executing short queries first, considering not to starve the long-running ones. This will improve the throughput of our system and decrease the overall response time.

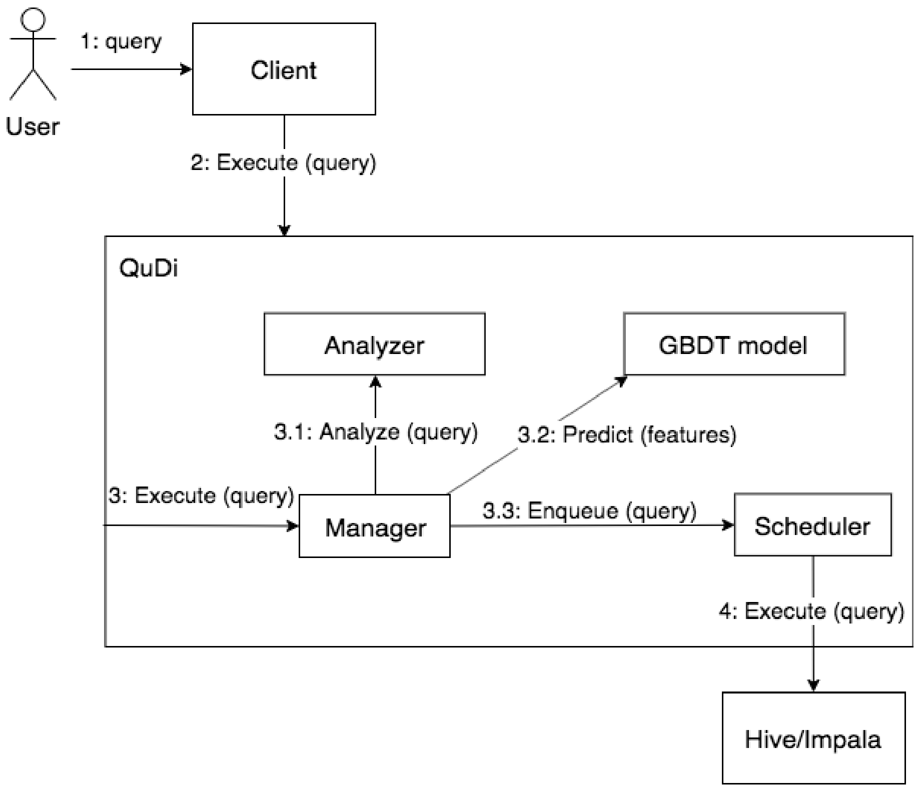


Figure 7.1: Scheduler inside QuDi

At the end of the year, I plan to present the project to both open-source communities: Impala and Hive. Their acceptance and opinion are important to put QuDi in production later, make my project open-source and find developers who would help to improve the system.

Acknowledgements

I want to thank Cloudera Hungary for providing resources for my research, and the Budapest Hive Team for helping to solve technical challenges and answering all my questions.

The results presented in this work were established in the framework of the professional community of Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the "BME Balatonfüred Knowledge Center", supported by EFOP 4.2.1-16-2017-00021.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Bibliography

[1] Andrew Sears. Hive llap. https://cwiki.apache.org/confluence/display/Hive/LLAP. [Accessed Oct. 3, 2019].

[2] Apache Impala community. Discrepancy searcher. https://github.com/apache/impala/tree/master/tests/comparison. [Accessed Oct. 8, 2019].

[3] Ben Gorman. Gradient boosting. http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/. [Accessed Oct. 17, 2019].

[4] Yun Chi, Hakan Hacigümüs, Wang-Pin Hsiung, and Jeffrey F. Naughton. Distribution-based query scheduling. *PVLDB*, 6:673–684, 2013.

[5] Cloudera Inc. Impala. https://docs.cloudera.com/documentation/enterprise/5-8-x/topics/impala_concepts.html. [Accessed Oct. 4, 2019].

[6] Dan Becker. Permutation importance. https://www.kaggle.com/dansbecker/permutation-importance. [Accessed Oct. 21, 2019].

[7] Gunther Hagleitner. Hive on tez. https://cwiki.apache.org/confluence/display/Hive/Hive+on+Tez. [Accessed Oct. 3, 2019].

[8] Jeremy Kun. Mapreduce formalism. https://jeremykun.com/tag/map-reduce/. [Accessed Oct. 2, 2019].

[9] Matt Jibson. Sqlsmith: randomized sql testing. https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/. [Accessed Oct. 7, 2019].

[10] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164217.

[11] Sarang Narkhede. Understanding auc-roc curve. https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5. [Accessed Oct. 16, 2019].

[12] Scikit Learn. Bayesian classifiers. https://scikit-learn.org/stable/modules/naive_bayes.html, . [Accessed Oct. 17, 2019].

[13] Scikit Learn. Model evaluation. `https://scikit-learn.org/stable/modules/model_evaluation.html`, . [Accessed Oct. 16, 2019].

[14] Sergey Shelukhin. Hadoop summit - hive llap. `https://www.slideshare.net/Hadoop_Summit/llap-longlived-execution-in-hive`. [Accessed Oct. 3, 2019].

[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. DOI: `10.1109/MSST.2010.5496972`. URL `http://dx.doi.org/10.1109/MSST.2010.5496972`.

[16] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop, 01 2010.