Information Systems, Analysis and Design
Class Projects (2023-24, Fall Semester)

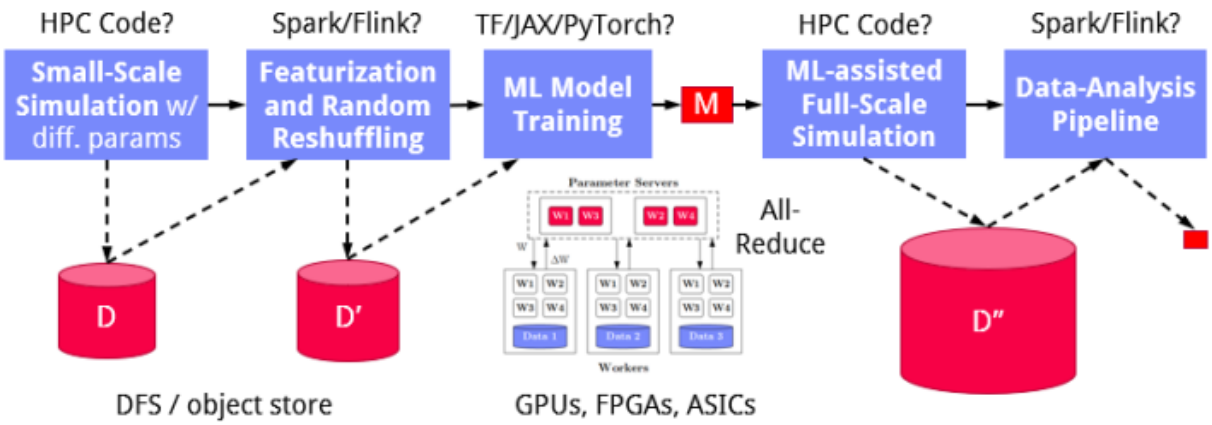
1. Daphne-project related **2**
Daphne1: Adding the COO sparse matrix representation in the DAPHNE Runtime 3
Daphne2: Implement DaphneDSL map() on entire row/column 4
Daphne3: Readers/writers for more file formats 5
2. Comparison between Time Series DataBases 6
Time1: InfluxDB, QuestDB 7
Time2: InfluxDB, TimescaleDB 7
Time3: CrateDB, QuestDB 7
3. Comparison between Python scaling frameworks for big data analysis and ML 7
Python1: Ray, Dask 8
Python2: Ray, Apache Spark 8
Python3: Ray, PyTorch 8
4. Distributed Execution of SQL Queries 9
SQL1: PostgreSQL, Cassandra, Redis 10
SQL2: MySQL, MongoDB, Redis 10
SQL3: MongoDB, Cassandra, Redis 10

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: Adding the COO sparse matrix representation in the DAPHNE Runtime

Many real-world applications in data science are characterized by the prevalence of sparse data, where almost all cells of a matrix are zero. Instead of storing and processing all cells in a dense representation, it is common to use sparse representations, which only store the non-zero values. That way, the storage requirement and runtime can be improved significantly. DAPHNE already supports CSR (compressed sparse row) as a sparse matrix representation. However, there are other representations with unique characteristics. Examples include (a) MCSR (modified CSR), which is more update-friendly as the values of each row are stored in a separate array, (b) CSC (compressed sparse column), which benefits column-major access patterns, and (c) COO (coordinate-based), which is better suited for ultra-sparse data. None of these sparse representations is the most suitable one in all situations. Therefore, being able to freely choose the format has the potential of better adapting to the circumstances at hand.

Task: This project is about extending DAPHNE by the additional sparse matrix representation COO, which represents a sparse matrix by three arrays of the same length: *row indexes*, *column indexes*, and *values*. Each non-zero element in the matrix is represented by three corresponding elements in these arrays, i.e., described by its row and column position and the value. Typically, the triples are sorted by row index (primary) and column index (secondary). For instance, the matrix

```
0 0 0 1 0 0
0 2 0 0 0 0
0 0 0 0 0 0
0 0 3 0 0 0
```

would be represented as

```
row indexes: 0, 1, 3
col indexes: 3, 1, 2
     values: 1, 2, 3
```

Addressing this task includes:

1. the implementation of COOMatrix, the corresponding subclasses of the Matrix class, including all required methods like get()/set()/append() and slicing

2. the implementation of new kernels (physical operators) tailored to the respective representation for a few decisive operations (e.g., elementwise unary/binary, matrix multiplication, transposition, full/row-wise/column-wise aggregations, …)
3. strategy to select COO over a dense matrix representation (e.g., based on the estimated sparsity and physical size in bytes)

Implementation in C++.

Daphne2: Implement DaphneDSL map() on entire row/column

DaphneDSL supports the second-order function map(), which is well-known from other programming languages. Given a matrix and a user-defined function written in DaphneDSL, map() applies the given function to each element of the given matrix. For instance, the following script

```
def timesTwo(x) {return x*2;}
X = reshape([1, 2, 3, 4], 2, 2);
print(map(X, timesTwo));
```

results in the following output

```
DenseMatrix(2x2, int64_t)
2 4
6 8
```

Currently, map() supports only the mapping of individual cells of a matrix. For more expressiveness, it would be desirable to be able to map an entire row (or column) of the input matrix to an entire row (or column) of the output matrix. Custom row/column-wise aggregations would be one special case.

Task: This project is to extend the current implementation of map(), such that UDFs on entire rows/columns are supported. If the input of the UDF is a row matrix, the output can be a row or a scalar; if the input is a column matrix, the output can be a column or a scalar. This requires changes to the DaphneDSL parser, the DAPHNE compiler, and the map()-kernel. Implementation in C++.

Some example uses of map() that should be enabled:

- Return the sum of the first and last element of the given row matrix (row to scalar).
- Return the top-three elements of the given column matrix (column to column).

Note that these are just simplified examples to demonstrate the idea. Make sure you understand the existing map() implementation first.

Daphne3: Readers/writers for more file formats

DAPHNE already supports reading files in a handful of typical formats, such as CSV, Matrix Market, and Parquet. However, there are other commonly used file formats.

Task: This project is to implement (C++) readers and writers for additional file formats in DAPHNE. These new readers and writers may use existing libraries for handling certain file formats. Readers shall read the contents of a given file into either a DAPHNE matrix or frame. Writers shall write a DAPHNE matrix or frame into a file.

File formats of interest include, but are not limited to:

- the “Technical Data Management System” (TDMS format)
- various image formats (e.g., JPEG2000)

Teams should take a look at (a) the existing file readers and writers in src/runtime/local/io/ as well as at the read() and write() kernels in src/runtime/local/kernels/Read.h and .../Write.h, and (b) at the concept of file meta data files (https://github.com/daphne-eu/daphne/blob/main/doc/FileMetaDataFormat.md) used in DAPHNE.

2. Comparison between Time Series DataBases

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

1) **Installation and setup of two specific Time-Series 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 yourself, you should identify and load a significant amount of tuples into the databases. 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 of records, b) 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 is).
3) **Query generation to measure performance:** A set of queries (common to both DBs) must be compiled in order to test the querying performance of the timeseries DB storage and indexing. Depending on the data, queries should target the time dimension in both point and range queries (and multiple ranges and grouping functions, i.e., average, group-by, window queries).
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 strongly suggest looking at the Time Series Benchmark Suite (TSBS, https://github.com/timescale/tsbs), which for some 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

time-series DB is, its data and query processing model and convey their strong/weak points. This project has 3 different DB combinations:

Time1: InfluxDB, QuestDB

Time2: InfluxDB, TimescaleDB

Time3: CrateDB, QuestDB

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 must be created/identified in order to test the performance of the two systems in scaling it. Scripts should target different ML- and big data-related operations that are common over different total 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

Trino and PrestoDB are distributed SQL query engines designed to query large data sets distributed over one or more heterogeneous data sources. In this project, you will be asked to benchmark their 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 Trino/PrestoDB. These will be used as the source of data for distributed execution of SQL queries via Trino/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 Trino/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 Trino/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 Trino/PrestoDB works and propose better data distribution strategies. This project has 6 different combinations (using Trino or PrestoDB for each of the following):

SQL1: PostgreSQL, Cassandra, Redis

SQL2: MySQL, MongoDB, Redis

SQL3: MongoDB, Cassandra, Redis