POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Ibis as a Unified Data Processing Language: Integration, Comparison and Benchmarking of Heterogeneous Backends

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Carlo Ronconi

Advisor: Prof. Alessandro Margara

Co-advisors: Prof. Gianpaolo Cugola, Davide Canali, Fabio Chini, Luca De Martini

Academic year: 2023-2024

---

1. Introduction

In the modern world, human activities generate vast amounts of data. In response, a wide range of data-processing platforms have been developed, based on different programming paradigms, offering unique features and specialized interfaces. First were relational databases and the SQL model, then data-warehousing solutions for Analytical processing. Later, dataflow engines were introduced, simplifying parallel, distributed computation, with their functional-inspired API. In the meantime, the Python ecosystem gained popularity, offering increased expressiveness for local data analysis. While having this many tools at our disposal is beneficial, it also introduces complexity. Choosing a data-processing platform often means committing to a specific interface and feature set, making it hard to switch between systems, while professional figures need to learn multiple paradigms to access and transform data. Ibis is a Python library that abstracts the underlying data-processing engine, providing a unified API to interact with different backends. It proposes a new model for data workflows, where the same code-base can be used for different systems, and shared among different professional figures and environments, from local development to production deployment. In our thesis, we aim to validate the effectiveness of Ibis as a common language for all data processing purposes. To do so, we propose scenarios representing real-world data-processing use-cases. We use Ibis to express these scenarios with different backends, evaluating its performance an usability. Additionally, we build a connector for our academic department's data-processing engine, Renoir, enabling it as an Ibis backend. We then evaluate its performance and the usability advantages introduced by the Ibis frontend.

2. Background

2.1. Data processing paradigms

Multiple data-processing paradigms have been introduced over the years. Here, we give a brief introduction to the most common ones, and finally describe Ibis, which aims to abstract and unify them.

SQL is a declarative language, introduced in the 1970s, based on the relational model [1]. It is used to interact with relational databases, and has been modified and expanded over the years to support different vendor's features. In the 1980s, a great effort was undertaken to standardize SQL, leading to the ANSI and ISO standards. With the inception of data warehousing, SQL was extended to support Analytical processing, enabling better business insights into operational data.

Dataflow engines emerged as a class of data-processing systems aimed at performing timely, continuous and unprompted computations of information [2]. As their name suggests, they are based on the *dataflow* programming model, which differs from traditional *imperative* programming in that it removes the notion of mutable shared state between instructions, freeing programs from implicit dependencies and enabling parallel execution. Two main approaches have emerged for materializing dataflow computations: *scheduling* and *pipelining*. In the first case, input data is split into batches, and groups of operators (i.e. instructions of the dataflow program) are scheduled to work on batches independently. In the second approach, all operators are active for the whole duration of the computation, and data is streamed through them.

With the rise of data science, SQL became limiting for quick, local analytical tasks. The Python ecosystem, with libraries like Pandas, offered a more expressive and interactive environment for data analysis. Pandas introduced the DataFrame data structure, which is an implicitly-ordered, two-dimensional, size-mutable, tabular data structure, with a query language including aspects from relational logic, linear algebra and spreadsheets.

Ibis defines a Python DataFrame API that executes on different query engines, abstracting away their differences. It is designed to be a common language for all data processing purposes, enabling users to write code once and run it on different backends, from local development to production deployment [5].

2.2. Backends

Before the analysis, we briefly introduce to the backends we use in our evaluation. **DuckDB** is an in-process, columnar database system designed for analytical workflows. It offers a feature-rich SQL interface, together with client APIs for numerous programming languages. **Polars** is a high-performance DataFrame library for Python and Rust, improving on Pandas's performance with three key features: it is written in Rust, allowing for a multi-threaded design; it uses columnar data structures; it leverages a lazy execution paradigm, increasing optimization opportunities. **Flink** is an open-source, distributed framework for batch and stream processing. It is a dataflow engine, with a *pipelined* architecture, well-suited for continuous, real-time processing of streaming data, although compatible with batch processing, by streaming the whole dataset. **Spark** is an open-source, distributed data-processing engine, based on the *scheduled* dataflow model. This approach favors overall throughput and is more akin towards batch datasets, although streaming is supported through micro-batching. **RisingWave** is an open-source, distributed SQL database designed for stream processing. It leverages a unique implementation of *materialized views* to provide always-updated streaming query results, using an actor-based architecture. **Renoir** is a distributed data-processing platform, formerly known as Noir [4], developed at Politecnico di Milano. It brings together MPI-rivaling performance with a high-level Flink-inspired interface in Rust, enabled by its dataflow, *pipelined* architecture.

2.3. Benchmarks

Along with custom-designed queries, we use the following benchmarks to evaluate Ibis's capabilities. **Nexmark**, a benchmark designed to evaluate the performance of stream-processing systems; **Derived from TPC-H**, a decision support benchmark to assess the performance of decision-support systems.

3. Scenario definitions

In this section, we describe the process of synthesizing the extensive variety of real-world data workflows into a set of scenarios, based on a few distinctive features. This is aimed at understanding the effectiveness of Ibis as a common interface for different data processing engines in different contexts. Four main variables are used to differentiate scenarios: input data type, processed data type, included backends and ad-hoc queries. *Input data*, can be of three categories: from CSV *file*, pre-loaded (i.e. *cached*) in the backend's database or *streamed* from a Kafka topic. *Processed data* also has three alternatives: be stored to a CSV *file*, *streamed* to a sink Kafka topic or *discarded* after being consumed. For each scenario, we also define a set of *ad-hoc queries* best representing the intended use-case. Below we describe the scenarios we defined, with a summary of the use-case they represent and the backends we chose to implement them with.

1. *Preprocessing*: Data has been collected from external sources and stored in a file. It is cleaned and aggregated, before being stored back to a file for future analysis. Input data: *file*. Processed data: *file*. Backends: DuckDB, Polars, Flink, Spark, Renoir.
2. *Views*: Data is streamed from a source Kafka topic, while the query is defined as a view, writing updated results in a Kafka sink topic. We measure the throughput between adding all input data and collecting all output results. Input data: *stream*. Processed data: *stream*. Backends: Flink, Spark, RisingWave.
3. *Analytics*: A first query is performed on a static dataset, and, based on its results, the analyst decides to perform a second query on the intermediate results. This scenario measures the incremental time-savings of caching the intermediate results with Ibis. Input data: *cached*. Processed data: *discarded*. Backends: DuckDB, Polars, Spark, RisingWave, Renoir.
4. *Exploration*: An analytical query is performed directly on raw data, and its results are immediately consumed by the user. Input data: *file*. Processed data: *discarded*. Backends: DuckDB, Polars, Flink, Spark, Renoir.

4. Ibis-Renoir compiler

An Ibis adapter for Renoir, our own department's data-processing engine, is implemented, providing a Python API for it and benchmarking it alongside other backends.

4.1. Compiler implementation

To implement the connector, we leverage Ibis's architecture, which provides an *Abstract Syntax Tree* (AST) for user-defined queries. We parse the AST and perform *source-to-source* compilation, generating Renoir Rust code for the query. Then, we use *Cargo* to compile the Rust code to an executable, which we run to obtain the query result. The portion of Rust code that the compiler writes is contained in the *main.rs* file, which is located within a template project, so that Cargo can compile it as customary. The AST transversal uses a *post-order depth-first search* algorithm to explore the tree, guaranteeing that method-call predecessors are written before the method-call itself. The transversal produces an ordered list of *Operator* instances, representing blocks of Rust code that can be autonomously compiled and executed in sequence. Operators, when generating code, produce an ordered list of *Struct*s, representing Rust struct definitions, which are compiled later and written on top of the *main.rs*. Finally, the template is compiled with Cargo and executed, yielding the query results.

4.2. Incremental processing and testing

To enable benchmarking Renoir in the *Analytics* scenario, we build a first implementation of interactive-Renoir using the *Evcxr* [3] library, an evaluation context for Rust code. This way, we can incrementally execute queries, caching intermediate results. A test-driven approach is used when developing the compiler, to ensure the correctness of the generated Rust code. Test cases are written before adding expressiveness to the compiler, and are run after each change to verify that the compiler is still working as expected.

5. Experimental methodology

In this section, we detail the implementation of the benchmark harness capturing the behavior of each scenario, with each of its backends.

5.1. Benchmark harness

We implement two base scenario classes, one for batch scenarios (Preprocessing, Analytics, Exploration) and one for streaming scenarios (Views), where each scenario is then defined as a subclass of these. The same is true for backends, where connectors for each backend, in the streaming or batch context, are implemented as subclasses, where base definitions are used to define a common interface. Queries are implemented using the Ibis API, and are shared among the different backends of the same scenario, thanks to Ibis. Additional components are added to the benchmark harness to orchestrate its execution, including connectors for Kafka, a logger and a policing mechanism to detect overly long-running queries.

5.2. Deployment setup

As benchmarks are ran on a remote machine, we use data generation scripts to create data for the scenarios. In the deployment, we use virtual environments to isolate the Python dependencies for backends, and we have scripts to automatically download and launch the binaries for standalone backends (i.e. Spark, Flink, RisingWave). We run the benchmarks on a centralized machine, without distributing the backends capable of doing so, to ensure a fair comparison, and because our focus is on the Ibis frontend rather than absolute performance.

6. Results

We present the results of our evaluation, showing the performance of Ibis with different backends in the scenarios we defined. Due to space limitations, only the most notable results are highlighted, while the complete analysis can be fond in the thesis. The dataset size was defined using trying to balance realistic amounts with manageable execution times, setting a maximum of 1-hour for each scenario. In some cases, when single backend is responsible for the delay, benchmark runs are repeated with different dataset sizes. 5 queries are performed for each test case in each scenario, after a single warmup run. The memory measurements we show below only represent the client-side consumption for standalone backends, for which memory consumption is steady and not informative, as they strategically deallocate memory only when full capacity is reached.

6.1. Preprocessing (s1)

Figure 1 and Figure 2 show the comparative results of the scenario. Centralized backends (DuckDB, Polars) are better suited for the use-case, where data is read and written locally, while distributed backends can't leverage their parallelism fully, due to the local deployment, but their performance is comparatively better with the larger dataset. Flink has a bottleneck in the sink stage, where parallelism is forced to 1 in order to write to a single file, making it incapable of running the 10-Million dataset. Renoir competes with the best dataflow backend, Spark, although limited in its performance by the compiler, having a significant overhead and producing un-optimized queries. Ibis works well in the query definition phase, but custom solutions are needed to connect to and configure each backend, according to its specific requirements.

6.2. Views (s2)

Figure 3 shows the summarized results of the scenario. All backends have solid results, being well-suited to a streaming scenario. RisingWave performs best, being designed from the ground-up to work with external connectors (e.g. for Kafka) and to provide good results with simple deployment architectures. Flink and Spark also perform well, with the latter being slower due to its *scheduling* dataflow design, less sited to purely-streaming datasets than Flink's *pipelined* approach. Here, Ibis doesn't provide adequate support for the scenario. Although query definitions can be shared, each backend requires a completely custom harness, due to the very different naming conventions and abstractions of the engines. It fails in enabling a common interface for the scenario.

6.3. Analytics (s3)

Figure 4 and Figure 5 show the complete results of the scenario, comparing *one-shot* ("os" in the plots) baseline scenarios to the cached analytic workflow. In the baseline, we measure the time to store the starting dataset, perform a first query, and then a second one from scratch, as if the intermediate results aren't available, In the cached case, we instead cache the first query's result, and execute the second query starting from these. Figure 6 shows an alternative implementation of the scenario, where the *cache* method is used, instead of *create_table*, on the backends that support it. All backends show small improvements afforded by the caching, which we can mainly attribute to the fact that using *create_table* introduces a high performance penalty as it does more than mere caching, and that *cache* works by storing intermediate results on the Ibis side and not delegating it to the backend, again penalizing performance. Ibis does not provide an adequate support for the scenario, as either caching mechanism gives little performance advantages.

6.4. Exploration (s4)

Figure 7 and Figure 8 show the summarized results of the scenario. Again as in the Preprocessing scenario, centralized backends, such as DuckDB and Polars, perform better than distributed ones, due to the local deployment; Flink encounters similar limitations; Renoir competes with Spark as best dataflow backend. In the thesis, we have a thorough comparative analysis to the Preprocessing scenario, as the two have similar characteristics, but sometimes the Ibis layer introduces unexpected overheads, leading to widely different results. Ibis supports this scenario adequately, but it often requires custom solutions for specific backends, and introduces overheads that could be avoided.

7. Conclusions

With the experience of implementing scenarios through Ibis, with different backends, we can draw conclusions on its current state. Starting from the Ibis-Renoir compiler, we notice that it introduces a overhead to Renoir's performance, but the engine can still compete with others, thanks to its excellent performance characteristics. The layer helps Renoir with an expressive Python API, which enlarges its potential user pool, and is ready for further development, especially in the incremental processing area. An additional Rust Kafka-connector for Renoir could also be built, enabling the engine to be used in the Views scenario, which is well-suited to its design. Regarding the Ibis framework, its capabilities are underwhelming compared to its bold ambition of being a unified data-processing language. Indeed, being able to define queries that run on multiple backends is helpful, and the DataFrame-inspired API is expressive and easy to use. However, working with data often spans beyond simple query-writing, and is also concerned with orchestrating engine and managing the system's resources. Ibis does not provide adequate support for these tasks, and often requires custom solutions for specific backends. When this is needed, the value-proposition of Ibis becomes less clear, as the user is forced to learn the intricacies of each backend, while needing to then translate this knowledge into the Ibis API. The fast-evolving nature of the framework also means that, especially for newer features like the streaming query definitions of the Views scenario, performing the same task with different backends can look different even when using the Ibis API. Again, having to research Ibis's documentation on top of each backend's is a significant overhead, offering few advantages compared to just using the backend's native API. In conclusion, the Ibis framework is an interesting tool for experimenting with data-systems locally without learning their specific APIs, and is catered towards relational databases and DataFrames, while its recent support for streaming backends is not mature. It is not well-equipped for actual production environments, where it lacks an abstraction for backend-specific configurations, and only adds friction to the deployment process. Ibis's ambition to be a common frontend for all data systems is commendable, but it is far from being a one-size-fits-all solution.

8. Acknowledgements

I thank Prof. Alessandro Margara and Prof. Gianpaolo Cugola for their guidance throughout the thesis work, Luca De Martini, Davide Canali and Fabio Chini for their precious advice, my family and friends for their love.

References

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377387, jun 1970.

[2] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44:15:1–15:62, 2012.

[3] David Lattimore. Evcxr github repository, 2024.

[4] Luca De Martini. Analysis of market data with noir: real world application and improvements of a streaming and batch processing framework. Master's thesis, Politecnico di Milano, 2024.

[5] Phillip Cloud Wes McKinney et al. Ibis documentation, 2024.

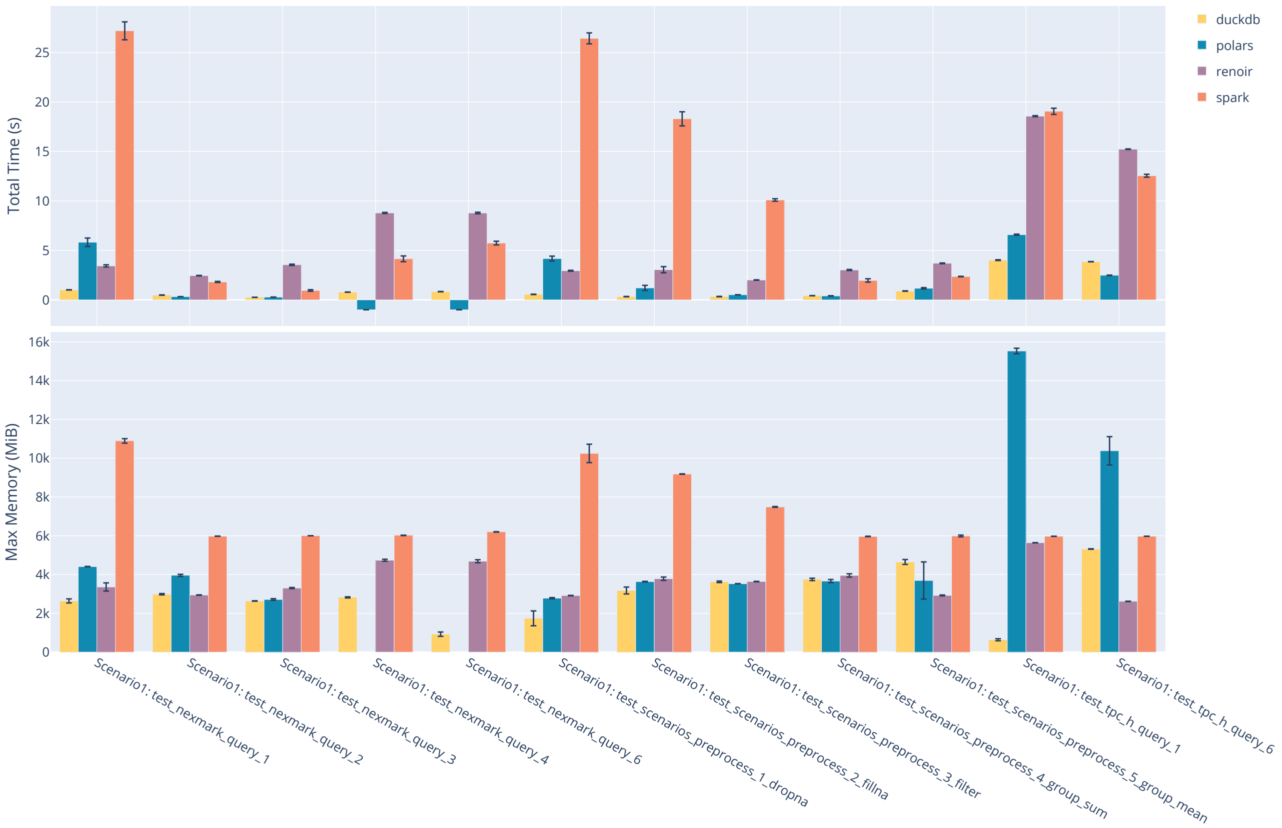

Figure 1: Preprocessing (s1) scenario with 10-Million dataset scale.

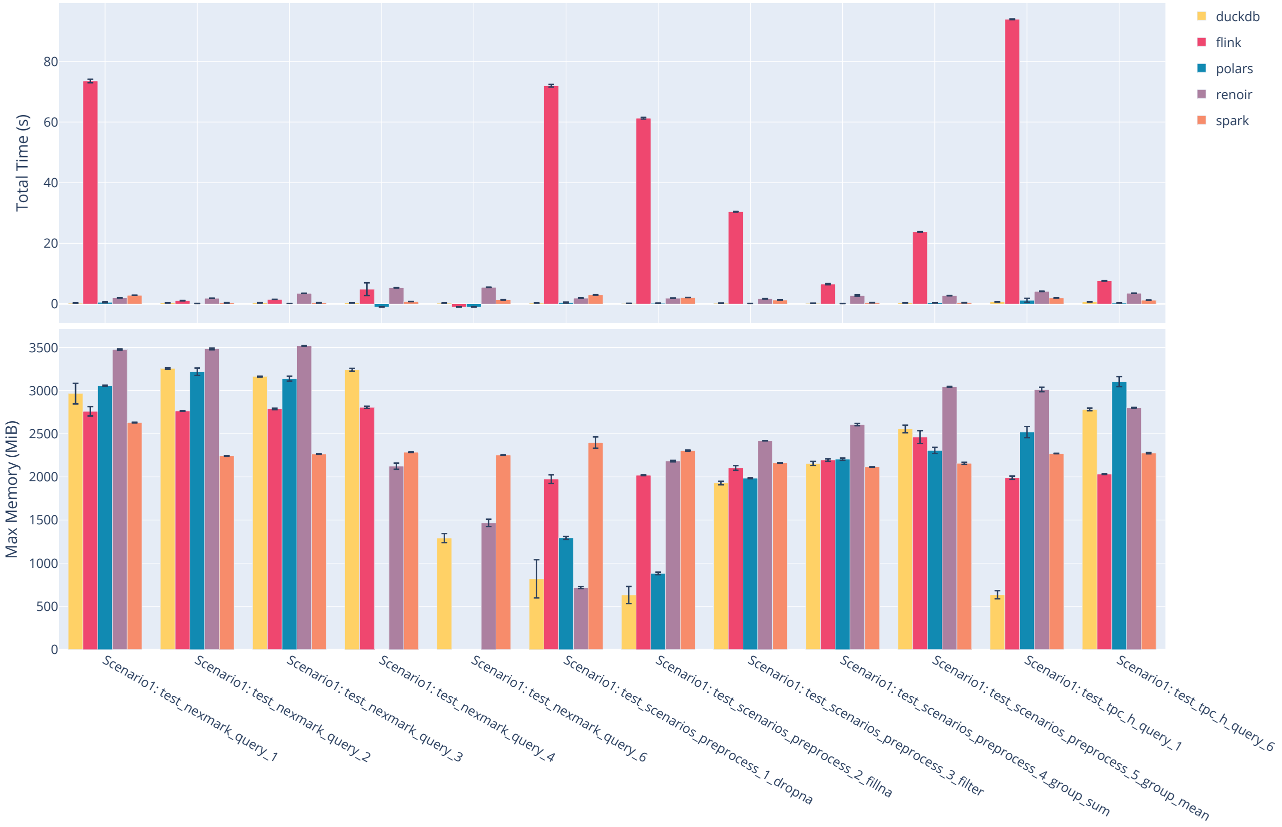

Figure 2: Preprocessing (s1) scenario with 1-Million dataset scale.

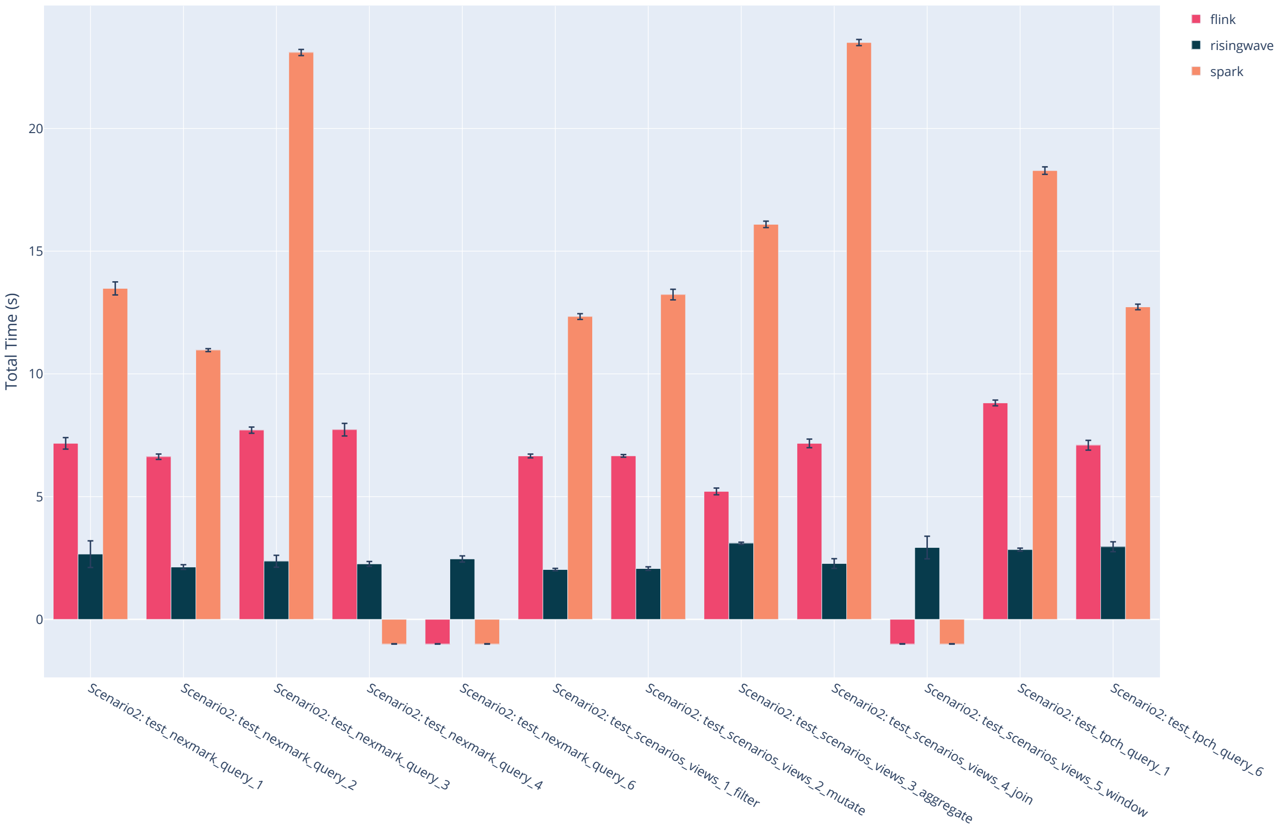

Figure 3: Views (s2) scenario with 10-Million dataset scale.

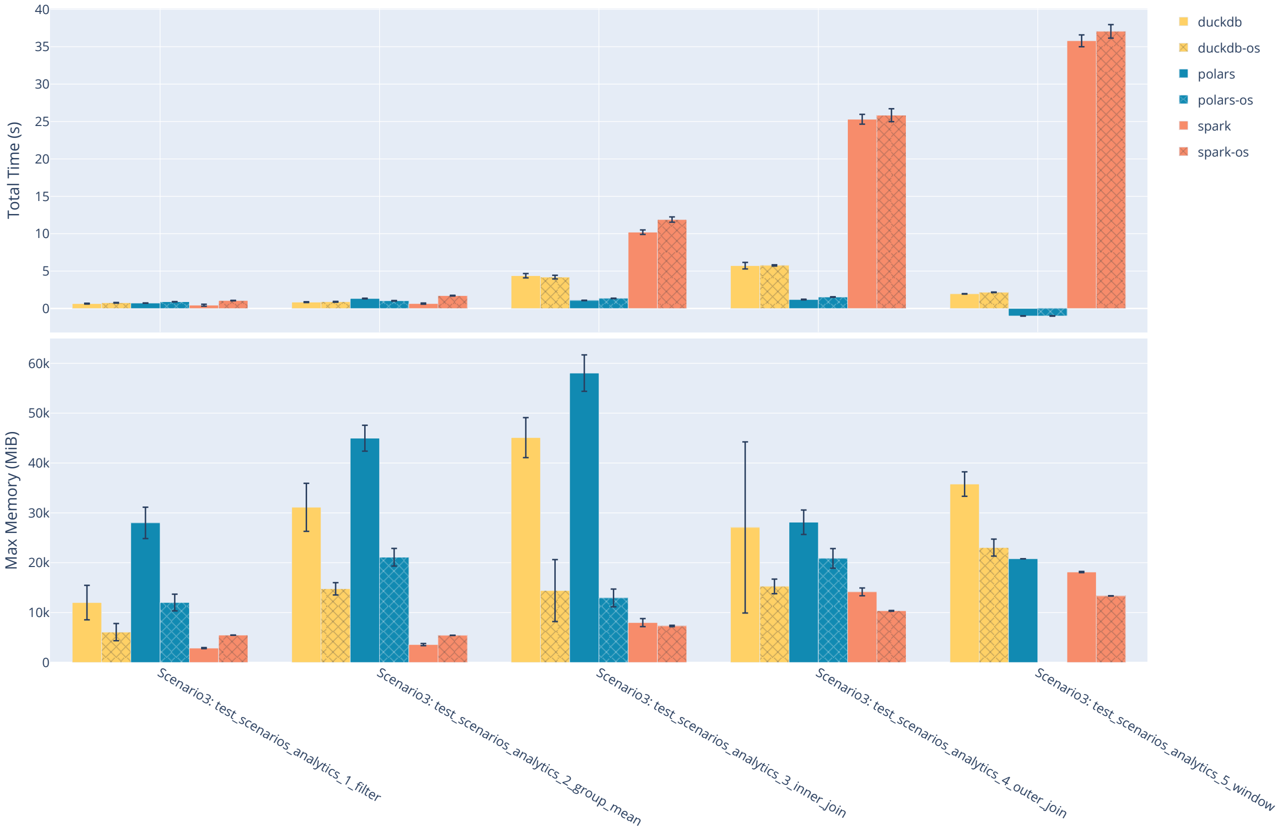

Figure 4: Analytics (s3) scenario with 10-Million dataset scale.

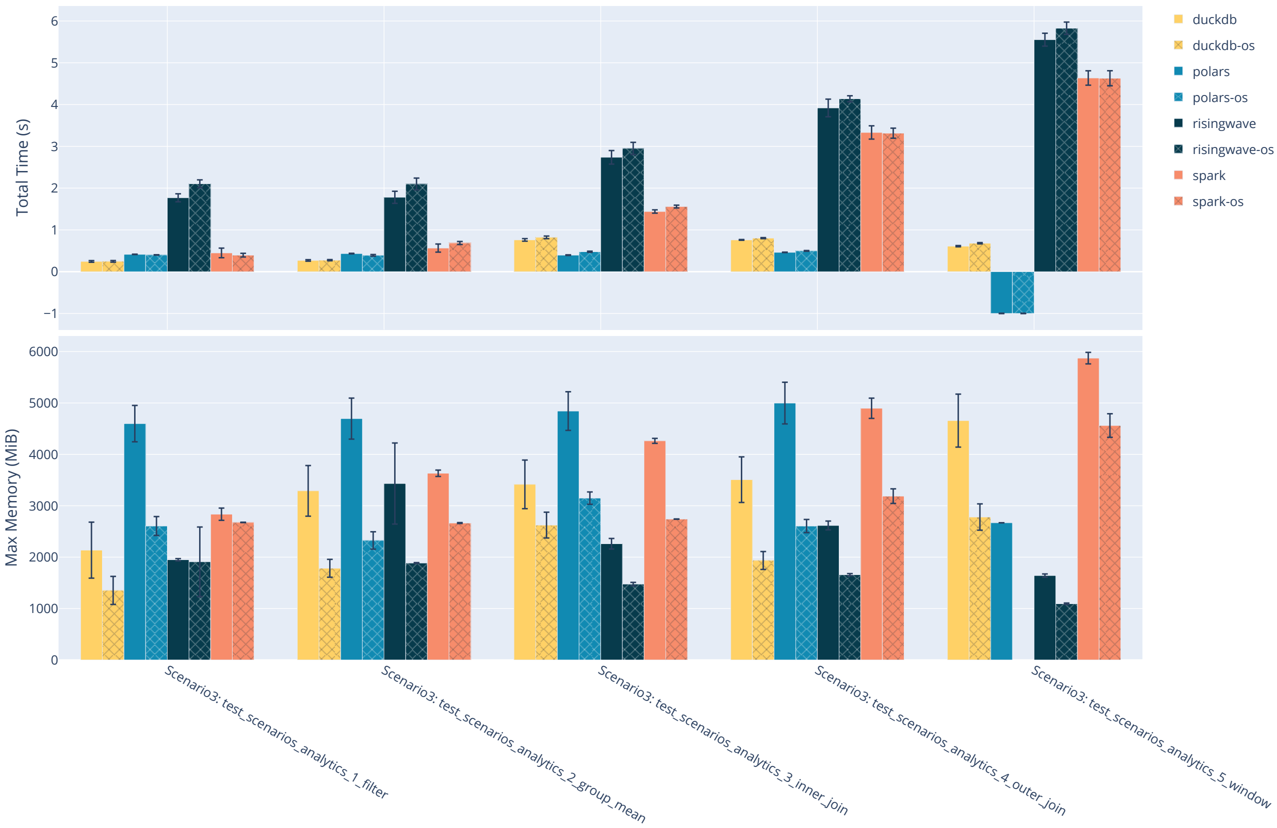

Figure 5: Analytics (s3) scenario with 1-Million dataset scale.

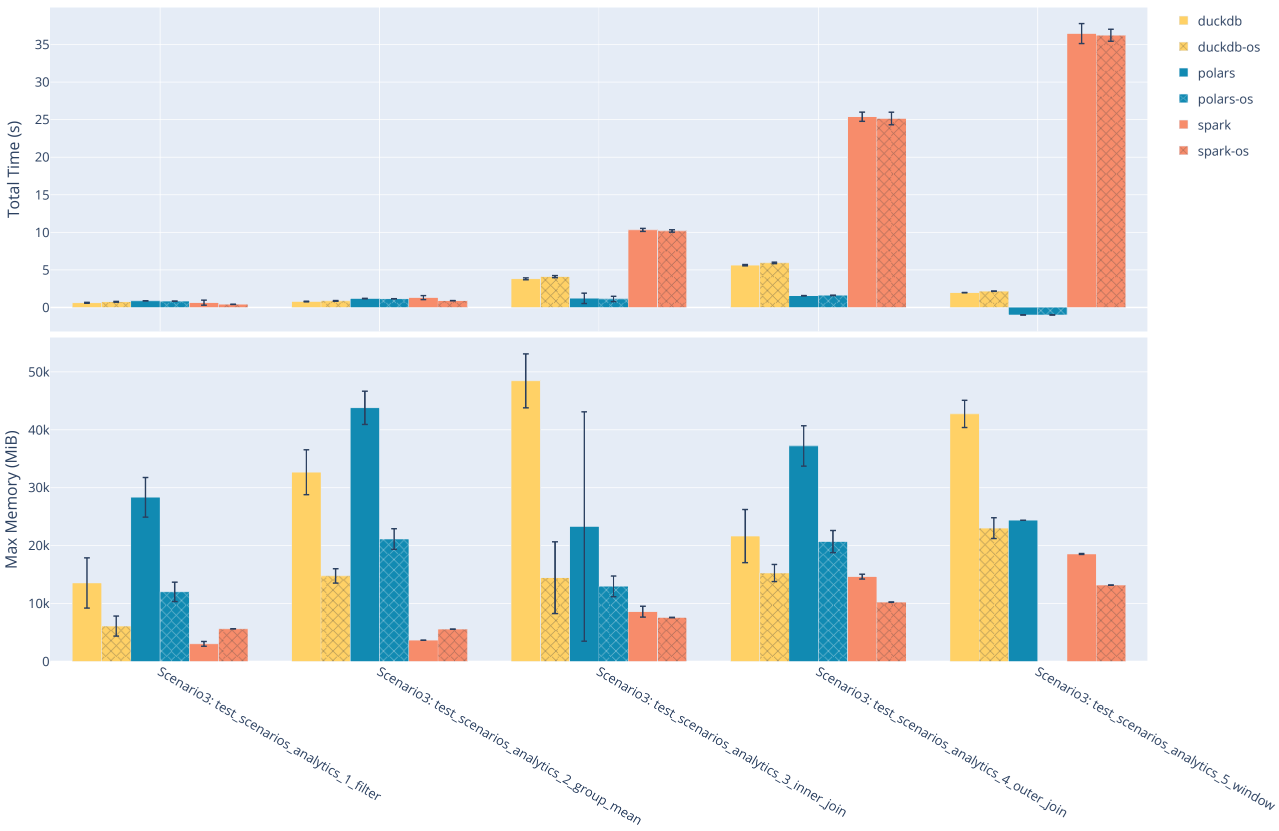

Figure 6: Analytics (s3) scenario with 10-Million dataset scale and cache method.

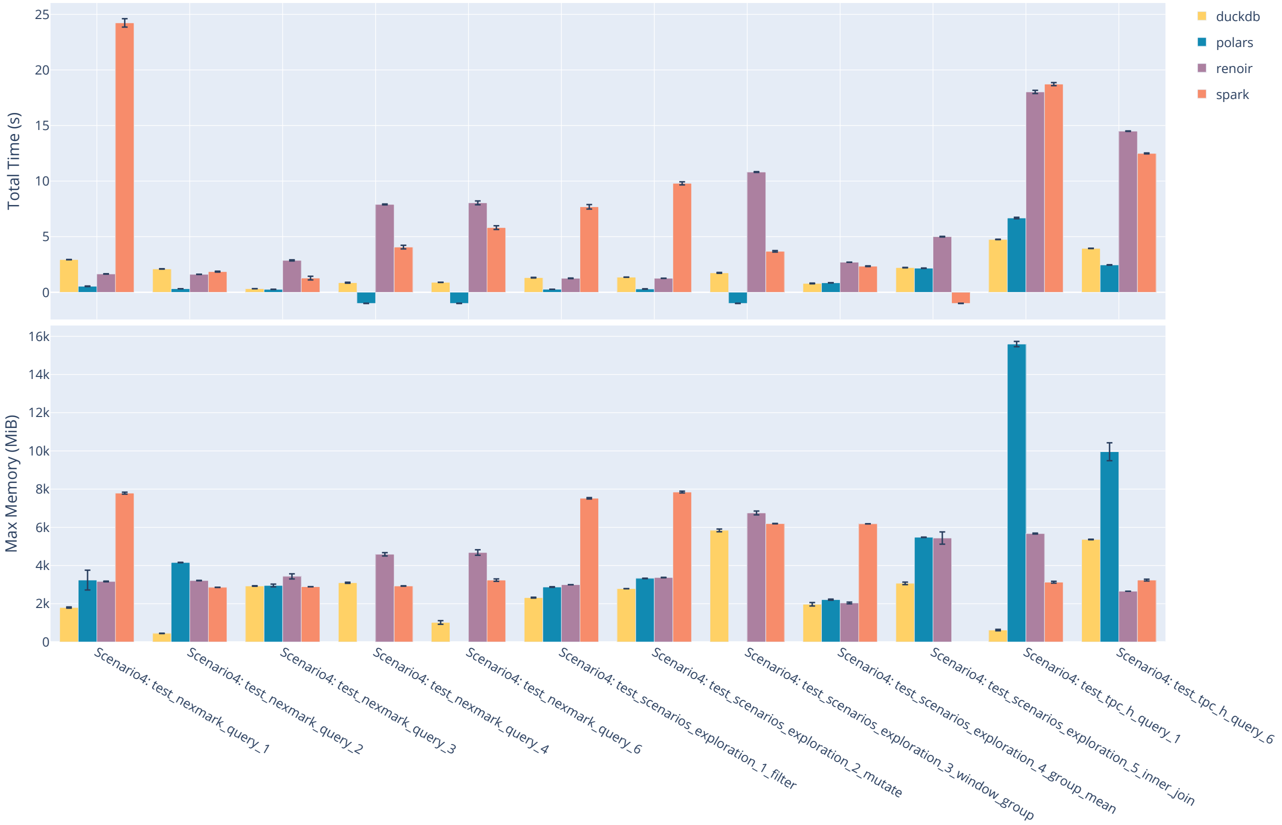

Figure 7: Exploration (s4) scenario with 10-Million dataset scale.

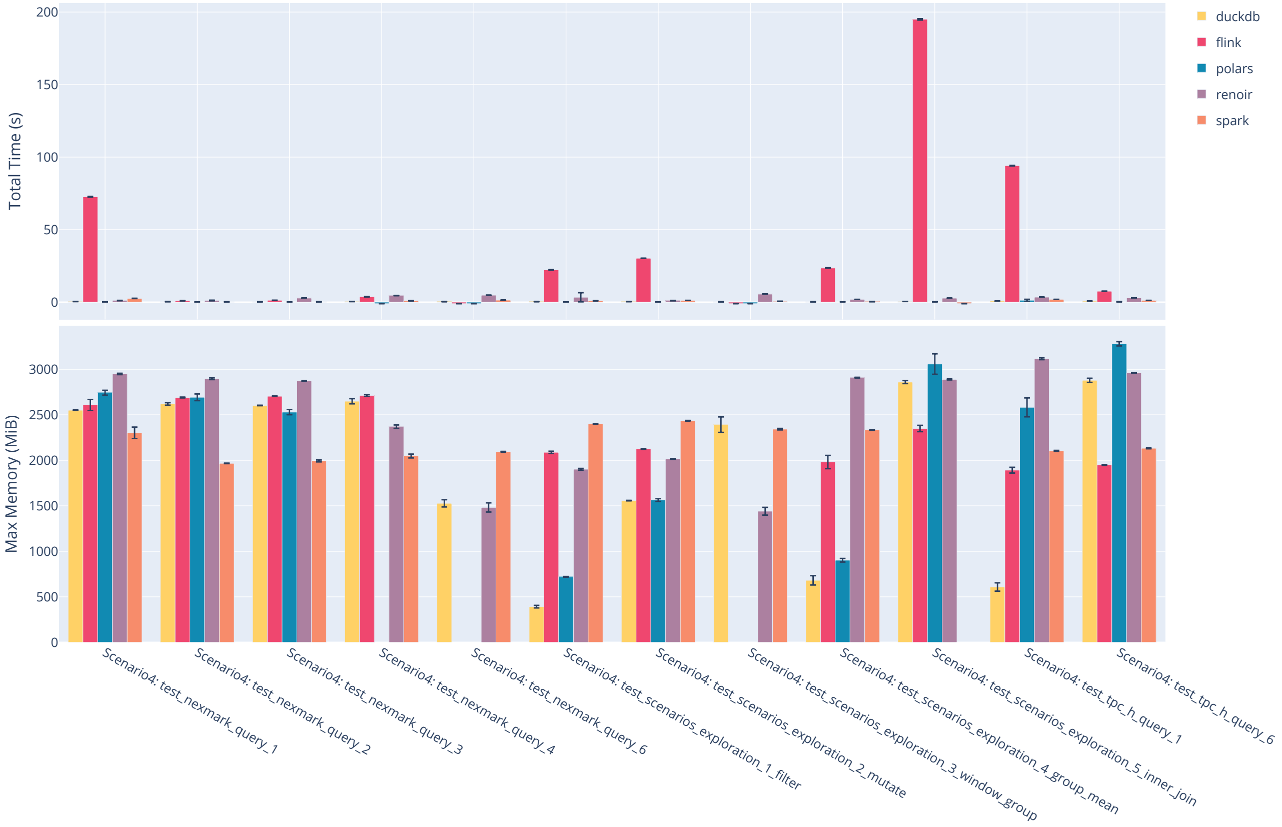

Figure 8: Exploration (s4) scenario with 1-Million dataset scale.