BDAPRO Report: Mediatorless Cross-Database Query Execution

Jan Vincent Hanack
Technical University of Berlin
Berlin, Germany
j.hanack@campus.tu-berlin.de

ACM Reference Format:
Jan Vincent Hanack. 2022. BDAPRO Report: Mediatorless Cross-Database Query Execution. In *Proceedings of BDAPRO (BDAPRO WiSe 21/22).* ACM, Berlin , BER, GER, 6 pages.

BDAPRO Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jhanack/BDAPRO_Project_Course.

1 INTRODUCTION

With the degree of globalization and digitalization increasing faster than ever before, and data becoming bigger, it becomes increasingly important to query, analyze and monetize data in the most efficient way possible.

Nowadays companies often use data federations, which means they have different kinds of heterogenous Database Management systems that are ideally supposed to work together seamlessly.

BDAPRO WiSe 21/22, March 2022, Berlin, Germany

ACM ISBN …$

This is needed for example when different companies are collaborating and sharing data between each other or even inside of one company that for example is present in multiple countries and wants to analyze all of their data from its different branches. Ideally, in the end a user can run a single query without having to care about whether it runs on e.g. PostgreSQL or MariaDB or Hive.

There are many real-life examples for this, and there are many companies offering their data for example on the snowflake data marketplace, but some use cases could be insurers that access live weather data for different regions, restaurant businesses accessing population and location data in order to expand their business, open banking consumer data or healthcare professionals accessing and sharing clinical trial data.

Another aspect of this is Data protection, because if we assume the ideal case where data is only being processed on the respective DBMS itself and only the query result would be shared with the user or other databases, the data would still be protected, which allows for more cooperation while also maintaining data compliance.

We introduce XDBX, an efficient system to query distributed data placed in different Database Management Systems without transferring the data to a central mediator.

2 PROBLEM

Currently most systems that are widely used, like for example Spark, follow a traditional Mediator Wrapper approach, where data is processed by a

centralized mediator component. While the workers themselves can be distributed, like for example in Spark, they are still an extra layer that data has to be transferred to, when it is loaded e.g. into Sparks Hadoop File System for processing.

Also, all the data from a relevant table has to be transferred, even though there might only be a single record filtered out, and additionally no indices can be utilized like for processing inside of the DBMS.

I.e. there is basically an extra step creating expensive overhead and increased latency, which becomes more the bigger the geographical distance between the nodes is and which would not be the case if data was processed directly on the connected database nodes and transferred only between them. While for Join operators, it cannot be avoided to transfer data between the nodes, for others, like an aggregation operator, transferring the data is an unnecessary step and also complicates data protection.

Since this project is supposed to be a part of DIMAs Agora project, this is even more of a problem, as Agora has a rather decentralized and global scope for using Big Data for innovative research or using querying data for different kinds of machine learning frameworks other than Spark.

This is also why we changed the original project approach, which was to use the Actor Model of Akka. Since an Akka Node always requires an active parent node, for a system like Agora with possibly many users, this would imply spawning and killing new instances of Akka nodes as part of the Database node for every user and having multiple Akka actors accessing the Database at the same time.

3 GOAL

The goal of this project was to develop an approach without a mediator, where as little data as possible is copied and transferred to a mediator or between Databases, or where no data at all is transferred, at least for e.g. aggregation operators. Furthermore, ideally we want interoperability with different DBMSs, so that in the end a user can run a single query without having to care about whether it runs on e.g PostgreSQL or MariaDB or Hive.

The goal is to create a more efficient system as part of the Agora project, that offers a similar functionality as similar systems from bigger industry players like AWS Data Exchange or Snowflake which offers a quite wide ranging system including Snowflake Data Sharing. These systems focus especially on data Ecosystems, where companies can access data and share data with other companies by simply accessing the data from their own cloud databases.

Snowflake for example even offers access control management so that data protection can be achieved by giving different users or different joint ventures different access rights to your data.

However our system, in the end should be used as an extension of the Agora project and use the output of an execution planner, which optimizes queries over different Database Management systems.

4 SOLUTION - XDBX

In this project, we developed a mediatorless approach for Query Execution that also allows for interoperability between different Database Systems, which can also be distributed over different nodes and is designed to let the DBMS-Nodes communicate directly with each other.

To achieve this, we use the SQL commands create foreign table and create view, so that a database can access data by accessing its own foreign table, which then points to a table or a view in a different database.

A view can be seen as a smaller part of a query that is executed as soon as the view is created and where the results can be accessed like from a normal table. Therefore the data transfer between the Databases will be initialized by one database querying over a foreign table that points to another DB, and possibly so on.

The details about which foreign tables and views to create we get from Agoras execution planner,

where the execution planner's output produces instructions specifying the databases and the tables that shall be used, and also includes query optimization that also considers how the data is distributed over the DBs.

Inter-Node/DB-Communication

For the connection and communication between the nodes respectively the different databases we use Protocol Buffers in combination with Google's open source Remote Procedure Call.

Protocol Buffers simply helps with the serialization respectively specifying the format in which messages and data are sent between the nodes, so that it is more efficient to read and write them, because they have a standardized format.

gRPC, which is used for example by Uber and Netflix, natively works with Protocol Buffers and is used for creating our server implementation for the different nodes, the communication with the clients for each node and the communication between the clients. gRPC allows for millions of RPCs per second and also offers some other features like authentication and timeouts.

Architecture

A general visualization example of our architecture is depicted in 1.

We implemented a slim server built with gRPC, which is present on each node.

Each node server also has a single Client, which is instantiated by the user and then is the interface for the communication of the user with the node.

The user can then call the methods createforeigtable and createview with the connection properties like database user and database name for the database, which are passed via the Client connection.

After the foreign tables and views are created by the user by directly connecting to the DB via the Client, the final query is then only sent to one DB. This last query is usually very short and in the format Select * From a table or from a view.

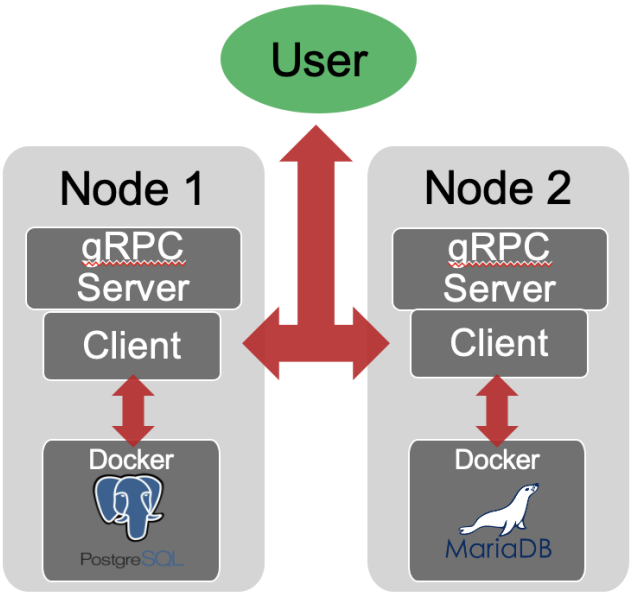

Figure 1: XDBX supports multiple nodes, which in turn can host one or multiple DBMSs.

The rest of the communication and possible intermediate data transfers are then happening between the DBs respectively the clients, and only the final query result is sent to the user node.

Here we have for example a Postgres and a MariaDB instance, but the system also supports Exasol and can be extended with more connectors to other DBs. In 1 the DBs are connected as Docker Containers, but they don't necessarily have to be inside of Docker Containers, however that's also how we tested and benchmarked the system.

Implementation

One part of the node server is a database manager, which maps one the database identifier, in this case the container name and the Database port to the local node port and two, the local node port to with the kind of Database.

The kind of Database is important, because there are several smaller differences in the syntax of the DBs; for example some use NUMERIC and some use DECIMAL as the column type. This is tackled by having a different connector class for every

Type of DB. When the columns from a foreign database are retrieved in order to create a foreign table on a database, the column types are then formatted to a standard format, which can then be formatted to the respective format of the database where the foreign table is created on.

For the system to work, we had to establish two connections, for which the user has to pass the connection details.

One connection is the connection between the user and the Node Client. This includes the server port and the server address, which is localhost in the case of our test and benchmark setting, as depicted in 2. XDBX also includes a user manager for this.

The second connection is the connection between the client and the Database, which uses JDBC or ODBC. In addition to the XDBX connection details that the user has to provide, to connect to the server, he or she now also has to provide the user and password for the actual database and the port that the DB is running on the node like in 3, in this case 8081. After that, the Databases can then communicate directly with each other by using the established connections.

5 EXPERIMENTS & BENCHMARKS

I order to test the system, I used the following experiment setup.

We initialized two node server and client instances on a single local machine, which mirrors the behaviour of two nodes running.

We tested a setup with multiple Postgres instances running in Docker containers, distributed over the nodes, which are accessed via the postgres JDBC client version 42.3.2. For the postgres containers, which are started via Docker Compose, we used the official PostgreSQL Docker image 12.10-bullseye and installed the Java 11 SDK, several of Linux packages, and the JDBC_FDW extension on top. While XDBX is designed to work with different DBMSs like MariaDB ad Exasol, it does so to a certain extent, but final testing in combination with Postgres Instances failed due to connector issues and could not be fixed so far.

For the testing as well as the benchmarks, we used the TPCH-Data and Query Set, which consists of artificial generated data and typical decision support queries and is commonly used to benchmarks query execution performance.

The Data is generated on a separated Docker container[1] and mounted on the database instances as volume *test-data*. All the data is copied and different tables queried based on the table distribution.

The TPC-H Data can be generated with different scale factors ranging from 1, which generates a dataset with 1 GB up to many terrabytes. We have tested and benchmarked XDBX with scale-factor 1 and TPC-H queries 3 and 10. For every query, we use a different table distribution over the databases, in order to make sure that there is the data-heavy transfer of joins between the server nodes included.

For benchmarking, we have compared XDBX with a SparkSQL setup and with Presto, that runs the same TPC-H queries over the same database instances, with the same table distribution. Spark and Presto are basically distributed processing systems, which do not natively use Foreign Tables and Views, but they also support connecting different kinds of Databases and therefore are the most related frameworks to compare XDBX with. The Spark benchmark is written in Scala, compiled with scala-sdk-2.12.9 and runs with a local master/worker setup. Presto was set up as a Docker container, joining the Docker-Network of the database instances, which then connects to the database instances. The connection details for both benchmarks read via separate .property files and both benchmarks use the Docker's container network for data transfer.

For Reproducibility, the Postgres JDBC driver might has to be added as a library to project's Gradle project and the ./Benchmarks directory has to

[1] See ./Docker/tpch_generator/Dockerfile in the project directory

```
new SimpleXDBConnection( address: "localhost", port: 7000, username: "max", authentication: "123");
```

Figure 2: XDBX implements a User <-> Node connection.

```
new PostgresInstance(clientFactory),
  dbAddress: "localhost",
  dbPort: 8081,
  dbUsername: "bdapro_user",
  dbPassword: "bdapro_password",
  database: "bdapro_database");
```

Figure 3: XDBX implements a Client <-> Database connection.

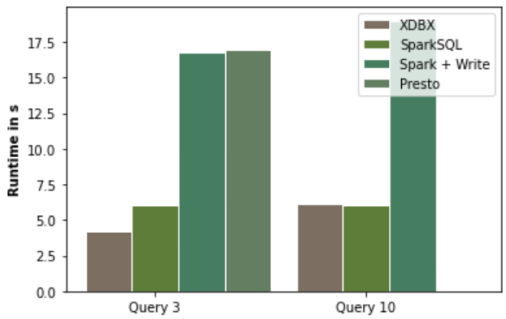

Figure 4: We compare TPC-H Query 3 and Query 10 with scalefactor = 1. XDBX in general outperforms Spark and Presto.

be imported to one's IDE as a separate project for optimal Scala compiling. While XDBX was tested with Java 11 and 15, the Benchmarks project, included in the project's source code only runs in Java 8, for compatibility with Scala 2.12.9.

6 EVALUATION

The results for Query 3 and Query 10 for scalefactor = 1 are shown in 4. In general our system XDBX performs better than Spark and Presto.

For query 3, our system takes 4.2 seconds and our Scala written Spark benchmark takes 6.0 seconds. For query 10, both perform the same.

Sparks as well as XDBX's execution time is less than 1 second, so only a fraction of the total time, indicating that the most runtime is used for setting up the spark job and loading the data into the HDFS file system, respectively setting up XDBX's server nodes and connections.

But there are few things to pay attention to when comparing the systems. For XDBX, the execution time s measures as the registration of the foreign tables and views, which is where the actual query computation takes place.

It is also important to mention that Spark doesn't write the files to disk, while our system creates views, that make the processed data constantly available.

If one includes the time that is needed to write the results to disk, Spark takes more than 15s (Query 3) respectively more than 17s (Query 10).

Presto performs the worst for Query 3 and notably takes significantly longer for Query 10, where we stopped measuring after 5 minutes.

In general, the query output for XDBX is in line with the sample output of the TPC-H specifications, i.e. XDBX is working.

It also outperforms Spark and Presto.

Here it is important to mention, that in the benchmarks no Indices were used yet, which would speed up runtime even more, which is a competitive advantage against Spark and Presto, since they cannot use Indices.

For future research, it will be important to measure how XDBX scales up with a higher scalefactor, where e.g. the setup cost of the spark job, including the workers, might become less relevant relative to the actual query runtime.

I general two things are important. First of all, it is important of course what kind of workload one works with. Spark might become more advantageous for CPU heavy workloads, where you would for example manipulate the data queried with UDFs.

However, for queries for example with a lot of filtering, a system like XDBX might be helpful, since one it could use Indices and two there is no unnecessary data copied to the Spark workers.

Also, XDBX is the better fit for slight data changes / manipulation where we want to write the data back into the database, because then one would actually save two data transfers in comparison to Spark.

Second, it is important to mention that for example this system and Spark do not necessarily rule each other out.

We mainly utilize foreign tables and views, however one could also use this system and then run a Spark job on top of it, reading only the more limited views into Sparks file system. In this case the foreign tables and views would act as a preprocessing step of a Spark job.

However, in practice on should always consider what queries are run often and distribute tables accordingly over the databases if possible, not adapt the queries to the distribution.