Information Systems, Analysis and Design
Class Projects (2024-25, Fall Semester)

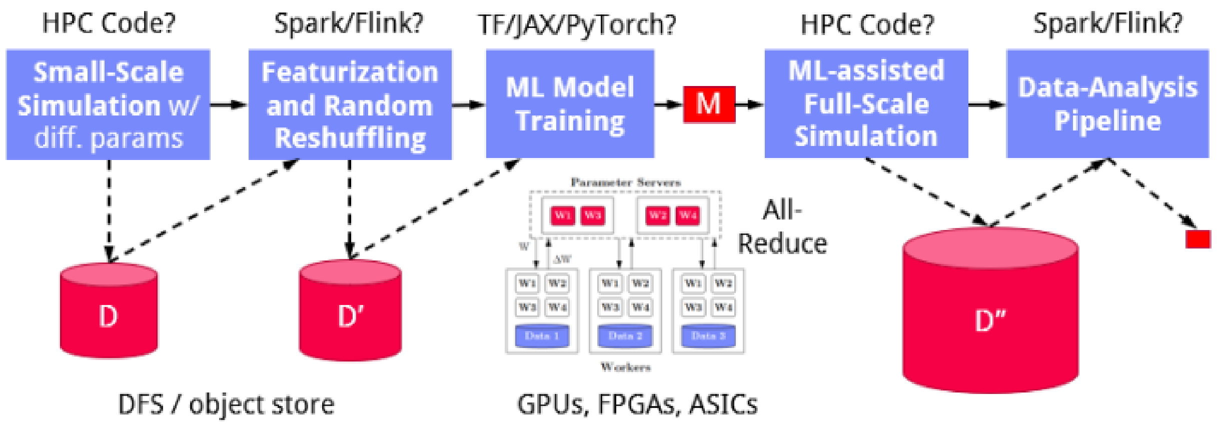
1. Daphne-project related **2**
Daphne1: Optimizing HDFS integration with in the DAPHNE Runtime 3
2. Comparison between Vector DataBases 4
Vec1: qdrant, milvus 5
Vec2: milvus, weaviate 5
Vec3: chroma, qdrant 5
3. Comparison between Python scaling frameworks for big data analysis and ML 6
Python1: Ray, Dask 7
Python2: Ray, Apache Spark 7
Python3: Ray, PyTorch 7
4. Distributed Execution of SQL Queries 8
SQL1: PostgreSQL, Cassandra, Kudu 9
SQL2: MySQL, MongoDB, Memory (main mem) 9
SQL3: MongoDB, Cassandra, PostgreSQL 9

1. Daphne-project related

The DAPHNE project aims to define and build an open and extensible system infrastructure for integrated data analysis pipelines, including data management and processing, high-performance computing (HPC), and machine learning (ML) training and scoring. This vision stems from several key observations in this research field:

1. Systems of these areas share many compilation and runtime techniques.
2. There is a trend towards complex data analysis pipelines that combine these systems.
3. The used, increasingly heterogeneous, hardware infrastructure converges as well.
4. Yet, the programming paradigms, cluster resource management, as well as data formats and representations differ substantially.

A depiction of an example data pipeline shows the typical challenges researchers are confronted with while building and executing such pipelines:



Therefore, this project aims – with a joint consortium of experts from the data management, ML systems, and HPC communities including ICCS (via CSLab and DBLab) – at systematically investigating the necessary system infrastructure, language abstractions, compilation and runtime techniques, as well as systems and tools necessary to increase the productivity when building such data analysis pipelines, and eliminating unnecessary performance bottlenecks.

Site: https://daphne-eu.eu/

Code: https://github.com/daphne-eu

Daphne1: Optimizing HDFS integration with in the DAPHNE Runtime

The Hadoop Distributed File System (HDFS) is a scalable, fault-tolerant file system used by Hadoop applications to efficiently distribute large datasets across multiple nodes. By design, HDFS facilitates data locality, thereby minimizing network congestion and improving system throughput. HDFS is a sophisticated file system framework designed to store large datasets across numerous commodity machines. HDFS follows a master/slave architecture comprising a single *NameNode* for managing the file system metadata and regulating access, and multiple *DataNodes* to handle data storage on individual cluster nodes. Files are split into large blocks, which are grouped at datacenter racks, distributed and replicated among DataNodes, ensuring high fault tolerance. The NameNode orchestrates the block storage, while DataNodes serve read and write requests from the clients.

The integration of DAPHNE with HDFS signifies a strategic advancement in managing distributed data processing tasks. In terms of the DAPHNE Runtime, our goal is to enhance its performance by taking advantage of **data locality**. Additionally, we wish to showcase DAPHNE's extensibility by integrating a state-of-the-art distributed file system with it. The use of *Libhdfs3* is instrumental in bridging DAPHNE with HDFS, ensuring that the runtime leverages its strengths. All details of the current integration of HDFS with Daphne are found here: https://github.com/daphne-eu/daphne/blob/main/doc/HDFS-Usage.md

In this project, you are asked to extend and test the performance of the current HDFS integration by changing the default HDFS chunk replacement policy and replication factor depending on the input data.
Specifically, you are asked to implement custom block placement strategies by extending the abstract `BlockPlacementPolicy` class in HDFS and implement the required methods. The BlockPlacementPolicy class defines the basic interface for HDFS block placement policies, and provides default implementations for many of the methods that you can override as needed. (https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUpgradeDomain.html)
You will implement a fair round-robin and a highly-skewed policy (and maybe a custom one) and test DAPHNE's I/O performance in each case.

2. Comparison between Vector DataBases

In this project, you will be asked to compare the performance of various aspects of two popular and open-source vector database systems. This entails the following aspects that must be tackled by you:

1) **Installation and setup of two specific Vector DBs:** Using local or okeanos-based resources you are asked to successfully install and setup the two systems. This also means that if any (or both) of the DBs got a cluster edition (distributed mode) that this will be also available for testing.
2) **Data generation (or discovery of real data) and loading to the two DBs:** Using either a specific data generator, online data or artificially creating data, you should identify and load a significant amount of vectors into the databases. Ideally, loaded data should be: a) big (or as big as possible), not able to fit in main memory, b) have varying dimensionality (vector size and type - integer, float) and c) the same data loaded in both databases. Data loading is a process that should be monitored, namely: the time it takes to load a small, medium and the final amount of data, as well as the storage space it takes in each of the databases (i.e., how efficient data compression or possible indexing is).
3) **Query generation to measure performance:** A set of *similarity queries* (common to both DBs) must be compiled in order to test the performance of the vector DBs. Queries should target similar similarity metric algorithms (if implemented on both DBs, e.g., Euclidean distance (L2), Inner product, Cosine similarity, etc.) with and without filters (https://github.com/qdrant/ann-filtering-benchmark-datasets)
4) **Measurement of relevant performance metrics for direct comparison:** Client process(es) should pose the queries of the previous step and measure the DB performance (query latency, throughput, CPU load if possible, etc). Teams should be careful and compare meaningful statistics in this important step.

Teams undertaking this project can take advantage of existing benchmarking code either in principle or in whole. I suggest looking at the qdrant Benchmark Suite (https://github.com/qdrant/vector-db-benchmark), which for many of the DBs contains code for data generation, loading, queries and measurement.
Besides the aforementioned project aspects, you are free to improvise in order to best demonstrate the relative strengths and weaknesses of each system. By the end of your project, you should have a pretty good idea of what a modern vector DB is, its data and query processing model and convey their strong/weak points.
This project has 3 different DB combinations:

Vec1: qdrant, milvus

Vec2: milvus, weaviate

Vec3: chroma, qdrant

Also possible: Marqo, pinecone

3. Comparison between Python scaling frameworks for big data analysis and ML

Ray (https://www.ray.io/) is an open source unified computing framework that facilitates the scaling of Data Analytics, Machine Learning and Artificial Intelligence workloads in Python. Ray Datasets is a component of the Ray ecosystem that provides a high-performance data processing API for large-scale datasets. It offers many advantages for working efficiently with datasets. As part of this work, you are required to compare Ray with other modern systems that scale python analytics code such as Apache Spark or Dask. This entails the following aspects that must be tackled by you:

1) **Installation and setup of two specific systems:** Using local or okeanos-based resources you are asked to successfully install and setup the two systems.
2) **Data generation (or discovery of real data) and loading to the two DBs:** Using either a specific data generator, online data or artificially creating data yourself, you should identify and load a significant amount of input data for testing the systems. Ideally, loaded data should be: a) big (or as big as possible), not able to fit in main memory, in the order of several GB or millions/billions of records, b) the same data loaded in all tests.
3) **Code for measuring performance:** A set of python scripts (workflows) must be created/identified in order to test the performance of the two systems in scaling it. Scripts should target both ML-related and standard data-management operations (ETL) that are common over different cluster resources. Each team is free to improvise but you can test both individual tasks (e.g., graph operators like PageRank, triangleCount, popular ML operations such as prediction, clustering, etc.) as well as more complex jobs (you can use https://www.kdnuggets.com/ and https://www.kaggle.com/ for example on this). These scripts should be posed over i) a different number of nodes/workers, ii) different input data size/type. Teams should be careful and compare meaningful statistics in this important step.

Besides the aforementioned project aspects, you are free to improvise in order to best demonstrate the relative strengths and weaknesses of each system (strengths in particular operations, scalability with memory/cores, etc). This project has the following different combinations:

Python1: Ray, Dask

Python2: Ray, Apache Spark

Python3: Ray, PyTorch

4. Distributed Execution of SQL Queries

PrestoDB https://prestodb.io/ is a distributed SQL query engine designed to query large data sets distributed over one or more heterogeneous data sources. In this project, you will be asked to benchmark its performance over different types of queries, data locations and underlying storage technologies. In more detail, the following aspects must be tackled by you:

1) **Installation and setup of three specific stores:** Using local or okeanos-based resources you are asked to successfully install and setup three open-source storage systems or Databases and PrestoDB. These will be used as the source of data for distributed execution of SQL queries via PrestoDB.
2) **Data generation and loading:** Using a specific data generator, you should identify and load a significant amount of tuples into the three databases. Loaded data should be: a) big (or as big as possible), not able to fit in main memory, in the order of several GB or millions/billions of records, b) Different tables of the data should be stored in different stores, according to a strategy that you will devise. This strategy will entail different ways of distributing tables so that you will be able to measure how Trino behaves.
3) **Query generation to measure performance:** A set of queries must be selected in order to test the performance of PrestoDB and its optimizer. Queries should be diverse enough to include both simple queries (e.g., simple selects) and complex ones (range queries and multiple joins and aggregations) and cover some or all the input tables.
4) **Measurement of performance:** You should then pose the queries of the previous step and measure the performance (query latency, optimizer plan) over different PrestoDB cluster resources. This means that each of the queries of the previous step should be posed over i) a different number of workers, ii) a different data distribution plan. Our goal is to identify how the distributed execution engine and optimization works under a varying amount of data, data distributed in different engines and with queries of different difficulty.

Teams undertaking this project should take advantage of existing benchmarking code, namely the TPC-DS benchmark (https://www.tpc.org/tpcds/), which contains code for data generation and relevant queries.

Besides the aforementioned project aspects, you are free to improvise in order to best understand the performance of different queries. By the end of your project, you should have a pretty good idea of how query processing in PrestoDB works and propose better data distribution strategies. This project has 6 different combinations (using PrestoDB for each of the following):

SQL1: PostgreSQL, Cassandra, Kudu

SQL2: MySQL, MongoDB, Memory (main mem)

SQL3: MongoDB, Cassandra, PostgreSQL