ETH Library

ML Data Processing on Relational Databases

Master Thesis

Author(s):
Kempter, Yves

Publication date:
2024

Permanent link:
https://doi.org/10.3929/ethz-b-000677800

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 486

Systems Group, Department of Computer Science, ETH Zurich

ML Data Processing on Relational Databases

by

Yves Kempter

Supervised by

Prof. Gustavo Alonso and Dan Graur

November 2023–June 2024

D INFK

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ML Data Processing on Relational Databases

Master Thesis

Yves Kempter

June 2, 2024

Advisors: Prof. Gustavo Alonso, Dan Graur

Department of Computer Science, ETH Zürich

Abstract

In this thesis, we introduce Snowpandas, an implementation of the Pandas API, that integrates the Snowflake virtual warehouse with the popular DataFrame syntax, enabling users to take seamless advantage of Snowflake's scalability, performance, and durability. Workflows unsuited to the declarative nature of SQL are thereby brought to the relational database of Snowflake, using its highly efficient operators and data schema. One such workflow ML data preprocessing, has become crucial to ML performance, though ever-increasing data volumes demand increasingly complex preprocessing pipelines that make use of specialized systems. Snowpandas aims to take over where existing implementations of the Pandas APIs can not compete with these specialized systems, when dealing with today's data volumes, in applications such as offline data preprocessing. Snowpandas' implementation interfaces with Snowflake's Snowpark API, making use of its inherent lazy execution paradigm. Designed to the core to preserve this lazy execution scheme, Snowpandas builds SQL queries iteratively through the Snowpark API, enabling Snowflake's service layer to optimize the resulting query holistically. We have implemented the core functionality necessary to perform workloads that are both analytical and transformative. Snowpandas has been tested against existing implementations of the Pandas API, on the star-schema-benchmark (SSB), as well as ML preprocessing pipelines sourced from Kaggle. The results have been promising, Snowpandas outperformed both Pandas and Modin, for all workloads on larger data volumes, though Snowpandas did not achieve competitive performance with Spark for ML preprocessing pipelines at scale. Spark performed preprocessing workloads 1.9 times faster than Snowpandas for high scaling factors, which Pandas and Modin failed to perform. For low scaling factors, Snowpandas finished the preprocessing pipelines on average faster than Pandas, Modin, and Spark by a factor of 1.2, 2.8, and 3.4 respectively. However, Snowpandas scaled considerably better than Pandas and Modin, outperforming them on average by a factor of 17.8 for Modin and 11.8 for Pandas, on the largest scaling factor completed by all systems. Snowpandas excelled at the SSB, on average outperforming Pandas by a factor of 113, Modin by a factor of 214, and Spark by a factor of 7.3.

Contents

Chapter 1

Introduction

In recent years, the ever rising popularity of machine learning (ML) has had a drastic impact on all applications related to data management and analysis. ML has introduced a need for an ever increasing amount of data that brings a variety of challenges to existing systems. For a long time, Pandas has been the go-to data analytics library in the Python stack. However, Pandas struggles to fulfill its role on today's data volumes. In the context of ML, Pandas is widely used for offline data preprocessing. Data preprocessing includes a variety of data transformations aimed at enhancing the performance of the ML model the data is used for. Such operations include data cleaning, feature selection, and feature engineering. This leads to performance problems in ML pipelines where Pandas can not scale up to the increasing data volumes. This can directly be attributed to Pandas' design, which does not support parallelism and data spilling. Thus, several competitors to Pandas have been developed. These systems implement the user familiar Pandas API, but are designed from the ground up with scaling in mind. One such system is Modin. Modin provides a Pandas API while using existing execution engines such as RAY [33] or DASK [39]. As such, Modin brings parallel execution to the Pandas API and offers features such as spilling to disk that are critical in order to scale to today's data volumes. Spark, the Python library of the Spark framework, also provides a Pandas API, offering a well-proven system as an alternative to Pandas. Spark has become a cornerstone of today's data warehouse paradigm because it provides scaling beyond a single machine, a feature Modin does not provide. While the Pandas API allows to read data from relational databases in the form of table reader functions, delegating execution of Pandas API transformations to the relational database in the form of automatically built queries is not a commonly supported feature. In this work, we aim to to just that.

We present Snowpandas, a Pandas API that uses Modin's front end and query translation layer, to bring execution directly to the relational database itself. As such, Snowpandas integrates the Snowpark API as a Modin back-end and thus provides cloud native execution to the Pandas API. Snowflake [23] is a contemporary software-as-a-service (SaaS) provider that aims to bring the advantages of cloud computing to customers with minimal technical overhead. The Snowflake virtual warehouse (VW) is optimized for seamless scaling and abstracts servers and machines away from the perspective of the user. Contrary to traditional database systems, Snowflake separates the execution and storage layer. Storage is provided by S3 or similar services, while the execution unit of the VW is ephemeral. These VW's are managed by Snowflake without the need for configuration or tuning from the side of the user. This makes Snowflake uniquely suited to the challenges posed by Snowpandas, since data can be assessed holistically true to the data warehouse paradigm. Furthermore, the Snowpark API that resembles DataFrame like syntax for Snowflake table is at its core a wrapper for Snowflake's native SQL execution engine, building up SQL queries through sequential transformations on the base Snowpark DataFrame. Thus, the resulting SQL query will be optimized by Snowflake's state of the art query optimizer.

Our implementation of Snowpandas reuses Modin's experimental back-end introduced for the HDK [19] execution engine. HDK is a relational execution engine developed by Intel. Thus, Modin's query translation layer specific to HDK is well suited to modifications that make use of Snowpark as the back-end. As envisioned by Modin's developers, our implementation replaces classes providing data manipulation and data storage with ones that work with Snowpark. At its core is the `SnowflakeDataframe` class. It implements the interface used by the Modin `DFalgQueryCompiler`, used to manipulate data. Some functionality provided by the `SnowflakeDataframe` include `take_2d_labels_or_positional` that perform selections of filtering on the underlying Snowpark frame or `bin_op` that implements binary operations between DataFrames. Since both HDK and Snowpark are based on relational operations, this interface is well matched with the relational mechanism of Snowpark. Most critically, our implementation ensures that transformations are sequentially applied and do not materialize any intermediate results. This ensures that the generated queries executed on Snowflake can be optimized holistically by Snowflake's cloud service layer.

Several experiments were conducted to test Snowpandas on a variety of different workloads. These experiments compared the Snowpandas' implementation against Pandas', Modin's, Spark's and handwritten SQL queries executed through Snowsql. The data was stored on cloud resources and the execution for Pandas, Modin and Spark took place on EC2 instances. We conducted three major experiments:

Star-schema-benchmark (SSB) is a benchmark for relational database systems that focuses on performance for large analytic queries. This starts with the database schema, since the star-schema is optimized for analytic queries by reducing the need for complex joins. The obtained results showed that our implementation of Snowpark preserves performance compared to handwritten SQL queries, indicating that the automatic query generation through Snowpandas does not introduce inefficiencies. This is a very promising result, since it implies our implementation can not be improved upon under the assumption that handwritten queries achieve optimal performance on Snowflake. However the margin between Spark and Snowpandas decreases with increasing scaling factor, this indicates that Spark scales better and that we simply have not reached the data volumne at which Spark outperfors Snowpandas. While Snowparks generally good performance can be attributed to SSB's design, which is suited to relational databases, it is nevertheless an important fact to establish.

Microbenchmarks were desined in order to isolate specific operations encountered in the SSB benchmarks and therefore serve to better understand the SSB results. The results of the microbenchmarks were again positive. Except for one operation, Snowpandas outperformed all other systems for the highest scaling factor. Only the join was performed faster by Spark. However we face the same problem as with the SSB queries, that beeing that we might have simply not reached the data volumne where Spark becomes the best performing system.

Kaggle notebooks served as the source of real world data preprocessing pipelines and here the picture is more clear, while Snowpandas significantly outperformed Pandas and Modin, it only performed better than Spark on the simplest pipeline. We say Snowpandas outperformed Pandas and Modin by a significant margin, though Snowpandas only outperforms Pandas by a factor of 1.2 and Modin by 2.8 on average. This however gives a limited picture, the initial data size is minuscule at under 1MB, when stored as a CSV file. At these scales execution overhead skews the results, since at the moderate scaling factor of 1000, Snopandas already outperforms Pandas by a factor of 11.8 and 17.8 for Modin. Higher scaling factors could only be performed for Spark and Snowpandas, Spark outperforming Snowpandas by a factor of 1.93. Further during these experiments we encountered a specific pattern of operations that severely deteriorates Snowpandas performance. Whether this is due to inefficiencies in our implementation or it is a fundamental problem with Snowpark is beyond our understanding at this point.

Chapter 2

Technical Background

Over the years steadily increasing data volumes have had a drastic impact on the technical frameworks used in data analytics. These challenges are not unique to the machine learning (ML) models themselves, but concern the complete ML pipeline. As such, frameworks have to be developed that can handle the ever-increasing volume of data. Data preprocessing is one such vital step in ML pipelines that has a direct and measurable impact on the performance of the system [15]. Therefore, it proves critical that systems are developed that are both scalable and easily deployable. State-of-the-art preprocessing pipelines employ a complex structure, that focuses on the effective use of GPU/TPU resources [42, 25]. Alternative systems such as Cachew are however being proposed [25], not only reducing complexity but increasing performance. While existing research has already discussed the ability of traditional relational database systems to handle nested data, by translating JSONiq queries to SQL queries that are executed on the Snowflake database [27] and evaluated the use of relational databases in high-energy physics applications [26]. There is little existing work that studies the translation and execution of DataFrame API systems such as Pandas on relational databases like Snowflake.

2.1 Data Preprocessing

While data processing includes a wide spectrum of operations, in the context of ML preprocessing, an emphasis is laid on operations that enable the ML models to accurately make predictions, while maintaining a high degree of generalization. These measures can take many forms, the most basic parts of preprocessing are made up of steps that transform the raw data into a form that can be fed to the model. More advanced methods aim to increase the predictive strength and generalization of the model. [29].

Imputation describes the operation of filling in missing data in the original data set. This is because inconsistencies in the data such as missing values or illegal entries have to be either imputed or eliminated to produce a consistent dataset.

Noise is present in any data source. These imperfections mean that the data does not perfectly represent the statistical distribution that is assumed to be the source of the data. This noise in the data can be treated in various ways, the simplest being the removal of statistical outliers.

Feature selection is a performance-critical step, especially when dealing with large amounts of data. While large amounts of data are collected, it does not mean all of this data has to be included in the input for a model. The data is likely to contain redundant information or data points of little statistical influence. In the best case, this leads to unnecessarily high computational load. In the worst case, it can impact the predictive strength of the ML model. Feature selection aims to select only the relevant data points and thereby aims to reduce these negative effects.

Instance manipulations describe the process of selection a subset of the initial data entries. Even though a large sample size is generally preferred, imbalances in the distribution between different instance classes can negatively influence the generalization of the model. Therefore, it is common practice to balance the dataset by eliminating, generating or selecting a balanced set of instances from the original data.

Feature engineering aims to create new features by combining and transforming the initial data. In doing so, features can be created that take into account relations between data points that would otherwise have to be captured by the model. An example of feature engineering would be to include a column that aggregates multiple data points.

2.2 DataFrame

DataFrames have become a popular data structure in data analysis and data processing. While they are often viewed as a form of matrix or series, this simplistic view does not take into account the unique place DataFrames take in modern systems. While no singular definition or a clear origin of the term DataFrame exists, many characteristics objects called DataFrame share. These similarities transcend programming languages and systems. Some common characteristics that are shared between different DataFrame implementations are:

Data layout is a key feature of DataFrames, it generally consists of columns and rows. In this aspect, they can be linked to matrices, however the underlying storage structure rarely resembles a matrix.

Typing is an often overlooked feature in languages such as Python or Scala, where Pandas originated. However, most DataFrame frameworks are enforcing strict typing of values stored in the same column, even in languages that are not strictly typed. As such, they provide an additional safety mechanism, making sure columns have a homogeneous data type. However, while columns are homogeneous in type, different columns can be of different types, providing an easy way to assemble complex differently typed data in a single data structure.

Meta data can be critical when manipulating and visualizing data. In most cases, DataFrames not only include the primary data, but additional data that would often be discarded e.g. when creating a standard matrix. Most commonly, such information would be column names and indices.

Abstraction is not the first feature people associate with DataFrames. However, in essence, DataFrames are an abstraction used to manipulate data. The information that is abstracted away in this case in plentiful. But most importantly, DataFrames do not expose the underlying storage data structure to the user. As such, the DataFrame acts as a form of encapsulation, ensuring data integrity at storage level. Furthermore, systems like Modin [36] distribute the stored data across multiple partitions. Spark goes even further and stores its form of a DataFrame across a cluster of several machines [41].

Functionality that is provided by DataFrames, is perhaps the most important factor in their wide adoption. Most frameworks implement a wide array of functions and transformations that can be performed on DataFrames. As such, the DataFrame implementation not only provides the ability to store data, but more importantly, to manipulate it. Furthermore, due to the functions being defined by the DataFrame implementation itself, optimal use can be made of the underlying architecture. Systems like Modin leverage this to operate parallel on multiple DataFrame partitions with the aim to increase performance [36].

2.3 Pandas

Pandas [32] is a DataFrame based data manipulation framework in form of a Python library. It brings many inherent benefits of Python, such as ease of use, rapid development and ease of portability. Pandas is the most popular python data analytics tool and is a widely deployed cornerstone of the Python stack. As a DataFrame based system, it is well-suited to ML tasks, and Pandas DataFrames are seamlessly integrated in most major Python ML frameworks. Over the years, Python, in combination with Pandas and ML libraries, has become somewhat of the default entry into ML applications, evident by the popularity on ML competition websites such as Kaggle.

However, due to its architecture, pandas is not suited to operate on the large data volumnes that are defining today's ML tasks. There are two main technical reasons for this. (1) Pandas does not provide the functionality of object spilling. As such, all data has to be held in memory during the entirety of the processing phase. This leads to a drastic limitation in terms of the size of data that can be handled using the system. While these limitations can be somewhat worked around by batching in some cases, it introduces programming overhead for the developer. (2) Pandas is a purely sequential framework and therefore does not take advantage of the increasing CPU count in modern hardware. This ultimately leads to performance that is insufficient to handle the large amounts of data common in contemporary ML jobs.

2.4 Modin

The shortcomings of Pandas regarding scalability have long been known. As a consequence, multiple projects have taken up the task of implementing frameworks that mirror the Pandas API but do so while tackling the challenges of scalability. One such framework is Modin [36]. Modin aims to provide the complete pandas API, and does so while making use of parallelism. In order to achieve this, Modin leverages existing execution engines to facilitate data distribution and schedules parallel executing jobs. As such, Modin operates as a layer between the Pandas API and the underlying execution engines such as RAY [33] or DASK [39].

Modin's implementation makes use of special classes in order to specialize for specific query execution engines. At its core stands the `QueryCompiler` class and its extending classes. This class translates the pandas API calls invoked on the DataFrame class into executions that can be performed by Modin. As such, Modin breaks down the broad Pandas-API into a smaller set of basic functions that are handled by Modin's own DataFrame class. In some cases, the QueryCompiler class is extended to an engine specific `QueryCompiler` in order to make use of engine specific optimization. In any case, the `QueryCompiler` holds a storage format specific `DataFrame`. This `DataFrame` is responsible for storing the data partitions and to schedule the execution of jobs on the partitions. Figure 2.1 illustrates how the `QueryCompiler` translates the simple call `mean` into a `tree_reduce` call on the underlying `PandasOnRayDataFrame`. In this case, the specific API call `m` ean is translated into a `tree_reduce` job with the parameter `mean`. This represents the mapping by the `QueryCompiler` of a specific function call to a more general set of operations that can be performed on the partitions and where parameters define the operation that is used for the aggregation. In the final step, the `tree_reduce` function on the `PandasOnRayDataframe` uses its `PartitionManager` attribute in order to schedule and distribute the operations, in this case, a maping accross the data partitions. Results of the produced mapping are then aggregated and a new `QueryCompiler` is built using the newly created `PandasOnRayDataframe` and subsequently returned.

Evidently, by the many different supported backends, Modin's architecture is designed from the ground up to enable execution on multiple backends. This fact is even highlighted in Modin's marketing material with the statement ”Bring your backend” [11]. The modularity of the Modin architecture, therefore, offers a potentially easier way of building a Pandas based API than starting from scratch or adapting Pandas itself. Furthermore, Modin is an active project with multiple features in development, one of which being execution on the HDK backend. HDK is an execution engine that is based on Relational Algebra developed by Intel. This difference is execution paradigm is represented in the corresponding `QueryCompiler` class, in this case `DFAlgQueryCompiler`, and could be reused for other projects aiming to bring a relational backend to Modin.

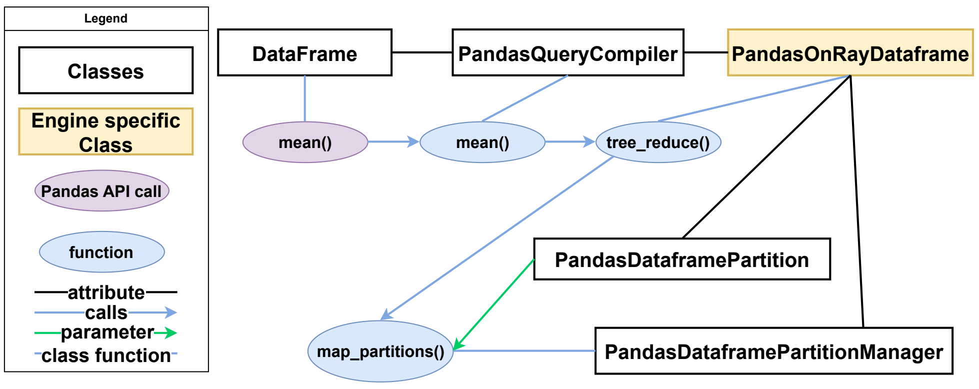


Figure 2.1: Simplified execution graph on Modin using RAY engine.

2.5 Spark

Apache Spark is a popular choice for data analytics and data processing in the realm of big data. Although traditionally not a native Python framework, Spark has gained popularity through the Pyspark module. Pyspark is often chosen when the traditional Python scientific stack can not offer choices that support the required scalability. In addition to its native DataFrame and SQL syntax, Pyspark offers a Pandas API.

Spark's architecture [41] is based on the basic data structure of the resilient distributed data(RDD), representing a set of data that might be distributed across multiple nodes in a cluster. To achieve this, the RDD is split into partitions which are commonly held in memory by the participating nodes. Computation on Spark RDD is performed as a series of transformations. To perform these transformations, Spark creates a direct acyclic graph (DAG) representing the necessary transformations. In a DAG, nodes represent the RDDs, and edges represent the transformations producing a new RDD. These plans are then executed in stages, where each stage contains the parts of the execution that can be performed sequentially on each partition. One such transformation is `map`, which applies a mapping function to each row. After a stage is executed, data has to be distributed between the different nodes of the cluster to enable further computation, for example in the case of aggregation. As such, a transformation like `groupby` will distribute key-value pairs so that each cluster node can perform aggregation on a range of key values. Figure 2.2 shows a visualization of a Spark execution plan and the resulting RDDs. While Spark is designed as a cluster computation framework, its design enables a high degree of parallelism that potentially scales well even on just a single, powerful machine. This, along with the aforementioned Pandas on Spark API, makes it an attractive alternative to frameworks like Modin.

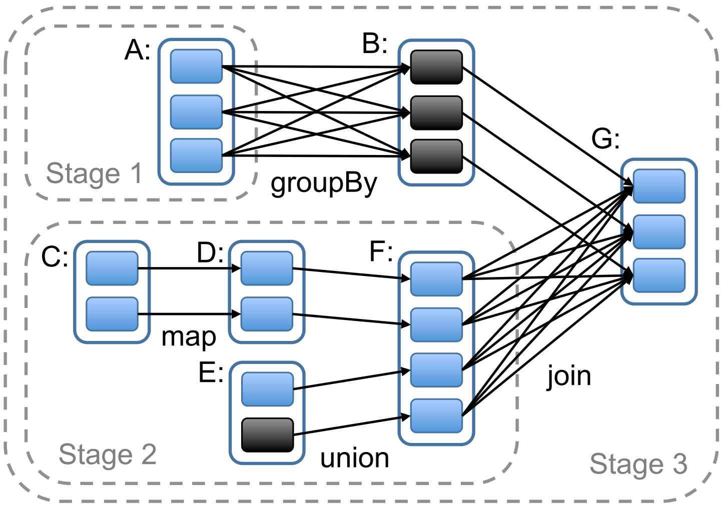


Figure 2.2: Visualization of a Spark execution plan cited from [41]. A, B, C, E, F, and G represent RDDs and their respective partitions.

2.6 Snowflake

2.6.1 Cloud Computing

Since its rapid growth starting in the early 2000's, cloud computing has become one of the cornerstones of today's IT infrastructure. While the elimination of the need for hardware is often seen as the main benefit to its end users, cloud computing has developed to be more than just hardware for rent. As such, cloud providers have taken steps and integrated numerous features into their architectures, enabling consumers to make better and more efficient use of their IT resources.[18] These advantages are plentiful and can differ based on the platform but some notable ones include:

Pricing is a difficult topic in relation to cloud computing. Of course, the fact that cloud computing eliminates the need for hardware is a significant driver of its adoption from the users' perspective. Computer hardware is heavy in initial capital invested and can take a potentially long time to generate profits. Furthermore, correctly assessing one's future hardware needs is complex and, even if dutifully executed, can fail if external factors change. Therefore, cloud computing can not only reduce investment costs but also decrease risk. Additionally, maintaining one's hardware incurs indirect resource demands, such as spatial requirements, among others. Pursuing cloud computing solutions frees up these resources for other opportunities. In contrast, cloud computing operates on a pay-as-you-go structure, where users only pay for the resources they use without any upfront investment besides development costs. Common pricing models will charge customers for execution time or storage space respectively. While the cost of cloud services can sometimes be underestimated, the general consensus seems to be that big cloud providers are competitively priced, due to market pressures [31].

Security is paramount for today's cloud computation customers. As global leading tech enterprises, cloud providers are capable of providing state-of-the-art security in their services. Consequently, cloud computing can free up internal IT resources without degrading security standards. In addition, most cloud providers provide tools that ensure compliance with the necessary legal standards, providing further benefits.

Scalability is a challenge for which cloud computing is uniquely suited. Cloud providers operate massive data centers. Wherever resources are plentiful, a single customer can scale up his own computational and storage needs almost indefinitely without impacting the system. As such, tasks can be quickly scaled out if needed, without costly time penalties. In contrast, a cloud computing system can just as quickly be scaled down if demand dries up. This once again proves as a major selling point, especially for emerging businesses that value flexibility[13].

Integration of a new system is a resource-intensive task for any customer. However, today's cloud providers offer an entire ecosystem of different services designed to work in conjunction with one another. As a result, the development of intensive tasks can be omitted if one can take advantage of the interconnected systems that are already in place.

Reliability is a major selling point of cloud computing services over traditional on-premise solutions. In the case of data storage, many providers enable easy backup systems. Furthermore, the inherent horizontal capabilities of cloud architecture enable fast recovery from node failure through redundancy. Many tools exist that enable proactive monitoring and recovery procedures can be implemented. Additionally, cloud providers maintain multiple sites, and applications can be spread across them. In the rare case of a site experiencing reduced availability, services can be kept alive through other sites[13].

2.6.2 Snowflake: Cloud-Native Data Warehousing

Introduced to the public in 2015, Snowflake is a modern take on the traditional data warehouse, combining a wide array of services in a single cloud computing system. Unlike traditional data warehouse solutions, Snowflake has been designed from the ground up with the capabilities of cloud computing in mind. It maximizes the characteristics of elasticity and ease of use, both often associated with cloud computing. To do so, Snowflake does not operate its own data centers, instead, it is deployed across the infrastructure of the major cloud providers. In certain cases, like storage, this can be helpful with integration, as Snowflake can be given direct access to existing cloud storage locations.

Snowflake is designed as a Software-as-a-Service (SaaS) product. SaaS is a software distribution model that, above all, else prioritizes the reduction of maintenance and monitoring on the side of the user. In the SaaS model, services are provided over the internet and users only interact with the software directly. This means the customer does not have to operate any servers himself. Furthermore, there is no need for installation, configuration, and tuning of the Software to one's systems. The software is hosted on servers by the SaaS provider. In the case of Snowflake, this would be one of the three big cloud providers: Amazon Web Services (AWS), Microsoft Azure (Azure), or Google Cloud Platform (GCP).

Snowflake Architecture

Snowflakes employs a three-layer architecture, in that it divides its operations into a cloud, storage, and compute layer. These layers operate independently, which leads to a separation, allowing individual resources to be scaled up without the need to scale the other layers equally. In this, Snowflake does away with the traditional data warehouse architecture under the shared-nothing model. In the shared-nothing model, each processor node possesses its own disk and has unique ownership over the data stored on it. While this reduces contention between processor nodes, it also leads to linear scaling between compute and storage resources, even when one of the two resources is already adequately scaled for the task at hand. Not only that, but different workloads require different resources for optimal execution. In a shared-nothing architecture, where nodes are typically homogeneous in design and hardware they operate on, this leads to the dilemma where every node must be capable of executing all kinds of tasks. As a result, compromises in node design have to be made to accommodate this variety of tasks. In contrast, Snowflake's separated layer architecture enables the dynamic creation of processing nodes without having to scale other layers equally [23].

Storage on Snowflake follows a different approach compared to traditional database systems. When Snowflake is operated on AWS infrastructure, it uses Amazon S3 (Simple Storage Service) as storage mechanism. On other providers, it uses S3 equivalent technologies. S3 follows the concept of object storage, where objects are uniquely associated with a key and can be retrieved using this key. The basic unit of storage is the bucket. Buckets are unique in name and serve as a prefix for the standard directory naming scheme that defines keys. A bucket can hold an infinite number of objects, while single objects can be up to 5 terabytes in size. Therefore, S3 provides unlimited scaling to the Snowflake warehouse. Interaction with S3 happens through a REST API, implementing the standard operations in the form of `GET`, `PUT` and `DELETE`. In this architecture, Snowflake takes seamless advantage of S3's features, such as the high availability and security. However, while S3 is simple and easily scalable, its design also comes with several downsides that need to be mitigated. Crucially, interaction with S3 over HTTPS in form of the REST API increases latency when accessing storage compared to a local disk. Snowflake employs mechanisms in the Virtual Warehouse layer, such as caching, to reduce these effects. Furthermore, Snowflake makes use of a proprietary storage format to get around S3's limiting file manipulation features. Files can not be edited or appended to in S3, the only operation providing any granularity is the `GET` request that enables retrieval of parts of files. Therefore, Snowflake's storage format partitions the data into large, immutable, horizontally partitioned [14] files and groups values in such a way that they can be retrieved in small packets through the information stored in the file's header. S3 is also used by Virtual Warehouses to store temporary results from large queries. In doing so, out-of-memory and out-of-disk errors can be avoided. This enables arbitrarily large queries.

Virtual Warehouses (VW) are an abstraction used in Snowflake that encapsulates a cluster of EC2 instances. Individual worker nodes that make up a VW are not exposed to the user. This abstraction is a key feature and selling point of Snowflake. Users do not have to think in terms of a cluster of machines but simply interact with a VW of a user-specified size. Users therefore specify the size of their VW by choosing out of a small set of predefined VW sizes, without needing to know the exact number of machines involved.

Virtual Warehouses are purely computational units that can be created, paused, and deleted without changing the state of the database. Individual VWs do not share worker nodes and do not affect each other besides possible contention on the underlying S3 storage. This fact enables each VW to integrate the totality of data stored in the database, which is a critical part of the data warehouse paradigm. As a result, users can issue queries to multiple VWs running in parallel without concerns about different computations interfering with each other.

Snowflake calls their implementation of the Virtual Warehouse an elastic VW and the elasticity is a major selling point of the system. One use case mentioned by the Snowflake team is the fact that this elasticity can drastically improve performance while incurring the same cost. This is because cloud providers charge users for execution time, whether it is incurred sequentially or in parallel. A large query that takes four hours on two machines can potentially be executed in two hours by four machines if the task parallelizes well. In both cases, the total execution time charged is the same, but the execution on four machines performs twice as fast in the user's perspective.

In its elasticity, Snowflake makes the most of horizontal scaling to leverage performance for its users. However, ultimately, cloud providers charge by machine execution time. As such, the cost-effectiveness of a system like Snowflake is directly linked to the performance of its query execution engine. Snowflake uses its own proprietary SQL execution engine, aiming to provide the best price/performance of any SaaS database provider. Just like in storage, the SQL execution engine also uses a columnar data format to make the best use of CPU caches and single-instruction-multible-data (SMID) instructions [14]. Further, the SQL execution engine uses vectorization which not only enables multi-value operations but also increases cache performance. Lastly, the execution engine operates on a push-based materialization system wherein worker nodes push down results to downstream nodes rather than waiting for those nodes to request results.

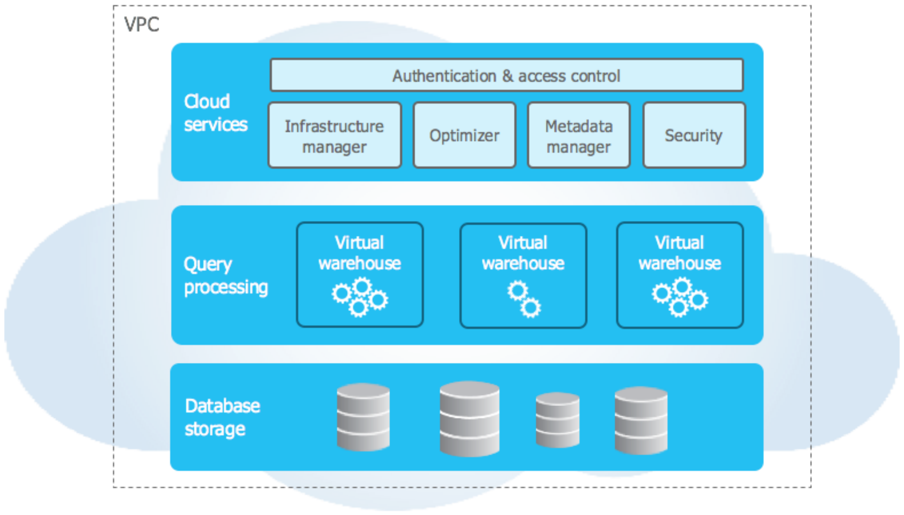


Figure 2.3: Schema of Snowflakes architecture cited from [5]

Cloud Services in Snowflake encompass auxiliary operations not connected to the data. As such, the cloud service layer is in many ways the brain of the Snowflake system. It not only provides the interface through which users interact with Snowflake, but it also manages the underlying systems. Authentication and access control are perhaps the most basic of functions that the cloud service layer fulfills. On the other end of the spectrum, Snowflake handles query optimization in the cloud service layer.

Snowflake's query optimization algorithm is based on a cascading approach [24]. In this method, a logical plan of the query is first created and then optimized using predefined rules. A logical plan in this context relates to a plan expressed in relational algebra and the aforementioned rules transform relational operation into a series of different relational operators that will produce an equivalent query. One commonly used rule for query optimization is called "predicate pushdown". The rule pushes down predicates to the earliest possible point in the query execution; if semantically possible, as early as data ingestion. The upside of this is that table sizes are reduced early in the execution. This in turn reduces the time required for all subsequent queries that, as a result, have to operate on less data. Figure 2.4 illustrates the application of the predicate pushdown rule on a logical execution plan. Many more rules of this type exist, such as decomposition rules, projection pushdown, or splitting rules, to name just a few. In recent years, research has started to develop machine learning as a means of optimizing besides simple rules [30]. However, in the cascading approach, it has to be noted that the optimizer does not take into account the data or its structure during the optimization of the logical plan.

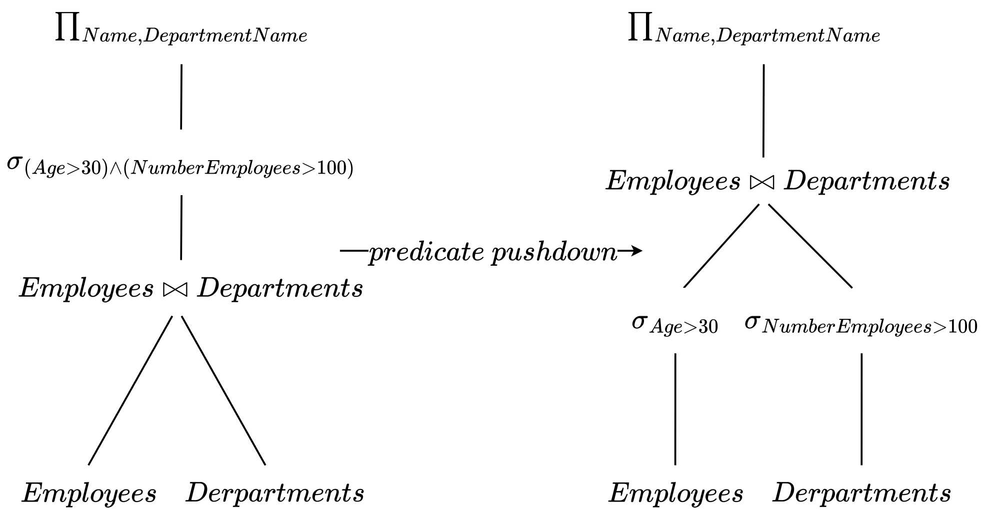

$\Pi_{Name,DepartmentName}$
$\sigma_{(Age>30)\wedge(NumberEmployees>100)}$
$Employees \bowtie Departments$
$Employees$
$Derpartments$
predicate pushdown
$\Pi_{Name,DepartmentName}$
$Employees \bowtie Departments$
$\sigma_{Age>30}$
$\sigma_{NumberEmployees>100}$
$Employees$
$Derpartments$

Figure 2.4: Application of the predicate-pushdown rule on a logical execution plan.

While the first step in cascading query optimization has only taken into consideration the logical plan based on the relational algebra representation of the query, the second step considers all available information, including statistics defining the data, to search the space of equivalent queries to find the best solution. To this end, physical plans are constructed. These physical plans are different from the logical plan in that relational operations are replaced by actual data operations implemented by the execution engine. The same relational operation can potentially be implemented in different ways, each performing best in a specific situation. To identify the best plan among all possible solutions, a metric to rank different plans against each other has to be deployed. A cost model is deployed that associates a cost with each operation in a physical plan. To achieve this, statistics about the data have to be taken into account. Snowflake maintains the necessary data to produce the cost model for data creation and updates. Furthermore, the data schema can influence the cost as well. Especially column types can affect the performance of different operators [22].

The query compilation time includes not only the optimizer but also query parsing. While this process might take up a small fraction of the total execution time for large queries, for small queries it can become a major driver of query execution time and has to be monitored to achieve optimal performance.

Lastly, the cloud service layer takes care of concurrency control. As a data warehouse, Snowflake optimizes for read-intensive tasks rather than transactional operations. Therefore, Snowflake employs snapshot isolation as its concurrency paradigm [21].

Snowflake API

As discussed in the previous section, Snowflake is at its heart a relational database that is optimized for large-scale data analytics tasks. Additionally, it was shown that query execution in Snowflake is done by an engine based on relational algebra. It should come as no surprise that the main way of writing queries for Snowflake is by using SQL. Snowflake's SQL dialect is based on ANSI SQL standard and includes several Snowflake-specific additions, such as functionality to manage VW states. To illustrate SQL syntax used by Snowflake, we can examine a query from the star-schema-benchmark [34] in Snowflake compatible SQL syntax in listing 2.1.

```sql
SELECT sum(lo_revenue), d_year, p_brand1
FROM lineorder, date, part, supplier
WHERE lo_orderdate = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_brand1 BETWEEN 'MFGR#2221' AND 'MFGR#2228'
AND s_region = 'ASIA'
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1;
```

Listing 2.1: SQL code of the SSB query 2.2

Additionally to SQL support, Snowflake also provides a DataFrame API called Snowpark. Snowpark is available for Java, Python, and Scala, and aims to mimic today's data analytics workflows based on the DataFrame abstraction. In essence, the Snowpark API is a SQL wrapper that builds queries iteratively as functions are applied on the DataFrame. Once a function is called that necessitates data materialization on the side of the user, the query representing the DataFrame's state is then executed and the result is returned. This means, that operations on DataFrames are lazily executed. This is a big advantage because at the point the execution is triggered, the query optimizer can holistically optimize the query. For example, the aforementioned SQL query can be rewritten in Snowpark syntax in the following as shown in the listing 2.2.

```
# Creating Snowpark DataFrames
lineorder = session.table("lineorder")
date = session.table("date")
part = session.table("part")
supplier = session.table("supplier")

# Applying DataFrame style transformation
result = lineorder.join(date, lineorder["lo_orderdate"] \
                == date["d_datekey"]
result = result.join(part, lineorder["lo_partkey"] \
                == part["p_partkey"])
result = result.(supplier, lineorder["lo_suppkey"] \
                == supplier["s_suppkey"])
result = result.filter((part["p_brand1"] \
                .between('MFGR#2221', 'MFGR#2228')) & \
                  (supplier["s_region"] == lit('ASIA')))
result = result.group_by(date["d_year"], part["p_brand1"])
result = result.agg(sum_("lo_revenue").alias("total_revenue"))
result = result.order_by(col("d_year"), col("p_brand1"))
```

Listing 2.2: SSB query 2.2 implemented in Pandas API syntax.

Chapter 3

Pandas Workflows

3.1 Overview

Pandas is capable of executing a variety of different workloads, from simple exploratory data analysis (EDA) workflows to complex preprocessing pipelines. The techniques of data preprocessing and its impact on machine learning models have been outlined in earlier sections 2.1. In this section, we want to hone in on the specific techniques that are used by Python developers using Pandas and how they are present in practice. To this end, we have curated a collection of Jupyter notebooks. This collection was then analyzed with the goal of better understanding how Pandas are used in ML tasks. Additionally, we outlined a traditional relational database workload presented by the star-schema-benchmark (SSB) that focuses on complex analytical queries to test relational database systems in terms of performance.

3.1.1 Star-Schema-Benchmark

SSB is a commonly used benchmark in the performance analysis of relational database systems [34]. It consists of a total of 13 queries that are executed on data that is stored in a star-schema on the relational database. While these types of queries are not commonly implemented through Pandas, it makes sense to examine how Pandas can be used for these non-traditional workloads. SSB is a modification of the ADL benchmark that has not only been used to compare different relational database systems but lends itself as a benchmark for different languages and implementations [37].

3.1.2 ML Preprocessing

Pandas is a commonly used data preprocessing tool of the Python stack. Together with a variety of ML frameworks, it offers a great amount of functionality while being easy to use. As such, it has become the default tool employed in the ML teaching and competition environment. However, Pandas does not translate well to real-world applications, where, in the context of ML preprocessing, scale is the major factor of concern. Large-scale ML preprocessing is currently mostly performed on specialized systems using clusters of machines, in order to achieve the high throughput necessary in today's applications [16]. Pandas do not integrate well into such systems, not only because of their lackluster performance but because the execution model introduces a high amount of data movement. Especially in the case of online preprocessing, one can take advantage of relational databases combined storage and computation capability to bring the operations directly to the data stored in a cloud-native system. Combining the popular Pandas API with a SaaS database system such as Snowflake could bring Snowflake's benefits of durability, scalability, and performance as a relational database to new workloads such as ML preprocessing.

3.2 Data Preprocessing on Pandas

Data preprocessing takes many forms. Here, we outline some of the Pandas syntax and semantics that can be used to achieve this.

Data ingestion in Pandas is performed through various reader functions. Pandas DataFrames can be built directly from virtually all popular data storage formats. Additionally, Pandas DataFrames can also be built from Python primitives and many other modules that implement conversion from their own data structures to Pandas DataFrames. Some examples include Dask, Modin, or Koalas [3, 36, 6]. Furthermore, modules exist that enable Pandas to read directly from object stores. One such module is s3fs. It enables Pandas to read directly from AWS S3 using the unique object ID:

```
#Reading directly from S3 into pandas dataframe
df = pandas.read_csv("s3//example-bucket/prefix/data.csv")
```

This is especially useful in today's environment where data size is constantly increasing and local storage is at a premium. In reverse, writing to S3 is of course also possible, eliminating the need for local storage.

Data transformations as discussed in 2.1, include a variety of tasks. Here, we will outline how Pandas is used in conjunction with common tools in the Python ML stack.

After the data has been ingested into pandas, the preprocessing phase begins. Data can be cleaned using a variety of functions. For example, the following script will fill in missing values in the `Prize` column with the `me`⌋ `an`, drop duplicate rows, and cast the values in the `Age` column to `Integer`:

```
df['Prize'].fillna(df['Prize'].mean())
df.drop_duplicates(inplace=True)
df['Age'] = df["Age"].astype('int')
```

Typically, computation will continue with feature engineering. To this end, the pandas API provides many functions. In this example, we first create a new column add up to columns, and then split a string into multiple components.

```
df['Sum'] = df['Income'] + df['Gifts']
df[['First Name', 'Last Name']] = df['Name'].str
                                    .split(' ', expand=True)
```

Machine learning relies on models, and once the data is preprocessed, these models are defined in ML frameworks such as SKLEARN. Models in most popular Python ML frameworks can work directly with Pandas DataFrames, eliminating the need for conversion.

3.3 Kaggle Competitions

Machine learning competitions have become a major factor in practical education for machine learning. Not only do universities include them in their courses, but there is a wide variety of competitions available to the general public. These competitions are used by trained professionals to push the envelope of ML capabilities, but also by individuals who want to train their ML skill set. One of the most well-known competitions of this kind had been the Netflix prize [17]. Announced in 2006, Netflix offered a prize of 1 million US dollars for anyone who could achieve a 10% improvement in accuracy over their recommendation algorithm. The prize was claimed in 2009, marking the competition as a success. Since then, ML competitions have surged in popularity, leading to the creation of several websites that host competitions. One such website is Kaggle. Competitions can generally be created by anyone, though the most popular ones are often organized by Kaggle itself or by companies that offer a cash prize.

Submissions are typically done using notebooks written in either R or Python [2]. These notebooks can be made publicly available and a large base of users does so. As such, Kaggle offers an extensive collection of Jupyter notebooks implementing various ML tasks.

The most popular competition on Kaggle has long been ”Titanic - Machine Learning from Disaster” with over 15000 teams [12]. In this competition, the goal is to predict the survival of passengers on a vessel based on an assortment of data.

3.3.1 Notebook Collection

Notebooks on Kaggle take many forms. Not all of them contain code relevant to the insights we want to gain. To increase the percentage of notebooks that include workflows of interest, we limited ourselves to notebooks that are associated to feature engineering using Kaggle's search function. We implemented a web scraper in Python using the selenium browser automation tool [9]. The web scraper operates in two steps. During step one, the scraper traverses the results starting from the search landing page and iterates over the pages. In doing so, it collects links to submissions that contain Jupyter noteboooks. In the second step, we start multiple parallel tasks that download the notebooks from the collected links. Through this process, a total of 2708 notebooks were collected.

3.4 Workflow Analysis

3.4.1 Preprocessing

Jupyter notebooks in the format `.ipynb` do not lend themselves well to analysis. Therefore, we translate the notebooks into Python scripts. This is done using `jupyter nbconvert` command. For a part of the notebooks, the conversion fails. This leaves us with a total of 2222 notebooks for analysis.

The Python module Abstract Syntax Tree (AST) can be used to create syntax trees for Python programs. These trees contain information about the structure of the parsed program. We are primarily interested in the nodes containing function calls as these nodes contain the name of the called function and its parameters. This allows us to detect function invocations that could be Pandas calls. It has to be noted that for Python as an untyped language, it is not possible to know the object type the function is called on. This would require complete execution of the code due to Pandas' dynamic typing system. This is beyond the capability of the AST module. This means, that in any case where other imported modules contain functions with the same name as a Pandas function, we can not differentiate which module the call has to be associated with. The Pandas functions we consider are those exposed by Pandas, as well as those that can be performed on DataFrames.

With our analysis, we aim to answer two questions:

1. What are the most used Pandas functions?

2. How are function calls distributed over Pandas' complete available API?

3.4.2 Results

3.1 shows the distribution of the 25 most used Pandas functions. The three most used functions, `read_csv`, `head` and `to_csv`, are functions related to data loading, storage and inspection.

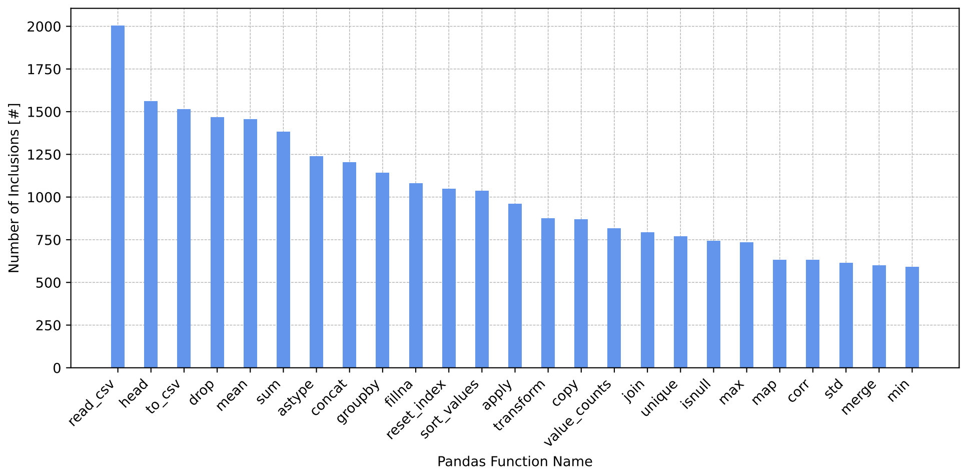


Figure 3.1: Number of inclusions of functions in notebooks the collected notebooks. If a function is used multiple times in a single notebook, we count this as a single inclusion.

Other than these three, all functions that made the list are DataFrame transformations. We can observe some patterns here. Many of the functions can be categorized into groups. `DataFrame.min`, `DataFrame.max` and `DataFram`⌋ `e.sum` fall into the category of aggregation. A further category consists of statistical functions such as `DataFrame.std`, `DataFrame.corr`, `DataFrame.mean`. Another category consists of functions that combine DataFrames. These are `DataFrame.concat`, `DataFrame.join` and `DataFrame.merge`. Lastly, `DataF`⌋ `rame.apply` and `DataFrame.map` serve the same functionality, although the `DataFrame.map` function is depreciated.

In figure 3.2, we illustrate the distribution of all calls associated with the Pandas API. It follows roughly an exponential distribution. Of the total 154884 calls associated with the Pandas API, 136832, or 88.344 percent are made up of 25 percent of all functions.

3.4.3 Discussion

The results are promising in two ways. (1) We have seen that several functions among the top 25 functions used can be categorized into groups that perform similar operations. This helps us in two ways. Firstly, functions that share the same operation structure can likely be implemented together. That means the architecture to execute one can be reused for other functions in

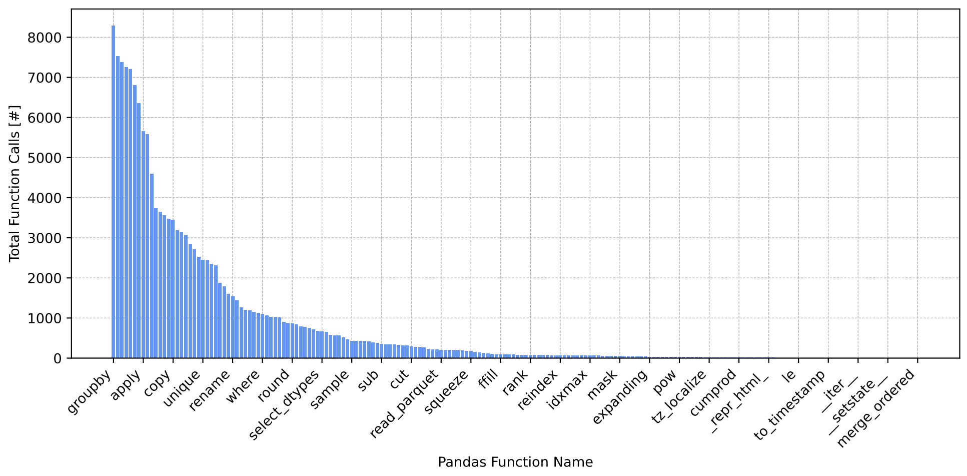


Figure 3.2: Distribution of all Pandas-associated calls, where each function call is counted by itself. A single notebook can therefore contribute multiple calls of the same function.

the group. This is the case for aggregations. Secondly, not all functions need implementing since some aliases exist. (2) The distribution of all Pandas calls over the Pandas API indicates that a disproportionately high amount of Pandas use cases can be achieved with only a small subset of Pandas functions. This is promising for any system aiming to provide an alternative Pandas API since it indicates that one does not have to implement the whole API to provide a system that can perform most real-world Pandas tasks.

Chapter 4

Implementation

4.1 Overview

In this work, we set out to implement a Pandas API based on Snowpark. In doing so, we intend to create a system that can leverage the seamless scaling of the Snowflake virtual warehouse architecture, in a simple-to-use Python environment. To achieve this we extend an existing Pandas API in such a way that it defers execution to Snowpark where possible. This approach follows the paradigm of scaling out, rather than the traditional approach of using specialized hardware commonly used in today's large-scale preprocessing pipelines. This shift in approach has shown to be promising already [16] and scaling out via the Snowflake VW follows a similar idea. Different existing projects could be used as the base for such an implementation. Most notably, Pandas and Modin, as they are well developed due to their popularity and are easily modifiable due to their open-source nature. We analyzed the different architectures of Pandas and Modin in search of parallels between Snowpark's API functionality and the frameworks interface. As a result, we decided to base our implementation on the Modin framework. Modin, as mentioned in 2.4, is purpose-built to enable modifications that allow the use of an alternative execution engine. We decided to take exactly this approach. The Snowpandas code base is publicly available as a Github repository forked from Modin at [1].

In section 2.6.2, we discussed how the Snowpark API is essentially a wrapper that iteratively builds SQL queries. As such, Snowpark DataFrames are lazily executed, which enables the query optimizer to operate on the query holistically. Our implementation is designed to preserve this lazy execution wherever possible. To achieve this, we use a tree data structure that tracks transformations on Snowpark DataFrames. The information stored in this fashion is then used in scenarios where the Pandas API uses DataFrames

[1] https://github.com/YvesRobinK/modin.git

as arguments in transformations of DataFrames. This is necessary because Snowpark does not support the kind of column and row-based assignments of a traditional DataFrame. Further tracking and replaying executions in places where Pandas would assign a Series to a DataFrame's column avoids the need for joins. This is because as a SQL-based framework, Snowpark has to join tables before columns can be used in the same expression.

Our implementation is based mainly on three classes that interact with one another. The `SnowparkDataframe` is the main class, implementing the state of the DataFrame. It is mainly responsible for tracking the state and does not directly manipulate the underlying `snowpark.DataFrame`. Transformations on the `snowpark.DataFrame` are performed by the `Frame` class, as each `SnowflakeDataframe` has a `Frame` attribute and the `snowpark.DataFrame` is an attribute of this `Frame`. As such, the `SnowflakeDataframe` controls transformation by calling functions on the `Frame` class with the appropriate parameters. A `Frame` will return a new `Frame` with the transformed `snowpark.DataFrame` as its attribute. Subsequently, a new `SnowflakeDataframe` is created with this new `Frame`. Additionally, the new `SnowparkDataframe` is assigned a transformation specific `Node` that represents the last transformation performed on the DataFrame. One such specific `Node` would be the `AggNode`. `AggNode` represents an aggregation and as such will include the parameters needed to perform the aggregation. Most notably, this would be the `agg_dict` that is used as a parameter in the `snowflake.snowpark.DataFrame.agg` function.

While the Modin framework lends itself well to modifications, there are some features of the implementation that do not translate well to the concepts of the `snowpark.DataFrame`. One such concept is Modin's use of indexing. Classes like `_LocationIndexerBase` are implemented on the assumption that the underlying storage supports index-based operations. In such cases, we have to use one of two mechanisms in order to proceed. In the first case, (1) we implemented dummy functions in order to skip over checks or transformations that might be produced in such a way. These functions return arbitrary values and thus steer the execution path in the way. Where this approach is not suitable, we had to resort to option two. (2) We make use of Python's capability of introspection in order to skip parts of implementations that can not reasonably be replicated.

4.2 Snowpandas

In this thesis, we set out to design and implement a system that implements the Pandas API using Snowpark as its backend. We named the resulting implementation Snowpandas. Based on Modin, we similarly aim to create a system that is as close to a drop-in replacement for Pandas as possible. Besides some additional syntax, this holds true for our implementation. Once all necessary dependencies are installed within the Conda environment, Snowpandas can be used without any additional resources.

4.2.1 Syntax

The standard Pandas API does not provide the functionality needed to define connection parameters and the desired Snowflake configuration. Therefore, users specify their credentials and the desired configuration via environment variables. These are implemented as part of Modin's `config` utilities and can be set in the following way.

```
#Snowpandas configuration

import modin.pandas as pd
import modin.config as cfg

snowflake_credentials = {
                   "account": "<snowflake-ID>",
                    "user": "<username>",
                    "password": "<password>",
                  }

cfg.SnowFlakeConnectionParameters.put(snowflake_credentials)
cfg.SnowFlakeDatabaseName.put(<database-name>)
cfg.SnowFlakeWarehouseName.put(<warehouse-name>)
```

We elected to implement non-Pandas native syntax for the initial DataFrame creation, because we deemed it unreasonable to use an existing IO method to return an object that does not follow the official documentation. Thus, we implemented the `from_sf_table` function, that creates a DataFrame object based on the Snowpark API. This leads to a simple one-line syntax not unlike the usual `read_csv` or `read_parquet` functions:

```
#Creating a Snowpandas DataFrame
lineorder = pd.from_sf_table(tablename="SSB_SF1.LINEORDER")
date = pd.from_sf_table(tablename="SSB_SF1.LINEORDER")
```

This function returns a `DataFrame` object that implements the standard Pandas API. Figure 4.1 illustrates the architecture of the created DataFrame. From this point on, no special syntax is used, as Modin will seamlessly trigger materialization by calling `to_pandas` where necessary. Once `to_pandas` is called, a Modin DataFrame is created from the Pandas DataFrame produced by Snowflake. This allows computation to continue using the Modin implementation without any extra steps.

4.2.2 Architecture

A DataFrame created by `from_sf_table` follows the basic structure of a any Modin DataFrame. Figure 4.1 illustrates this initial structure of a Modin DataFrame and Snowpandas specific classes are highlighted in yellow. There are three main classes that are Snowpandas specific, these are `SnowflakeDataframe`, `Frame` and `Node`. The `SnowflakeDataframe` class is the central point around which our implementation is constructed. It also acts as the singular interface with the Modin implementation and implements the API that is used by Modin's translation layer. This also includes the various attributes that are accessed by the `DFAlgQueryCompilter`. Some of the fields that are implemented by the `SnowlakeDataframe` include `columns`, `index` and `dtypes`. On the other hand, we have the attributes related to the Snowpark API, namely `sf_session`, `Frame` and `Node` which are all critical in the interaction with Snowpark, even though the latter are not directly Snowpark objects.

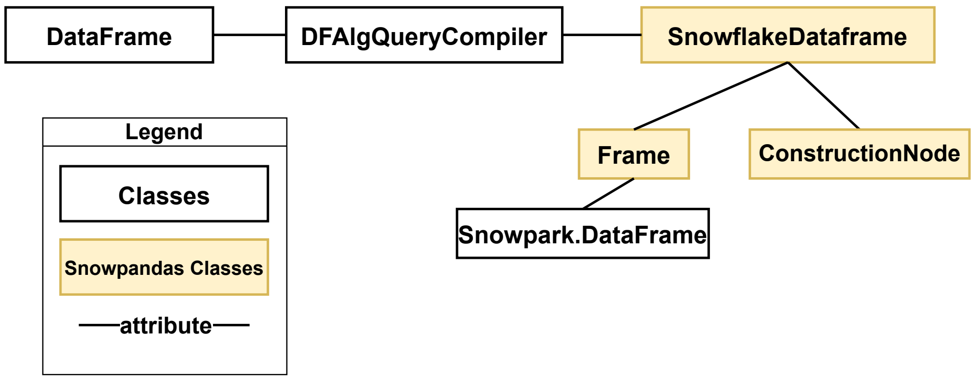


Figure 4.1: Basic architecture of a DataFrame after construction.

SnowflakeDataframe

The `SnowflakeDataframe` implements a lazily executed DataFrame, while eagerly applying transformations to a `snowpark.DataFrame` representing the underlying data. The interface used by the 'DFAlgQueryCompiler' is implemented here. Through it, Modin's translation layer manipulates the data. Thus, the `SnowflakeDataframe` serves as the center point of the Snowflake-based backend for Modin. A further functionality implemented by the `SnowflakeDataframe` is managing the DataFrame's metadata such as `dtypes`, `columns` and `index`.

Frame

The `Frame` class acts as a wrapper for the `snowpark.DataFrame` that represents the data stored in the virtual warehouse. This makes the `Frame` class essentially a second layer of translation, it maps functions and parameters from Pandas API like syntax to the Snowpark API. Here, we show the implementation of filter operations of the form:

```
df = df.filter.loc[df["Age"] > 18]
```

To perform this operation, we make use of the `op_tree` attributes of the DataFrame passed as an argument. In this case, the class of the passed `Node` determines the execution within the `filter` function. Specifically, we differentiate between whether the DataFrame was last transformed by a comparison or a logical operation. The differentiation leads to the differing function calls on the `snowflake.DataFrame`. In the case of a comparison, we simply apply the same predicate to the `snowpandas.DataFrame`. If the parameter DataFrame was created by a logical operation between two DataFrames, we need to fetch the information about both predicates. These predicates are then applied in the `filter` function in a singular expression defined by the logical operator. Use of the Snowflake API from a syntactic point of view differs from case to case. The first case is handled by using the standard function definition, in the second case we resort to dynamic code generation to maximize ease of implementation and code reuse. The exact implementation can be seen in listing 4.1. As such, we construct a `command_string` that we dynamically evaluate at runtime.

```
OPERATORS = {
    "<=": "<=",
    ">=": ">=",
    "=": "==",
    "<": "<",
    ">": ">"
    }
LOGICAL_OPERATORS={
    "or": "|",
    "and": "&"
    }

def filter(self,
           comp_Node: Node = None
           ):
    """
    Performs filtering on a snowpark.DataFrame
    Parameters
    ----------
```

```
comp_Node : Node              | op_tree defining the filtering
Returns
-------
Frame                         | Transforemed DataFrame
"""
if isinstance(comp_Node, ComparisonNode):
    new_frame = self._frame /
                        .filter(f'"{comp_Node.comp_column}" '
                        f"{comp_Node.operator} "
                        f"'{comp_Node.value}'"
                        )
elif isinstance(comp_Node, LogicalNode):
    left_comp = comp_Node.prev
    right_comp = comp_Node.right_comp
    command_string = (f"self._frame. \
          filter((col(\"{left_comp.comp_column}\")"
          f" {OPERATORS[left_comp.operator]} "
          f"'{left_comp.value}') "
          f"{LOGICAL_OPERATORS[comp_Node"
          f".logical_operator]} "
          f"(col(\"{right_comp.comp_column}\")"
          f" {OPERATORS[right_comp.operator]}"
          f"'{right_comp.value}'))")
    new_frame = eval(command_string)
return Frame(new_frame)
```

Listing 4.1: Implementation of the `filter` function of the `Frame` class.

Node

Nodes are the basic unit used to create the tree data structure that makes up the field `SnowflakeDataframe.op_tree`. This tree logs the transformations that are applied to a DataFrame throughout its existence. As discussed in the previous section 4.2.2, this information is used to perform transformations on the `DataFrame` the function is called on. This is necessary because, in the Pandas API, arguments to functions like filter are generally of class `Series` and, as such, we do not have direct access to the original values if we were not storing them. Further, this allows us to eliminate the need to perform joins as we will see later on. To this end, the `Node` classes generally store at least the parameters required to perform the corresponding `snowpark.DataFrame` function.

```
JoinNode
├── name:str
├── other:Node
│   ├── name:str
│   └── ...
├── own_colnames:[str]
├── other_colnames:[str]
├── colnames:[str]
├── prev:Node
└── frame:snowpark.DataFrame
```

For all `Node` objects, this includes the attributes `name:str` of the node, the corresponding `frame:Frame` and the parent `prev:Node`. In the specific case of the `JoinNode` illustrated here, the operation specific attributes are the `other:Node` the join is performed with, as well as `own_colnames:[str]`, `o⌋ ther_colnames:[str]` and the new `colnames:[str]`.

4.2.3 Translation

Figure 4.2 is a UML representing the execution of a single Pandas API call. Invoked on the `DataFrame` class, the initial call gets immediately deferred to the `_stat_function`. There Modin validates the function arguments and passes the execution on to its `DFAlgQueryCompiler`. At this point, we have passed through Modin's API layer and entered the translation layer implemented by the `DFAlgQueryCompiler`. `DFalgQueryComputer` is an HDK-specific class that we reuse for our implementation because HDK is also a relational engine, operating in a similar paradigm to Snowflake, as we will see. The `mean` call on the `DFalgQueryCompiler` serves only to implement the interface with with the API layer and the call is passed on to _agg. In doing so, it passes on the name of the aggregation function (here "mean") as an argument. This is a critical step as it reduces the initial broad space of aggregation functions into a single method. As a result, we only have to implement a single function for aggregation in the `SnowflakeDataframe`. This function agg gets called by the `DFAlgQueryCompiler`, and the name of the aggregation function is again passed as a parameter. At this stage, we have entered the Snowpandas-specific part of the implementation.

In `SnowflakeDataframe.agg` parameters are parsed in order to invoke the correct `Frame` function with the corresponding parameters. Initially we make a case distinction between `axis=0` and `axis=1`. In the case of `ax⌋ is=0` we need to perform column-wise aggregation and to do so we first need to extract the names of the columns that are of numeric type. This is because, in the Pandas API, columns of non-numeric type are dropped when performing aggregation with a function that takes only numeric values. On the other hand, Snowpark will produce an error if we attempt to

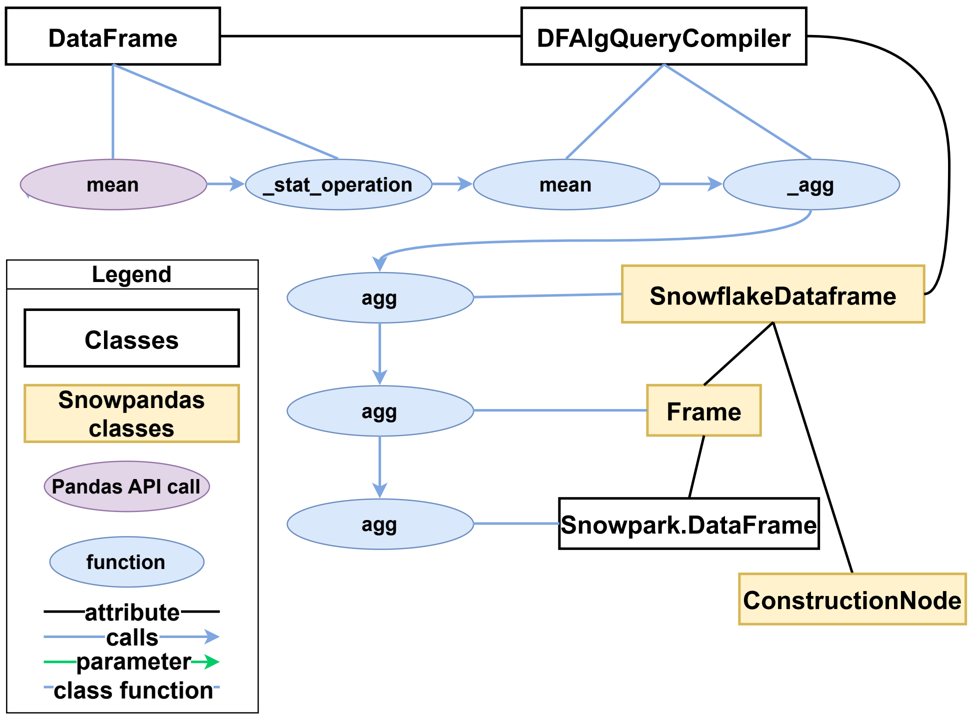


Figure 4.2: Execution of the Pandas API `mean` function with a SnowflakeDataframe.

perform aggregation on columns that are non-numeric. Therefore we have to specifically compute the numeric columns at this point. We do so by using `snowpark.DataFrame.schema`, Snowpark manages the metadata such as column types locally, and therefore this can be done without breaking the paradigm of lazy execution. Once we have selected the numeric columns of the DataFrame we create a `dict` that will serve as the argument to the Snowpark API call. The implementation of `Frame.agg` is very simple, unlike other `Frame` functions a single Snowpark API call will suffice. As such the total implementation consists of a single line:

```
return Frame(self._frame.agg(agg_dict))
```

Once `Frame` returns a new `Frame` object that encapsulates the manipulated `snowpark.DataFrame`, a new `SnowflakeDataframe` is constructed. Most notable in the construction of this new `SnowflakeDataframe` is the new `Node` object, representing the transformations performed in order to produce the corresponding `snowpark.DataFrame`. In the case of `agg`, we construct an `AggNode` which stores the information needed to perform the aggregation, namely the `agg_dict`, and construct a tree by providing the `SnowflakeDataframe`'s own `Node` as an argument. This field becomes necessary in operations that perform column assignments.

4.2.4 Column Assignments

The Pandas API is in many ways built around the mechanism of column assignments. This mechanism though is not simply replicated in a relational database, because in order to combine two tables in a database, a join is needed. Consequently, much of the Pandas API syntax does not have a direct equivalent in the Snowpark API. The following are some Pandas DataFrame manipulations used during ML preprocessing that illustrate this form of operation:

```
#Preprocessing with Pandas
self.train[["Deck", "Num", "Side"]] = self.train['Cabin'] \
                                     .str.split('/', expand=True)
self.train["SumSpends"] = self.train[col_to_sum].sum(axis=1)
self.train['Total_Billed'] = self.train['RoomService'] \
                                     + self.train['FoodCourt']
```

All of these functions make use of column assignments. This is no problem for a system that eagerly executes expressions. The Pandas implementation can simply materialize the expression result and replace the current column data. However, in Snowpark, this is not possible. The Snowpark implementation, based on SQL-like operators, would achieve the same by doing the following in order to implement the binary + operation:

```
#Snowpark binary operation between independent DataFrame
res_df = df1.join(df2, df1["id"] == df2["id"]))
res_df = res_df.with_column("Total_Billed"), \
             (col("RoomService") + col("FoodCourt")))
```

However, this implementation introduces two main issues that are problematic in terms of performance. (1) Just like in normal SQL, Snowpark DataFrames have to be joined in order to perform operations between two columns of different DataFrames. Joins are among the most resource-intensive operations. Avoiding them whenever possible is crucial for maintaining efficiency. (2) In order to perform a join between two DataFrames, we need a unique column that can be used as the value to perform the join on. This column needs to be the same in both DataFrames, leaving us with two options to implement it. One approach is to keep a unique column of the DataFrame throughout all operations. However, this requires the user to specify the column explicitly at some point, as nothing prevents the unique column from being dropped or not selected in an operation. While feasible, this increases the DataFrame size, which could lead to performance penalties. Another way is to dynamically create a column that matches in both DataFrames. This approach, however, leads to computation that has to be performed at runtime. This would again lead to a decrease in performance.

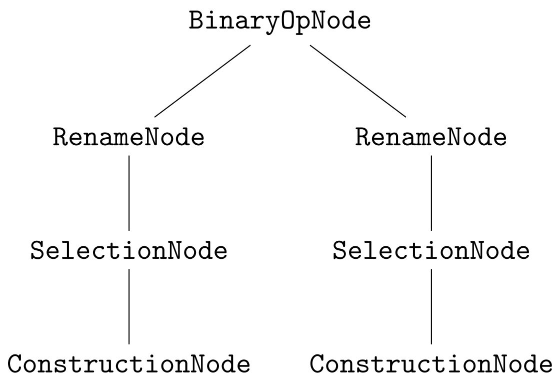
Faced with this problem, we had to design an implementation that achieves the same semantics without storing intermediate results in separate DataFrames. As such, we devised a system that tracks the transformations performed on the DataFrame or series and enables us to replay these operations on the DataFrame that is subject to the column assignment. We illustrate this process based on the following Pandas API code snipped:

```
#Python API column assignment of binary operation result
df['Total_Billed'] = df['RoomService'] + df['FoodCourt']
```

This would be the right side of the assignment statement.

```
#Pandas API binary operation
df['RoomService'] + df['FoodCourt']
```

The binary operation of the Pandas API will produce a `Series` object, as can be seen in Figure 4.3. Just as in the case of the `mean` function, discussed earlier in this section. The operation is handed down through the layers and a single function of the `SnowflakeDataframe` implements the different binary operations based on the `op` parameter passed. The critical point, however, is not the manipulation of the `snowpark.DataFrame` inside the `Frame`, but the information stored in the resulting `SnowparkDataframe` objects `op_tree` field. In this case, this will be a `BinaryOpNode`, the state of which at this point of execution is as follows:



With the completion of the expression evaluation, `_setitem` is called on the original `DataFrame`, with the parameters `key` and `value`. The parameter `key` is of type string, specifying the column to which we are assigning the value and the `value` parameter being of type `Series` as produced by a binary operation between two `Series`. The execution is passed through the upper layers much the same way as described before, and finally arrives in `SnoflakeDataframes` as a call to `setitem`. The `setitem` function parses the arguments in order to defer to the correct `Frame` manipulation. In our example, since the `key` is not already in the columns of the `SnowflakeDataframe`, the column

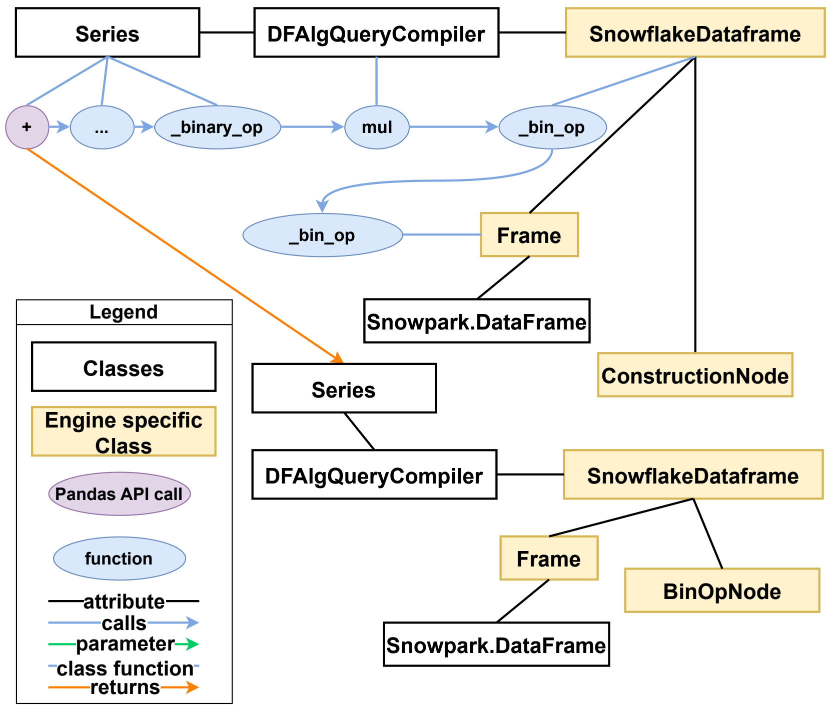


Figure 4.3: Execution of the Pandas API + operator with a DataFrame of class `SnowflakeDataFrame`.

name is added to the DataFrame's columns. The resulting call will include the `new_column` parameter to create the column in the Snowpark DataFrame. A further distinction is made based on the type of the `value`. In the example case, the type is `DFAlgQueryCompiler`, as the upper layers have stripped away the API layer class `Series`. This results in the execution path leading to the call:

```
#Create a new Frame with transformed data
new_frame = self._frame.assign(
                    new_column=key,
                    op_tree=value._modin_frame.op_tree
                    )
```

Notice that the `snowpark.DataFrame` of the original value is not within the transitive closure of the `Frame` on which the `assign` function is called. All the information needed to perform the transformation on the original DataFrame is present within the `BinaryOpNode`, and we do not need the actual `snowpark.DataFrame`. As a result, within the `assign` function, we can perform the equivalent binary operation and assignment on the original DataFrame without having to perform a join.

```
#Simulating column assignment
```

```
left_col = op_tree.left.prev.colnames[0] # fetch column name from
    #selectionNode of the left side argument to the binary operation
right_col = op_tree.right.prev.colnames[0] # fetch column name from
    #selectionNode of the right side argument to the binary operation
new_frame = self._frame.with_column(new_colname, \
            (self._frame[left_col] + self._frame[right_col]))
```

4.2.5 Interface and Deference

As mentioned in 2.4, Modin was chosen as the base framework for our implementation because it is purpose-built to enable operating with different backends. In practice, this means only a few classes need to be adapted to bring a different backend to Modin. In our case, these classes are the `SnowflakeDataframe` and its auxiliary classes `Frame` and `Node`. Therefore, by implementing the interface with the `DFAlgQueryCompiler` and implementing fields such as `dtypes`, we provide a large amount of functionality that the upper layers interact with. However, some functionalities are unsuitable for our implementation that relies on Snowpark's lazy execution scheme. The most notable functionality of this kind used by Modin is indexing. Since in the local paradigm that Modin expects, it is unproblematic to select data via a mask or slicing. However, Snowpark does not support such operations, and creating indexes based on the data itself would force materialization, at least in part.

We employed two strategies to skip or defer execution in the upper layers where it was not possible to implement the necessary functionality in the interface implemented by `SnowflakeDataframe`. Firstly, in any case where the upper layer tries to access an attribute of the `SnowflakeDataframe` that we can not reasonably implement. Most often, this is because we do not have the necessary information due to the lazy execution scheme and can therefore not return a meaningful value. Secondly, in some cases, the implementation of the functionality would introduce a disproportional amount of engineering effort and, as such, we can not implement this part of the interface. In such cases, we resorted to one of two methods to continue. (1) Whenever we encountered such an issue, we first attempted to create a dummy function that returns a value designed to direct the execution in the upper layer as intended. These dummy functions come in two flavors. Firstly, some dummy functions return a hard-coded value; an example of this is the `_has_unsupported_data` which simply returns `False`. In other cases, we return unrelated fields that are then passed back down in a place that is of benefit to our implementation. (2) In cases where we could not reasonably implement a dummy function or attribute, we resorted to modifying the upper layers to skip over parts of execution that are not implemented by `SnowpandasDataframe`. These modifications largely rely on introspection, checking the type of the QueryCompilers `_modin_frame` field. In most cases, this takes the form of including an `isinstance` evaluation within a conditional statement to force the execution to either enter or bypass a particular block of code.

4.2.6 Types

A critical feature of Pandas and other systems implementing the Pandas API is the typing system. As described in section 2.2, for example, the feature column-level type safety is an important characteristic of modern DataFrames. For our implementation, this introduces a problem since Snowflake, and in extension Snowpark, do not support the same typing system. Furthermore, the Python implementation of Snowpark uses its typing system, which differs from Snowflake's. This is especially critical since the API layer and translation layer of Modin require typing information to correctly translate API calls to engine-level execution. Fortunately, the Snowpark data types are stored as part of the schema on the local representation of the `snowpark.DataFrame` and are managed by Snowpark implementation as transformations are performed. This allows us to retrieve the types of columns stored in the `snowpark.DataFrame` in isolation of the data. Therefore, we can simply retrieve the types upon creation of a `SnowflakeDataframe` from its `snowpark.DataFrame`. However, as mentioned, these types are part of Snowflake's proprietary typing system [4] and must first be mapped to their corresponding `numpy.dtype`. Table 4.2 shows some of the relevant mappings.

Category	**Snowflake Data Type**	**Numpy Dtypes**
String Types	StringType	numpy.dtype('O')
Numerical Types	DecimalType	numpy.dtype('float64')
	IntegerType	numpy.dtype('int64')
	LongType	numpy.dtype('int64')
Logical Types	BooleanType	numpy.dtype('bool')

Table 4.2: Mappings applied to Snowflake Data Types

The only case where a mapping from Numpy Dtypes to Snowflake Data Types is needed is in the implementation of the `astype` function. This function effectively casts the relevant column to the Dtype specified in the Pandas API. We handle this directly in the `Frame.astype` function, where the relevant columns of the `snowpark.DataFrame` are cast to the specified type. Since the schema of the resulting `snowpark.DataFrame` is used to initialize the `dtypes` field of the resulting `SnowflakeDataframe`, it will automatically be constructed with the `dtypes`.

4.2.7 Fillnan Problem

During the implementation of the `fillna` function of the Pandas API, we encountered early execution in a way we had not observed before. To understand why we encountered this problem, we must first understand the relevant Pandas syntax.

```
#Filling missing values in columns of numeric types
for col in numeric_columns.columns:
    mean = df[col].mean()
    df[col] = df[col].fillna(value=mean)
```

In Pandas syntax, we iterate over the columns that need filling and compute the value used as a parameter in the `fillna` call from the series object representing the corresponding column. Replicating this execution scheme in Snowpark, however, will lead to early execution. This is because the `na.fill` function on `snowpark.DataFrames` does not support the passing of a lazily executed object such as a `snowflake.DataFrame`. As such, an implementation making use of the already existing agg functionality will not work.

```
#Naive Snowpark implementation
mean = self._frame.select(colname).agg({colname: "mean"})
new_frame = self._frame.na.fill(mean)
```

This implementation will eagerly evaluate the expression of `mean` to pass the resulting scalar value to the `snowpark.DataFrame.na.fill` function. As a result, the query representing the DataFrame computation on Snowflake is split into multiple sub-queries that are executed sequentially. This is problematic for performance, firstly, because it means that the query optimizer can no longer optimize the query holistically. Secondly, due to the sequential nature of these operations, we are reducing parallelism, while potentially increasing overhead.

We conceived of an optimization to work around this problem. However, we had to take a shortcut by creating a new Pandas syntax, due to the considerable engineering effort necessary to implement our optimization for the already existing syntax. We did so by adding an option for the `method` parameter in the Pandas API `fillna` function. These two new options are `"snow_mean"` and `"snow_mode"`, representing the operation of filling the missing values with the column mean or mode respectively. As a result, the new syntax does not make use of the `df.agg` function, thus not triggering early execution. Now we are back in the same paradigm as described in section 4.2.4, where we dealt with the translation of column assignments. While we can generally copy the mechanism used for column assignments, we still need a way to avoid using the `df.agg` function to compute the scalar value needed. This can be achieved without much difficulty by using SQL expressions. The Snowpark API provides a `select_expr` function, and we can include the column function that represents the scalar value as an SQL expression. Unlike the `agg` function, the SQL expression provided in this way is directly inserted into the SQL statement representing the `snowpark.DataFrames` state. In practice, the Snowpark API call achieving this is relatively simple:

```
#Lazy fill nan, with mean
new_frame = self._frame.selectExpr("*",
                        f"COALESCE({assign_col}, \
                        AVG({assign_col}) OVER()) \
                        AS {assign_col}")
```

Chapter 5

Experiments

5.1 Experimental Setup

5.1.1 Workloads

To analyze our implementation in relation to differing workloads, we conducted experiments on two different datasets and three workload types to our experiments. The first component aims to test our systems' performance on workloads that are traditionally well-suited to relational databases. The second component is made up of microbenchmarks that enable a better understanding of the performance of specific operators. The third workload reproduces real-world ML preprocessing tasks that are based on ML pipelines sampled from Kaggle.

Star-Schema-Benchmark

The star-schema-benchmark (SSB) [35] is a popular benchmark, which tests the performance of database systems in executing SQL queries. The name star schema describes a specific way in which the relational schema of the data is constructed. In a star schema, a large central table (fact table) serves as the only connection between tables. This divides the schema into fact and dimension tables. Figure illustrates the organization of the SSB schema. Note that dimension tables are magnitudes smaller in size than the fact table. This type of schema is often used where highly analytical workloads are expected and aims to optimize for query execution [40]. It does so by reducing complex joins by simplifying the relation between the fact table and dimension tables. The scale factor increases the row count per table. The number of rows increases linearly with the scale factor for all tables except the `PARTS` table, where the row count increases logarithmically. As can be seen in 5.1.1, the row count for scaling factor 1 is 30'000 rows for `CUSTOMER`, 2'000 rows for `SUPPLIER`, 6'000'000 rows for `LINEORDER`, 200'000 rows for PART and 7 years worth of dates for the DATE table.

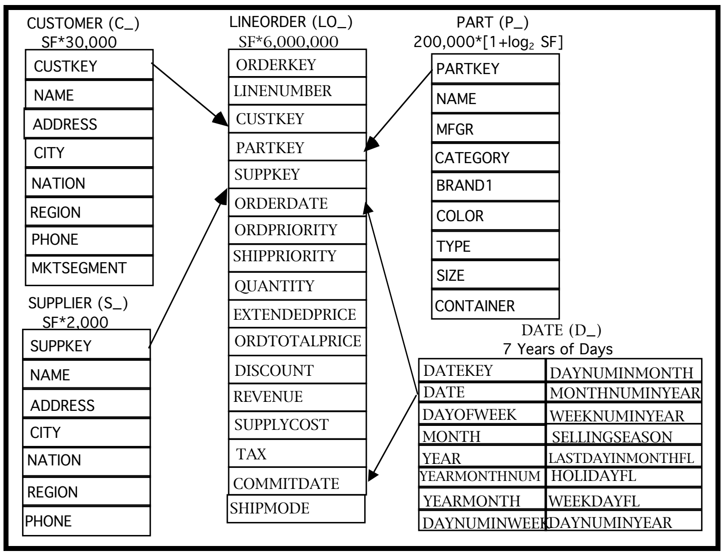


Figure 5.1: The SSB schema as cited from the paper [35].

Additionally to defining a data schema, SSB also defines a set of queries. These queries are grouped into four flights, containing three or four queries each. These four groups aim to test for a variety of common query patterns to enable users to assess their database's performance according to the tasks they expect to perform. While different groups aim to perform similar structural operations, the queries inside a group are designed to distinguish themselves in specificity. In total, 13 queries are making up the benchmark. Figure shows the SQL code for query 3.1 of the star-schema-benchmark. These queries are then evaluated on four different scaling factors 1, 10 , 100 and 1000.

```sql
SELECT c_nation, s_nation, d_year,
    SUM(lo_revenue) AS revenue
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdate = d_datekey
AND c_region = 'ASIA'
AND s_region = 'ASIA'
AND d_year >= 1992
AND d_year <= 1997
GROUP BY c_nation, s_nation, d_year
ORDER BY d_year ASC, revenue DESC;
```

Figure 5.2: SQL code for the SSB query 3.1

Microbenchmarks

To better understand the performance of the different frameworks during the SSB queries, we conceived a series of microbenchmarks. These microbenchmarks break down the SSB queries into their specific operations. These operations being `selection`, `filter`, `sort`, `groupby`, `binaryOperation` and `Join`. This resulted in the following code for the `selection`. The experiment setup was essentially the same as with the SSB experiment. However, the execution for the scaling factor 1000 was omitted. This is because the microbenchmarks are less selective and therefore, the resulting DataFrames for scaling factor 1000 would have been more than 1 TB for some of the benchmarks. The resulting network overhead would lead to unreasonably long execution times.

```
--SQL syntax
SELECT lo_commitdate, lo_discount, lo_orderpriority FROM lineorder;

#Pandas syntax
df = df[["LO_COMMITDATE", "LO_DISCOUNT", "LO_ORDERPRIORITY"]];
```

and the *Join* benchmark respectively:

```
--SQL syntax
SELECT * as FROM date, customer,
    supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey;

#Pandas syntax
df = lineorder.set_index('LO_ORDERDATE') \
        .join(date.set_index('D_DATEKEY'))
df = df.set_index('LO_CUSTKEY') \
        .join(customer.set_index('C_CUSTKEY'))
df = df.set_index('LO_PARTKEY') \
        .join(part.set_index('P_PARTKEY'))
df = df.set_index('LO_SUPPKEY') \
        .join(supplier.set_index('S_SUPPKEY'))
```

ML Preprocessing Pipelines

To test our implementation on ML preprocessing pipelines that resemble real-world applications, we looked to Kaggle. As discussed in section 3.3, Kaggle is an ML competition platform. One of the competitions hosted on Kaggle is named ”Spaceship Titanic” [10]. This competition is the spiritual successor to the most popular competition in Kaggles’ history by the name of ”Titanic - Machine Learning from Disaster” [12]. In the ”Titanic - Machine Learning from Disaster” competition, the goal is to predict the survival of passengers based on the provided data. To the same effect, in ”Spaceship Titanic”, the goal is to predict if a passenger reaches his destination. In both cases, a binary variable `[0,1]` has to be predicted. The task falls into the field of supervised learning. This means we have example data for which we know the ground truth. From this, we have to train a model that can predict the outcome for samples where the ground truth is not known. As such, two datasets are provided to competitors. The `train.csv` file contains the training data and the `test.csv` contains the data upon which predictions have to be made. The data consist of 13 columns that are `PassengerID`, `HomePlanet`, `CryoSleep`, `Cabin`, `Destination`, `Age`, `VIP`, `RoomService`, `FoodCourt`, `ShoppingMall`, `Spa`, `VRDeck` and `Name`. The `train.csv` file additionally contains the field `Transported`, representing the ground truth.

We searched the publicly available submissions of Jupyter notebooks to the ”Spaceship Titanic” competition for notebooks that would be suitable for our experiment. As discussed in section 3.2, these notebooks are often divided into distinct phases. For our experiments, we are solely interested in the preprocessing phase. However, in some cases, this preprocessing phase is interwoven with other frameworks that make a clear separation impossible. For example, in instances where models such as SKlearns `SimpleImputer` are used early on in the preprocessing pipeline, their inclusion would force the materialization of the data. Such an operation contrasts with the paradigm of lazy execution, which is performance-critical for our system. To address this, we only included notebooks in our experiment that start with Pandas-based preprocessing and end the portion of the pipeline included in the experiment, once such an invocation would force materialization. While this may seem like an advantage for Snowpandas, in reality, in a real-world scenario, the developer could compensate for this by refactoring the code to be more favorable to Snowpandas. Although we could also take this route, doing so might introduce a bias from our side into the experiment. Thus, we only selected notebooks with a clear separation between the preprocessing phase and other parts of the pipeline.

Snowflake Virtual Warehouse Sizes

Early on in section 2.6.2 we alluded to the fact that selecting a bigger Snowflake VW size can increase performance in terms of execution time, while potentially incurring the same cost if execution time decreases in inverse proportion to the prize. We wanted to test this hypothesis by executing the largest scaling factor of the ML preprocessing experiment on four different warehouse sizes. We chose the small, medium, large and xlarge warehouse sizes. The costs of the warehouses of our choice are 2, 4, 8, and 16 credits per hour respectively. We note that for the hypothesis to hold, execution time would need to halve every time a VW size is increased by one step, since the price doubles for each step.

Additionally to the prizing hypothesis, this experiment can provide some insight into how well the queries are parallelized and how much the execution resources increase depending on the chosen warehouse size.

5.1.2 Cloud Setup

EC2 Instance

We executed all experiments on AWS cloud infrastructure. An AWS EC2 [1] instance builds the basis of all our experiments. The EC2 virtual machines were of type `m5d.12xlarge`. These machines possess 48 CPU cores (Intel Xeon Platinum 8175), 192GB of RAM, 32 GB of integrated storage, and two 900GB NVMe SSDs. At a cost of around 4 USD per hour, these general-purpose machines are situated towards the upper end of the available machine size. However, the cost of these machines has been selected to be roughly equal to those incurred by our selected Snowflake VW size. As such, both services are roughly equally priced per hour of run time.

We created a virtual Python environment with Anaconda, which included all the necessary dependencies. Additionally, the EC2 instance was configured with Snowsql and the AWS CLI. All of our services are located in the AWS `eu-central-1` region. It is important to collocate them in the same region to reduce latency and increase the data transfer speed. The NVMes are mounted to a single directory `/data` using `mdmda`. This directory is not permanent and is only used for object spilling by the Modin and Spark engines.

Snowflake

We conducted all of our experiments on Snowflake VWs of size `SMALL`. Initially, the data is stored in a public S3 bucket. The files are of the table (`.tbl`) or column-separate-values (`.csv`) file formats. To make this data available to Snowflake to create Snowflake native tables, we first needed to integrate Snowflake with S3. This is achieved by creating an AWS role that can be assumed by the Snowflake integration stage. This integration is created on the Snowflake side and will later serve as the access point for S3 buckets. It defines the buckets that can be accessed through the integration. Amazon's access control scheme is based on policies that are linked to roles. We defined a policy that grants access to the required buckets and assigned it to the AWS role linked with Snowflake. The linkage is done by providing AWS with the ID and access keys of the S3 integration from Snowflake. Once this process is complete, we can directly load S3 data from our bucket into Snowflake.

The table creation process was done via Snowsql commands. The process not only loaded the data from the files but also defined the database schema. The original files did not include header and type information, it was therefore critical that we defined those in this step. Further, we created the necessary warehouses needed. All of this was done using a bash script that pipes parameters into a SQL template. This created a temporary SQL command that can be executed through Snowsql. This automated the process for different scaling factors. The process works the same for both SSB and "Spaceship Titanic" data, only the SQL template has to be changed to represent the differing data schema.

As described in 4.2.1, the connection parameters for Snowflake are provided to Snowpandas through extended Pandas syntax. This was done using a Python `dict` object that specifies `account ID`, `username` and `password`. To ensure runs happened in isolation, for all executions on Snowflake both data caching and result caching were disabled.

Amazon Simple Storage (S3)

The data used by Pandas, Modin, and Spark was also stored in the cloud. We used S3 to do so. The original data was read from its source by a Python script that scales the data according to the scaling factor provided as a parameter. To make file handling easier, the `LINEORDER` data is partitioned into smaller files. For the higher scaling factors of the SSB, data chunks of 10 million lines were read and saved to the S3 bucket as a single file under the prefix `/SF1/lineorder/`. Such a file is about 1 GB in size. Additionally to the `.csv` file format, we also stored the data in parquet `.parquet`. Contrary to the CSV file format, Parquet is a file format developed by the Apache Foundation [8] that is optimized for big data applications. It implements columnar data storage which has become a key feature of today's big data applications. Another key feature is its support of schemas, seamlessly storing metadata such as typing and column names. Parquet files are not only considerably smaller than CSV files but can also be processed more efficiently due to the optimizations described. In our case, it means that a partition of 10 million rows of the `LINEORDER` table is just 316 MB in size. For our application, this means that it decreases network overhead and the time necessary to read the files. We performed the SSB benchmark on both data formats. However, for the other experiments, we limited ourselves to parquet.

5.1.3 Engine Configurations

Both Modin and Spark, but especially Modin, claim to provide well-performing out-of-the-box solutions. Our experience has been considerably different. Testing has been plagued with crashes and errors of different varieties. The fact that this happened even on our well-spaced testing machine is somewhat surprising. It took us a considerable amount of time to configure the setting and fine-tuning of the systems so that consistent testing could be undertaken.

Modin

The first problem encountered when scaling Modin to bigger datasets was out-of-memory (OOM) errors. These occur in the underlying execution engine, in our case RAY. The RAY memory monitor will kill tasks that use more than a specified percentage of their resources. And it will retry the task after a back-off time. Turning off the memory monitor as described by the RAY docs [7] did not resolve the problem, but led to complete crashes compared to just performance degradation. Why these problems occur can not be conclusively answered. Especially since we used object spilling to `/data`, which should have been more than big enough to hold all the data. The counter-intuitive solution to this problem has been to initiate RAY with more cores than the machine has virtual CPUs. As such, we arrived at a configuration where we initiate RAY separately before starting Modin-related operations and specifying the number of cores as 48. This is exactly the opposite approach from what we tried early on, by reducing the number of cores used by RAY, under the rationale that each core could be assigned more memory. We tried this both implicitly and explicitly, meaning we specified the exact amount of memory per worker or we reduced the `ncpu` count specified in the RAY initialization. Both did not prove to resolve the problem. So we resorted to the above-described solution.

Another problem we encountered was the inability to shut down RAY manually between different experiments. The RAY engine possesses a `shutdown` function purpose-built for testing, which should allow us to clean the state of RAY. However, we did not succeed in using this functionality in combination with Modin. This is because Modin seems to keep references to the shutdown RAY processes. This could not be solved by reimporting the module or other means. After fruitless troubleshooting of this issue, we had to resort to running the experiments through Bash scripts so that the entire Python interpreter would be reset between experiments.

Spark

While Spark proved to be much easier to configure in terms of performance, it seems to be much better at adjusting to the local machine's resources. Problems regarding the Java runtime were plentiful. These problems were of two kinds. (1) After the simple installation through `pip install pyspark`, the environment variables would not be set correctly. To solve this, we used the Findspark module. Findspark is a Python module that is imported before Spark is initialized, and it will make sure Pyspark will be supplied with the correct paths to the necessary resources. (2) The second problem originated from the fact that we are storing our data remotely on S3. Our initial installation did not include the correct binaries needed for Spark to interact with S3. As such, we had to initialize the Spark session with additional configuration parameters that specify the inclusion of the needed Hadoop-aws libraries as can be seen in figure 5.1. In term of resource configuration, we started spark with 180GB of `--driver-memory`, 6GB of `--executor-memory`, 48 `--executor-cores` and set the correct spill location `spark.local.dir`.

While Pyspark provides a Pandas API, this API does not support reading from S3. Because of this, we first have to read the data from S3 using standard Spark syntax. After the data is loaded correctly, we can then use the `pandas_api` function on the resulting RDDs. This will return DataFrame objects that implement the normal Pandas API. As such, we initialized Pyspark with the code in listing 5.1.

```
#Pandas syntax
import findspark
        findspark.init()
        import pyspark.pandas as pd
        from pyspark import SparkConf
        from pyspark.sql import SparkSession

        os.environ['PYSPARK_SUBMIT_ARGS'] = \
        '--driver-memory 180g \
        --executor-memory 6g \
        --executor-cores 48 \
        --conf spark.local.dir=/data \
        pyspark-shell'

        conf = SparkConf() \
            .setAppName("Connect AWS") \
            .setMaster("local[*]")

        conf.set("spark.jars.packages", \
             "org.apache.hadoop:hadoop-aws:3.3.2")
    spark = SparkSession \
        .builder \
        .config(conf=conf) \
        .getOrCreate()
    spark._jsc.hadoopConfiguration() \
                            .set("fs.s3a.access.key", \
                                "<aws-access-key>")
    spark._jsc.hadoopConfiguration() \
                            .set("fs.s3a.secret.key", \
                                "<aws-secret-key>")
```

Listing 5.1: Initialization and configuration of Pyspark.

5.2 Results

5.2.1 Star-Schema-Benchmark

5.2.2 Procedure

We measured the performance of the SSB on five different systems: Pandas, Modin, Pyspark, SQL, and our own Pandas API with the Snowpark backend. We will call our implementation in this section Snowpandas. Pandas serves as the baseline against which all other implementations will be compared. Modin is a direct competitor to Pandas and will show what a modern implementation, purpose-built to scale the Pandas API to greater data volumes, can achieve. Spark has long been a cornerstone of data warehousing. Therefore, it is a very mature system and shows what a system, that is locally executed, can achieve. We also measure the performance when the original handwritten SSB queries are directly executed with Snowsql. This will serve as the baseline to compare our Snowpandas implementation. So that we can deduce how much overhead our implementation introduces during query execution. The smaller the delta between the handwritten queries and Snowpandas, the better, as it indicates minimal inefficiencies. We are interested in two forms of inefficiency that could be introduced by the Snowpandas implementation. (1) Query compilation time will tell us whether the SQL queries constructed via Snowpark can efficiently be parsed and optimized by Snowflake. This is because queries constructed by the Snowpark API are visibly more complex in terms of their structure. Such queries often consist of several levels of nested expressions. It has to be shown that this does not impact query compilation time on Snowflake. Secondly (2) we need to analyze whether the query execution time on Snowflake increases when queries are generated by Snowpandas. It is possible that the query optimizer, discussed in 2.6.2, can not perform as well on these more complex nested queries, resulting in increased query execution time. This is especially important for bigger scaling factors since data volume increases exponentially.

In all cases, we are interested in the time between the query start and the time when the result is available. This leads to a key distinction between local executions and executions on Snowflake. Since in the case of local execution, data must first be downloaded. In this case, we measure the download time and the execution time for the query. In the case of execution on Snowflake, the data is directly accessed by Snowflake. Thus, we do not have to account for download time. However, as mentioned before, for Snowflake executions, we measure both the execution time and the query compilation time. While one might argue that this paradigm favors executions on Snowflake, such as the handwritten queries and Snowpandas, we argue that in the context of offline preprocessing and the fact that we are already storing the data in cloud services, bringing the data locally for Snowflake execution would be impractical. In such a scenario, it is reasonable to assume that the resulting data would be stored either by Snowflake directly or on AWS S3. Therefore, this way of measuring can be seen as disadvantageous to Snowflake executions, since we do not include data upload times that would not apply to Snowflake executions. Lastly, for local executions, we include experiments executed with `.csv` and `.parquet` files. Not only because it influences the download time, but also because systems might be able to benefit from Parquets optimized data format.

5.2.3 Results

Figure 5.5 shows the results of SSB for scaling factor 10. The performance of the handwritten SQL and Snowpandas is roughly equal for all queries, for both the query execution time and the query compilation time. This shows that the implementation of Snowpandas does not introduce a decrease in performance compared to the handwritten queries. Furthermore, the performance of both Snowflake executions is almost a magnitude faster than the next best-performing system in the form of Pyspark. Pyspark outperforms Pandas and Modin by a magnitude. Most surprisingly, Modin's performance has been consistently the worst among all systems tested. This is surprising as Modin's parallelism should be able to make use of the machine's resources to speed up the execution, just as Pyspark does. The fact that Modin is even outperformed by Pandas indicates that its parallel execution introduces overhead that can not be compensated for by parallelism.

Scaling factor 1000 visible Figure in 5.6 shows that Modin and Pandas did not scale to the largest scaling factor. Neither system managed to perform any queries under the 1100-second runtime cutoff for the scaling factor 100. Pandas, unsurprisingly, failed not only due to exceeding the cutoff time but also due to a lack of memory in some queries. For scaling factor 100, the Pandas DataFrame is about 153GB in size. Although this is theoretically small enough to fit into memory, crashes occurred. Modin, on the other hand, has the functionality of object spilling and does not encounter any memory issues from scaling factor 100. However, it has to be noted that even the effective total of 1.78TB of NVMe storage would not have sufficed for Modin to execute the scaling factor of 1000 queries. We observed that Modin spills more than 1.5TB to disk for some queries, including large joins at scaling factor 100. Additional figures completing the experiment measurements can be found in section A.1.2.

Lastly, a noticeable difference in download time can be observed between Parquet and CSV executions. This is as expected due to the smaller file size. Additionally, a clear trend can be observed in execution times between queries that read from Parquet or CSV files. The Parquet execution consistently outperforms the execution reading from CSV files even when excluding download time.

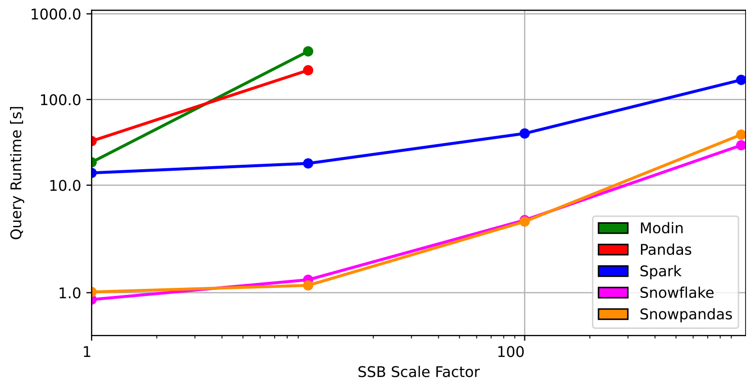


Figure 5.3: Query runtime depending on scaling factor for SSB query 2.1

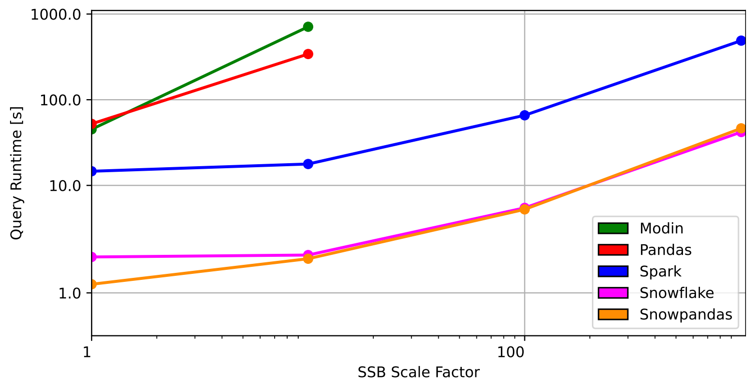


Figure 5.4: Query runtime depending on scaling factor for SSB query 4.1

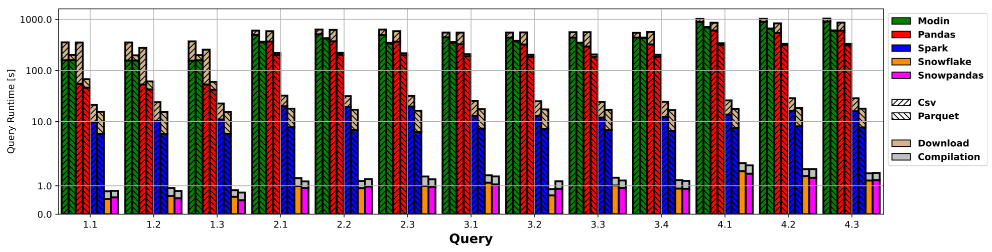


Figure 5.5: This figure shows the query runtimes, download time, and compilation time for all systems for scaling factor 10 of the SSB.

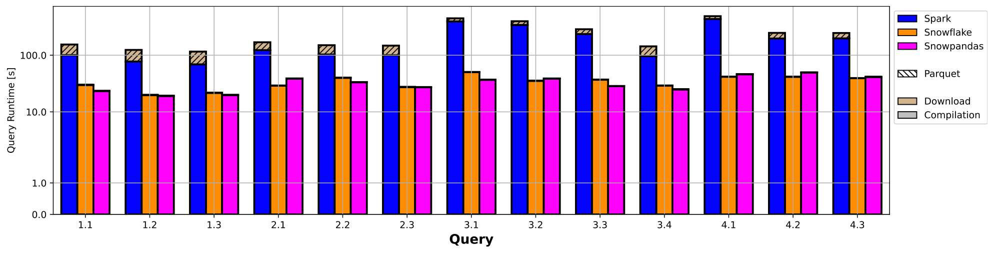


Figure 5.6: This figure shows the query runtimes, download time, and compilation time for all systems for scaling factor 1000 of the SSB.

5.2.4 Microbenchmarks

5.2.5 Procedure

The microbenchmarks were evaluated for Pandas, Modin, Spark, and Snowpandas. Since the SSB tests have shown a performance increase when Parquet is used as the file format, we decided to only evaluate the microbenchmarks with Parquet. With the microbenchmark, we aim to isolate function-specific performance, which is why we did not include download time in our analysis, but exclusively focused on execution time.

5.2.6 Results

Figure illustrates the performance of all tested systems when performing a `filter`. Snowpandas outperforms all other systems for all scaling factors, with Pandas, Modin and Spark having a multiple of Snowpandas' execution time. Besides scaling factor 1, this trend holds for all operations except one: for the join operation, shown in figure 5.8. Spark outperformed all other systems by a large margin for this single operation. The `join` was only performed for scaling factor 1 and 10 since execution times had been unreasonably high. Figure 5.9 shows the results of the `groupby` experiment, Snowpandas once again being the best performing system.

Modin performed significantly better in the microbenchmarks compared to the SSB, mostly outperforming Pandas. This is much more in line with the performance expected of Modin and seems to indicate that Modin is sensitive to query complexity.

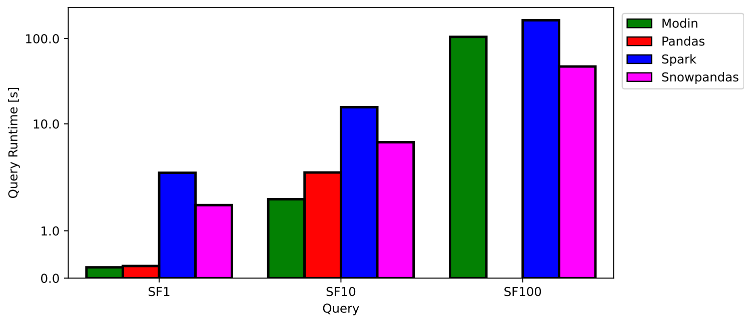


Figure 5.7: Performance of the `filter` micro benchmark.

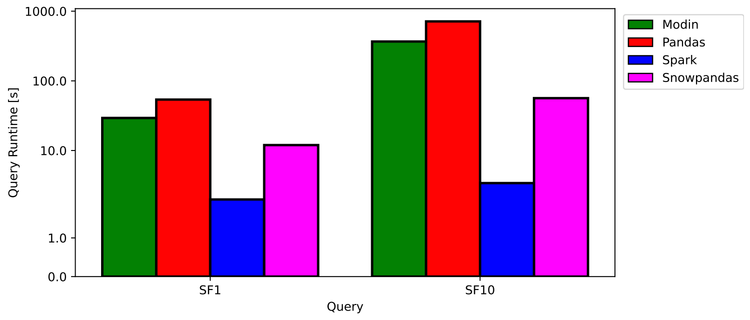


Figure 5.8: Performance of the `join` micro benchmark.

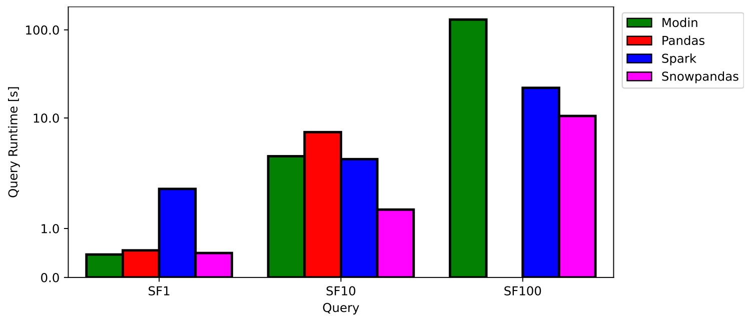


Figure 5.9: Performance of the `groupby` micro benchmark.

5.2.7 ML Preprocessing Pipelines

Procedure

As mentioned before, this experiment focused on real-world workloads in the form of Jupyter notebooks collected from Kaggle. From these original notebooks, we separate the preprocessing logic and test it much in the same way as the micro benchmarks. In this case, we also elected to exclude download times, which will generally benefit any local executions, for the reasons mentioned in section 5.2.2. This isolated the data manipulation part of the execution.

The data used in the ”Spaceship Titanic” competition consists of two CSV files. The training dataset consists of 8693 rows, with 14 columns each. The testing dataset consists of 4277 rows, with 13 columns. This data was then scaled by the scaling factors 10, 100, 1000, 10000, and 100000. The data stored in CSV files for the biggest scaling factor was 77GB in size and 6GB for Parquet. These values are not representative however, since Parquet compressed the data heavily, a Pandas DataFrame constructed from these files is about 350GB in size. Once again, we used S3 to store the files. This experiment was only conducted using Parquet files, due to our earlier results showing that using Parquet increases performance. For Snowpandas, the data was staged into tables just the same way as in 5.2.2.

The first notebook we tested is based on a fork of a popular introduction notebook [38] as such we will refer to it as ”Fork” here. This resulted in the code displayed by listing 5.2. The extracted preprocessing pipelines from the Endeavor and CatBoost notebook can be found in the appendix, in listing A.2 and listing A.1 respectively.

```
#Drop superficial columns
self.train.drop('Name', axis=1, inplace=True)
self.test.drop('Name', axis=1, inplace=True)

#Replace boolean
self.train = self.train['Transported'] \
                        .replace("False", 0, inplace=True)
self.test = self.test['Transported'] \
                        .replace("True", 0, inplace=True)

#Split Cabin into distinct values
self.train[["Deck", "Num", "Side"]] = self.train['Cabin'] \
                        .str.split('/', expand=True)
self.test[["Deck", "Num", "Side"]] = self.test['Cabin'] \
                        .str.split('/', expand=True)

#Define expenses columns
expenses = ['RoomService', 'FoodCourt', 'ShoppingMall', \
            'Spa', 'VRDeck']

#Aggregate expenses
self.train["SumSpends"] = self.train[expenses].sum(axis=1)
self.test["SumSpends"] = self.test[expenses].sum(axis=1)
```

Listing 5.2: Extracted preprocessing logic from the Fork notebook.

We selected two additional Jupyter notebooks to test, these have been selected in a way that they build upon the first one. The second notebook we tested can be found on Kaggle by the name of ”Spaceship Titanic Higher Score Endeavor” [20], here we will call it ”Endeavor”. The most notable functionality added by this notebook is the use of `fillna` function. Often competition notebooks will make use of some form of `Imputer` class to fill in missing values, though we deemed it critical to test our implementation to perform this task as well. In this specific case this takes the form of the following code:

```
#Fill in numeric columns
for col in numeric_tmp.columns:
        print(col)
        df[col] = df[col].fillna(value = df[col].mean())
#Fill in categorical columns
for col in categ_tmp.columns:
    print(col)
    df[col] = df[col].fillna(value = df[col].mode()[0])
```

The third and last notebook of the name ”Titanic spaceship feature selection catboost” [28], here referred to as ”CatBoost”. Aside from the need of the `astype` function, this notebook is largely a permutation of the first two notebooks. This is unavoidable because the preprocessing pipeline of Jupyter notebooks within the same competition turned out to largely make use of the same transformations. Especially in terms of feature engineering, notebooks within the same competition follow similar approaches.

5.2.8 Results

Figure 5.10 shows the performance of Pandas, Modin, Spark, and Snowpandas in executing the preprocessing pipeline extracted from the Fork notebook. This is also the only notebook that we managed to execute with Modin, for all others its execution time surpassed the time cutoff set at 30 minutes. It has to be noted that Modin defaults to standard Pandas during the execution of all 3 notebooks since Modin on the RAY engine does not support the `str.split` function, which is prominently used in all notebooks. Pandas similarly failed to perform the execution for scaling factors 10000 and 100000 due to surpassing the maximum execution time or due to out-of-memory errors.

As we can see in Table 5.1 Snowpandas execution time for the Fork notebook is, starting from scaling factor 1000 is at most half of all other engines.

	Execution Time Mean [s]			
Engine	Modin	Pandas	Spark	Snowpandas
Scaling Factor				
1	0.786	**0.030**	5.494	0.564
10	1.005	**0.187**	5.472	0.939
100	3.383	**1.911**	10.195	4.062
1000	21.134	18.694	17.638	**5.917**
10000	229.797	-	62.481	**31.221**
100000	1607.300	-	506.860	**220.654**

Table 5.1: Measured execution times for the Fork preprocessing pipeline.

This significant performance advantage does however not hold for the other two notebooks. In Figure 5.11 the performance for the CatBoost pipeline is plotted. We can see that starting from scaling factor 1000 Spark starts to outperform Snowpandas. Sparks poor performance for the lower scaling factors does not come as a surprise. The overhead of creating the Spark environment as well as the overhead related to the distribution of data across multiple nodes is something we have observed in all experiments. This makes Spark especially suited for large data volumes where this overhead becomes proportionally less important. From the numeric data in table 5.2 we can deduce that the performance is more or less reversed when compared to the Fork pipeline. Spark now being the engine that performs the task in half the time of Snowpandas.

	Execution Time Mean [s]			
Engine	Modin	Pandas	Spark	Snowpandas
Scaling Factor				
1	0.627	**0.092**	2.901	0.755
10	3.737	**0.426**	2.894	1.357
100	29.329	**5.566**	5.673	5.914
1000	100.078	56.719	**7.856**	10.531
10000	-	-	**23.768**	42.264
100000	-	-	**165.458**	338.006

Table 5.2: Measured execution times for the CatBoost preprocessing pipeline.

The Endeavor notebook introduced a new function of the Pandas API in the form of `fillna`, this introduced a problem we had not encountered so far. As discussed in section 4.2.7 we faced a problem with early materialization, while our new approach managed to prevent this, the query profile did show that our new implementation does not alleviate the problem. Instead of the early materialization, Snowflake computed the necessary values column aggregations sequentially. This chain of `WindowFunctions` made up a total of 76.4 percent of the total execution time for the largest scaling factor. As displayed in table 5.3, for the total execution time of 1145.824s this is a total of 875.409s. As such the calculation of the values within the SQL `COALESCE` clause alone, exceeds Sparks execution time.

	Execution Time Mean [s]			
Engine	Modin	Pandas	Spark	Snowpandas
Scaling Factor				
1	1.652	**0.110**	8.251	1.282
10	3.113	**0.540**	8.581	1.938
100	18.895	**6.755**	12.198	12.902
1000	101.832	70.447	**15.353**	21.401
10000	-	-	**39.301**	111.296
100000	-	-	**284.883**	1145.824

Table 5.3: Measured execution times for the Endeavor preprocessing pipeline.

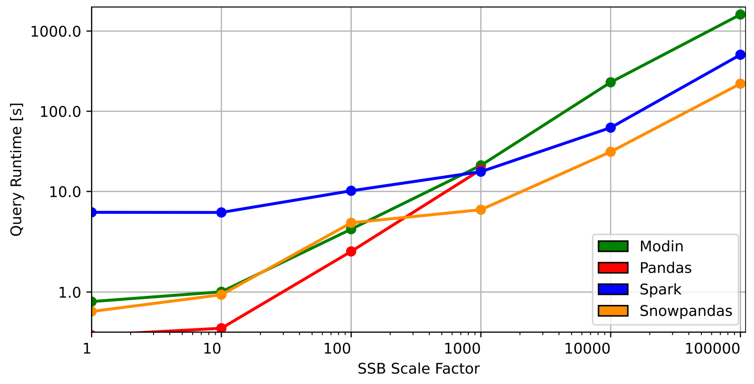


Figure 5.10: Performance of the "Fork" preprocessing pipeline.

5.2.9 Snowflake Virtual Warehouse Sizes

Results

Figure 5.13 illustrates the performance results for different VW sizes over three runs per configuration. Initially, the execution time when increasing the VW size decreases significantly. Not only is the decrease significant, but it closely matches the increase in price and, as such, the results support the idea that increasing VM size can be a net equal decision in terms of costs.

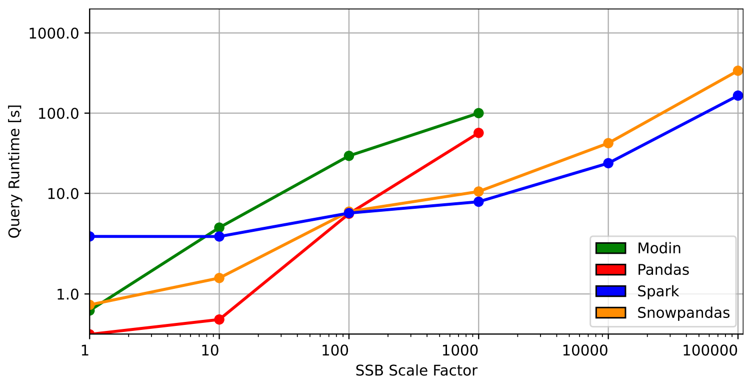


Figure 5.11: Performance of the "CatBoost" preprocessing pipeline.

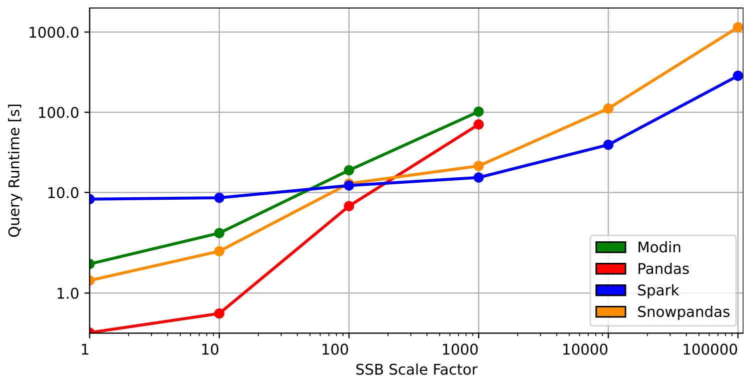


Figure 5.12: Performance of the "Endeavor" preprocessing pipeline.

However, when evaluating the execution times shown in table 5.4, more closely, we can see that this trend does not hold indefinitely. While the step from a small VW to a medium VW decreases the execution time by a factor of 1.99, the step from a large VW to an xlarge VW only leads to a decrease by a factor of 1.56. The decrease in the effectiveness of increasing the VW size becomes even more severe when looking at the Fork pipeline, where the step from large to xlarge VM only leads to a decrease by a factor of 1.33. This is not surprising, however, since the constant parallelization overhead impacts the overall runtime for faster executions.

Overall, the results of this experiment strongly suggest that the computational power of VWs increases proportionally to the price. However, this is not positive for Snowflake. Because one would suspect a bulk discount by choosing the bigger product.

	Execution Time Mean [s]		
Mode	Catboost	Endeavor	Fork
Scale			
Small	342.053015	1146.948731	220.909072
Medium	176.446166	576.312729	115.390882
Large	94.154802	296.262303	59.539095
Xlarge	65.842719	189.944288	44.688998

Table 5.4: Measured execution times of the preprocessing pipelines for different VW sizes.

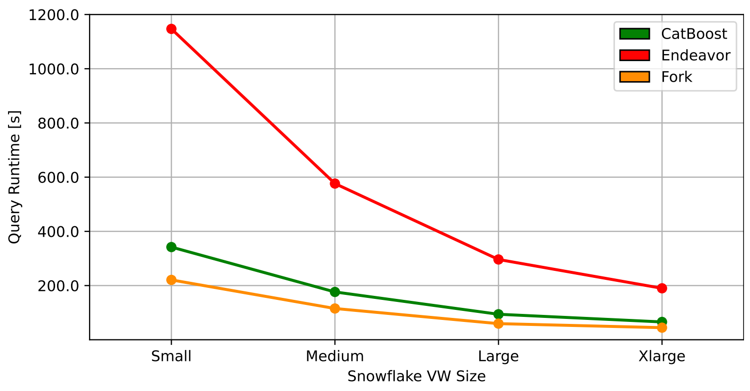


Figure 5.13: Query runtimes for the preprocessing pipelines on VWs of different sizes.

Chapter 6

Conclusions and Future Work

In this thesis, we aimed to perform ML preprocessing on a relational database. To achieve this, we designed Snowpandas, a system that is based on Modin and implements the popular ML preprocessing API of Pandas. In doing so, Snowpandas implements the Snowpark API as an execution backend for Modin, utilizing Snowpark's inherent lazy execution. We meticulously ensured that lazy execution is preserved as long as possible to make optimal use of Snowflake's query optimizer.

While preventing early execution is simple in most cases, the Pandas API's core functionality of column assignments presents a unique challenge. To address this, we developed a system whereby executions on DataFrames are tracked and stored in a purpose-built data structure. Upon column assignments, this allowed us to reapply the necessary operations on the given column within the original DataFrame. By doing so, we avoided performance-sensitive operations such as joins.

In a series of experiments, we first established that Snowpandas does not introduce inefficiency in the Snowflake query compilation and execution layer compared to handwritten SQL queries. Additionally, we showed that for traditional relational database workloads, Snowpandas outperforms competing Pandas APIs such as Pandas, Modin, and Spark for the scaling factors tested. Pandas, Modin, and Spark perform significantly worse, the query runtime being higher on average by a factor of 214 for Modin, 123 for Pandas, and 7.3 for Spark. However, as data volume increases, the margin between Spark and Snowpandas narrows. This trend indicates that for larger data volumes, Spark could potentially outperform Snowpandas. By executing a series of microbenchmarks, we further showed that this is also the case for most of the individual operators used in the SSB experiment, again under the caveat that the scaling factors tested might not have reached the scale where Spark would gain the upper hand. However, our experiments testing Snowpandas on real-world ML preprocessing pipelines conclusively showed that Spark outperforms our Snowpandas implementation even at moderate data volumes. Compared to the more traditional Pandas APIs in the form of Pandas itself and Modin, Snowpandas showed significant performance improvements. We say Snowpandas outperformed Pandas and Modin by a significant margin, although Snowpandas only outperformed Pandas by a factor of 1.2 and Modin by 2.8 on average. This however gives a limited picture, the initial data size is minuscule at under 1MB when stored as a CSV file. At these scales execution overhead skews the results, since at the moderate scaling factor of 1000, Snowpandas already outperforms Pandas by a factor of 11.8 and 17.8 for Modin. Higher scaling factors could only be performed for Spark and Snowpandas, with Spark outperforming Snowpandas by a factor of 1.93. As such when combined with the ease of use that Snowpandas provides via the Snowflake VW, Snowpandas presents an attractive solution for intermediate data volumes. It offers a balance of performance and usability, allowing users to avoid the server/cluster management burdens inherent to a Spark solution.

Additional functions will undoubtedly be necessary if Snowpandas is to be tested on more workloads. The Pandas API is vast, and while we have focused on the most common functions, we have only touched a small subset of Pandas' total functionality.

Optimizations might still be possible, especially in cases where Snowflake's execution is not yet fully understood. The Snowpandas API offers many different ways to achieve the same semantics, while the query optimizer is effectively a black box, there are still many ways in which our implementation can be improved. One notable area of improvement is the aforementioned problem from section 4.2.7. Developing an effective solution to this problem would significantly enhance Snowpark's performance.

Testing Further testing is needed to clearly understand the workloads for which Snowpandas delivers competitive performance and those where it can not stand up to traditional data warehouse systems such as Spark. Understanding Snowpandas' sensitivity to differences in workload could be crucial in identifying a niche where a Snowpandas-like system would excel.

Appendix A

Appendix

A.1 Experiments

A.1.1 Notebook Code

The extraction and translation of the preprocessing pipeline from the Endeavor notebook resulted in the code displayed in listing A.2. The code for the Catboost notebook can be found in listing A.1.

```
if self.engine == "spark":
    from pyspark.sql.functions import split
    self.train = self.train.to_spark() \
        .withColumn('Cabin1_', split(self \
        .train.to_spark()['Cabin'], '/')[0]) \
        .pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('Cabin2_', split(self \
        .train.to_spark()['Cabin'], '/')[1]) \
        .pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('Cabin3_', split(self \
        .train.to_spark()['Cabin'], '/')[2]) \
        .pandas_api()

    self.train = self.train.to_spark() \
        .withColumn('Pid1_', split(self \
        .train.to_spark()['PassengerId'] \
        , '_')[0]).pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('Pid2_', split(self \
        .train.to_spark()['PassengerId'] \
        , '_')[1]).pandas_api()
        self.train[["Pid1_", 'Pid2_']] = self \
            .train[["Pid1_", 'Pid2_']].astype('int')

        self.train = self.train.to_spark() \
            .withColumn('Fname_',split(self.train \
            .to_spark()['Name'], ' ')[0]).pandas_api()
        self.train = self.train.to_spark() \
            .withColumn('Lname_',split(self.train \
            .to_spark()['Name'], ' ')[1]).pandas_api()

        self.train = self.train.to_spark() \
            .withColumn('Fname_',split(self.train \
            .to_spark()['Name'], ' ')[0]).pandas_api()
        self.train = self.train.to_spark() \
            .withColumn('Lname_',split(self.train \
            .to_spark()['Name'], ' ')[1]).pandas_api()

    else:
        self.train[["Cabin1_", 'Cabin2_', 'Cabin3_']] = \
            self.train['Cabin'].str.split('/', expand=True)
        self.train[["Pid1_", 'Pid2_']] = self.train['PassengerId'] \
            .str.split('_', expand=True)
        self.train[["Pid1_", 'Pid2_']] = self.train[["Pid1_", 'Pid2_']] \
            .astype('int')
        self.train[["Fname_", 'Lname_']] = self.train['Name'] \
            .str.split(' ', expand=True)

    self.train['sum_exp_'] = self.train['RoomService'] \
        + self.train['FoodCourt']
    self.train['sum_exp_'] = self.train['sum_exp_'] \
        + self.train['ShoppingMall']
    self.train['sum_exp_'] = self.train['sum_exp_'] \
        + self.train['Spa']
    self.train['sum_exp_'] = self.train['sum_exp_'] \
        + self.train['VRDeck']

    self.train['sum_exp_'] = self.train['sum_exp_'] \
        / self.train['Pid2_']
    #self.test['sum_exp_'] = self.test['sum_exp_'] \
        / self.test['Pid2_']

    #Original call
    self.train.loc[self.train['Age'] <= 5, ['Age_cat_']] = 'Toddler/Baby'
```

```
self.train.loc[((self.train['Age'] > 5) & \
    (self.train['Age'] <= 12)), ['Age_cat_']] = 'Child'
self.train.loc[((self.train['Age'] > 12) & \
    (self.train['Age'] <= 18)), ['Age_cat_']] = 'Teen'
self.train.loc[((self.train['Age'] > 18) & \
    (self.train['Age'] <= 50)), ['Age_cat_']] = 'Adult'
self.train.loc[((self.train['Age'] > 50) & \
    (self.train['Age'] <= 150)), ['Age_cat_']] = 'Elderly'

#Trigger execution for all engines
if self.engine == "snowpandas":
    batches = self.train._query_compiler._modin_frame \
        ._frame._frame.to_pandas_batches()
    result = next(batches)
else:
    result = self.train
print(result.head())
```

Listing A.1: Extracted preprocessing logic from the Catboost notebook.

```
# Some feature engineering
def fill_nans_by_age(df, age_limit=13):
    df.loc[df['Age'] < age_limit, \
        ['RoomService', 'FoodCourt', 'ShoppingMall', \
        'Spa', 'VRDeck']] = 0
    return df

def fill_nans_by_cryo(df):
    df.loc[df['CryoSleep'] == True, ['RoomService', \
        'FoodCourt', 'ShoppingMall', 'Spa', 'VRDeck']] = 0
    return df

def age_groups(df, age_limit = 13):
    df['AgeGroup'] = 1.0
    df.loc[df['Age'] < age_limit, ['AgeGroup']] = 0.0
    return df

def fill_missing(df):
    '''
    Fill NaNs values or with mean or most commond value...

    '''

    numerics = ['int16', 'int32', 'int64', 'float16', \
            'float32', 'float64']

        numeric_tmp = df.select_dtypes(include = numerics)
        categ_tmp = df.select_dtypes(exclude = numerics)

        for col in numeric_tmp.columns:
            if self.engine == "snowpandas":
                df[col] = df[col].fillna(method="snow_mean")
            else:
                m = df[col].mean()
                df[col] = df[col].fillna(value=m)

        for col in categ_tmp.columns:
            if self.engine == "snowpandas":
                df[col] = df[col].fillna(method="snow_mode")
            else:
                mode = df[col].mode()[0]
                if self.engine == "spark" and \
                    (mode == False or mode == True):
                    df[col] = df[col].astype(bool)
                    mode = bool(mode)
                    df[col] = df[col].fillna(value=mode)
                else:
                    df[col] = df[col].fillna(value=mode)
        return df

    def total_billed(df):
        '''
        Calculates total amount billed in the trip to the passenger...
        Args:
        Returns:

        '''
        df['Total_Billed'] = df['RoomService'] + df['FoodCourt']
        df['Total_Billed'] = df['Total_Billed'] + df['ShoppingMall']
        df['Total_Billed'] = df['Total_Billed'] + df['Spa']
        df['Total_Billed'] = df['Total_Billed'] + df['VRDeck']
        return df

    def cabin_separation(df):
        '''
        Split the Cabin name into Deck, Number and Side

        '''
    df[["CabinDeck", "CabinNum", "CabinSide"]] = \
        df['Cabin'].str.split('/', expand=True)
    df.drop(columns = ['Cabin'], inplace = True)
    return df

def name_ext(df):
    '''
    Split the Name of the passenger into First and Family...

    '''
    df[["FirstName", "FamilyName"]] = df['Name'] \
        .str.split(' ', expand=True)
    df.drop(columns = ['Name'], inplace = True)
    return df

def extract_group(df):
    df[['TravelGroup', 'ID']] =  df['PassengerId'] \
        .str.split('_', expand = True)
    df.drop(columns=['PassengerId'], inplace= True)
    return df

if self.engine == "spark":
    self.train = fill_nans_by_age(self.train)
    self.train = fill_nans_by_cryo(self.train)
    self.train = age_groups(self.train)
    self.train = fill_missing(self.train)

    self.train = total_billed(self.train)

    self.train = self.train.to_spark() \
        .withColumn('CabinDeck',split(self.train \
        .to_spark()['Cabin'], '/')[0]).pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('CabinNum',split(self.train \
        .to_spark()['Cabin'], '/')[1]).pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('CabinSide',split(self.train \
        .to_spark()['Cabin'], '/')[2]).pandas_api()
    self.train = self.train.drop('Cabin', axis=1)
    self.train = self.train.to_spark() \
        .withColumn('FirstName',split(self.train \
        .to_spark()['Name'], ' ')[0]).pandas_api()
    self.train = self.train.to_spark() \
        .withColumn('FamilyName',split(self.train \
            .to_spark()['Name'], ' ')[1]).pandas_api()
        self.train = self.train.drop('Name', axis=1)
        self.train = self.train.to_spark() \
            .withColumn('TravelGroup',split(self.train\
            .to_spark()['PassengerId'], '_')[0]).pandas_api()
        self.train = self.train.to_spark()\
            .withColumn('ID',split(self.train \
            .to_spark()['PassengerId'], '_')[1]).pandas_api()
        self.train = self.train.drop('PassengerId', axis=1)
    else:
        self.train = fill_nans_by_age(self.train)
        self.train = fill_nans_by_cryo(self.train)
        self.train = age_groups(self.train)
        self.train = fill_missing(self.train)
        self.train = total_billed(self.train)
        self.train = cabin_separation(self.train)
        self.train = name_ext(self.train)
        self.train = extract_group(self.train)
    # trigger execution for all engines
    if self.engine == "snowpandas":
        batches = self.train._query_compiler \
            ._modin_frame._frame._frame \
            .to_pandas_batches()
        result = next(batches)
    else:
        result = self.train
```

Listing A.2: Extracted preprocessing logic from the Endeavor notebook.

A.1.2 SSB Results

A.12 and A.13 show the results of the SSB experiment for scaling factor 1 and 100 respectively. They follow the same pattern as the results for scaling factor 10 and 1000 presented in 5.2.3. Lineplots for SSB queries 1.1, 1.2, 1.3, 2.2, 2.3, 3.1, 3.2, 3.3, 3.4, 4.2 and 4.3 can be found in A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, A.10 and A.11 respectively.

A.1.3 Microbenchmark Results

We conducted further microbenchmarks not yet displayed. Figure A.14 shows the performance of the `sort` microbenchmark, figure A.15 the performance of `bin_op` and figure A.16 the performance of `selection` microbenchmark.

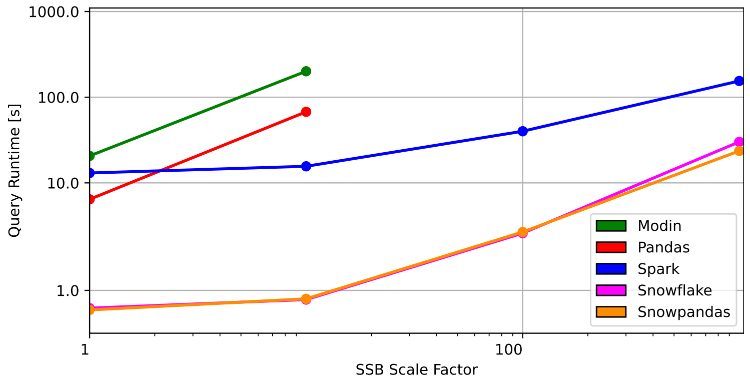


Figure A.1: Query runtime depending on scaling factor for SSB query 1.1

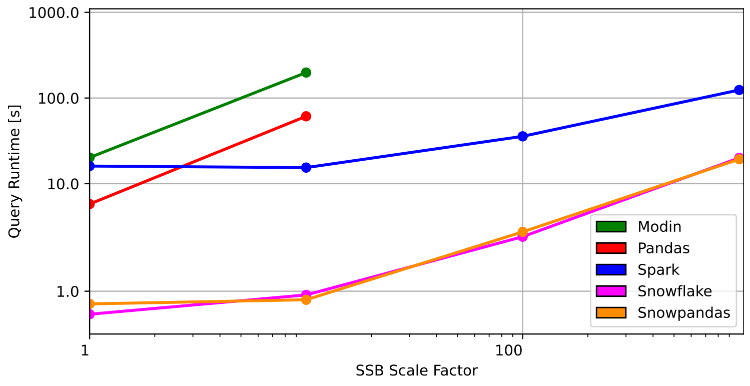


Figure A.2: Query runtime depending on scaling factor for SSB query 1.2

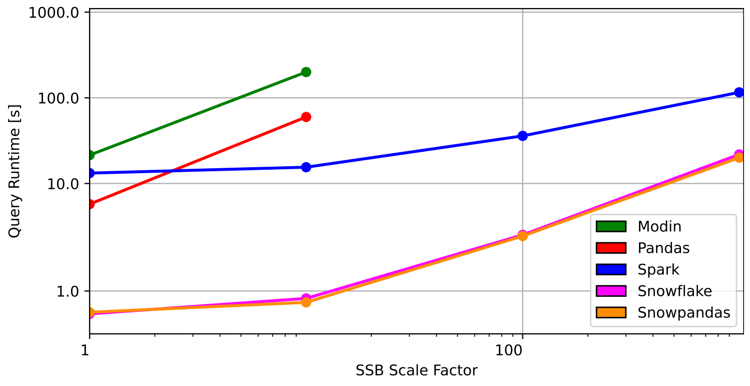


Figure A.3: Query runtime depending on scaling factor for SSB query 1.3

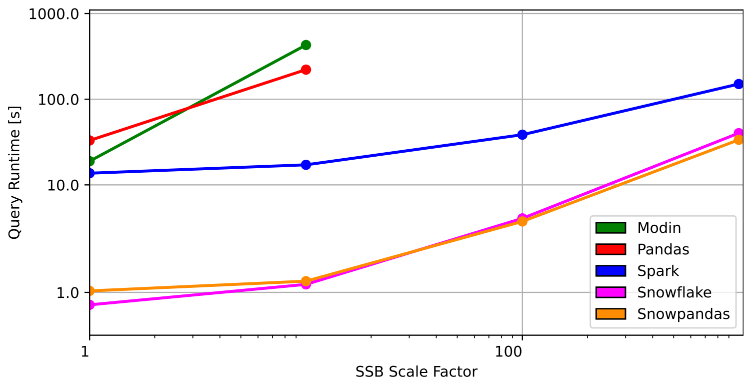


Figure A.4: Query runtime depending on scaling factor for SSB query 2.2

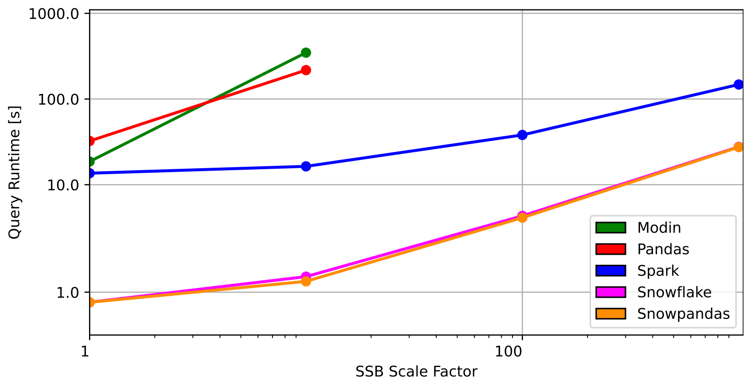


Figure A.5: Query runtime depending on scaling factor for SSB query 2.3

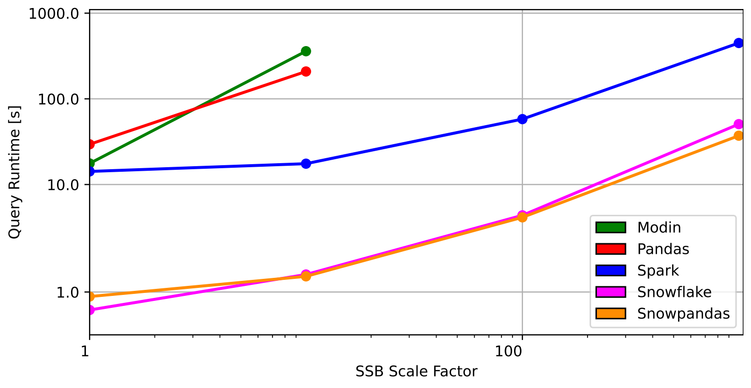


Figure A.6: Query runtime depending on scaling factor for SSB query 3.1

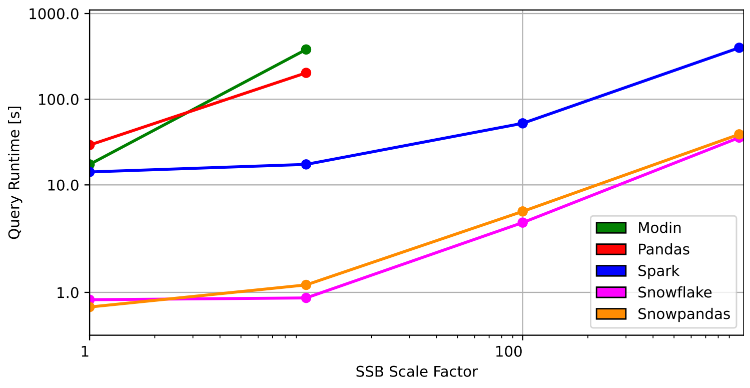


Figure A.7: Query runtime depending on scaling factor for SSB query 3.2

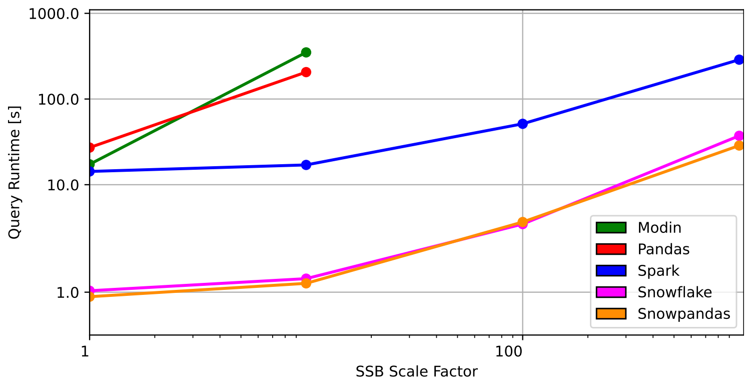


Figure A.8: Query runtime depending on scaling factor for SSB query 3.3

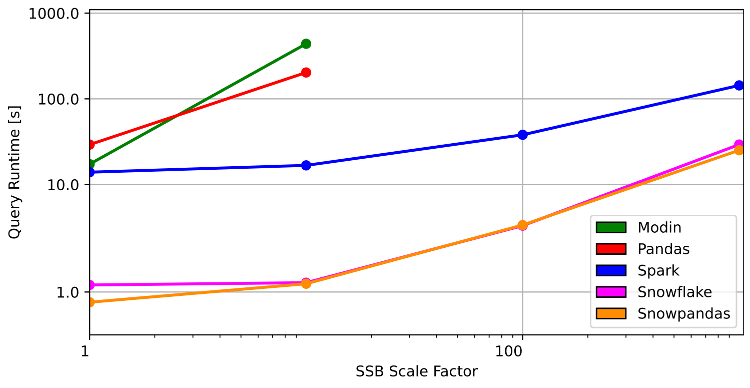


Figure A.9: Query runtime depending on scaling factor for SSB query 3.4

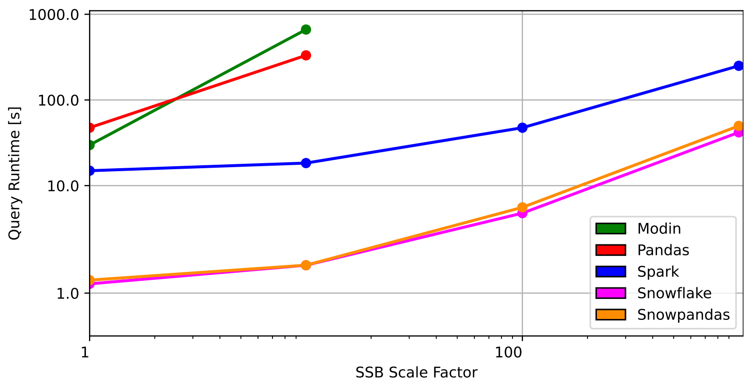


Figure A.10: Query runtime depending on scaling factor for SSB query 4.2

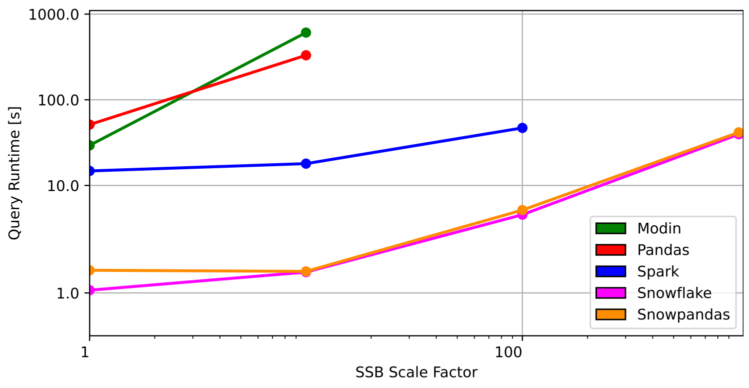


Figure A.11: Query runtime depending on scaling factor for SSB query 4.3

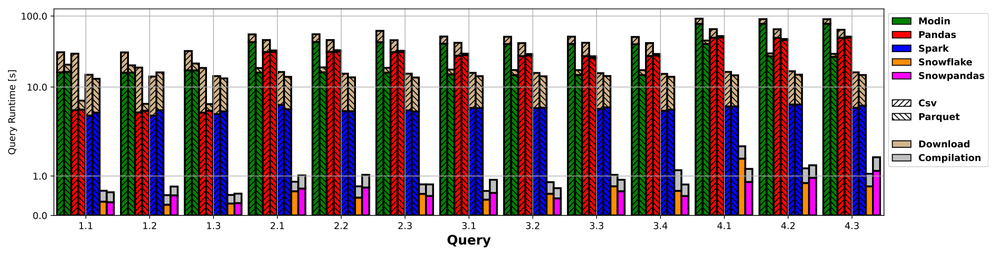


Figure A.12: This figure shows the query runtimes, download time, and compilation time for all systems for scaling factor 1 of the SSB.

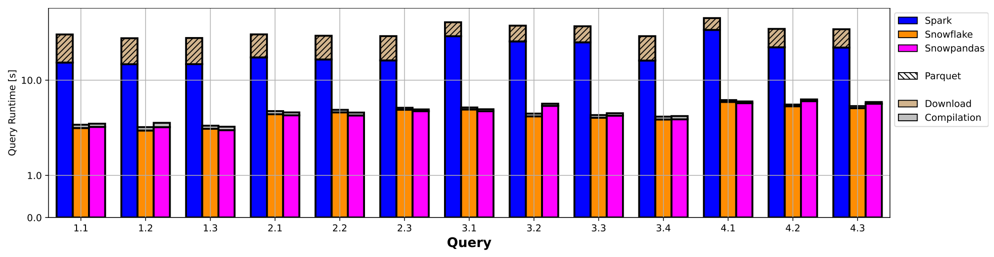


Figure A.13: This figure shows the query runtimes, download time, and compilation time for all systems for scaling factor 100 of the SSB.

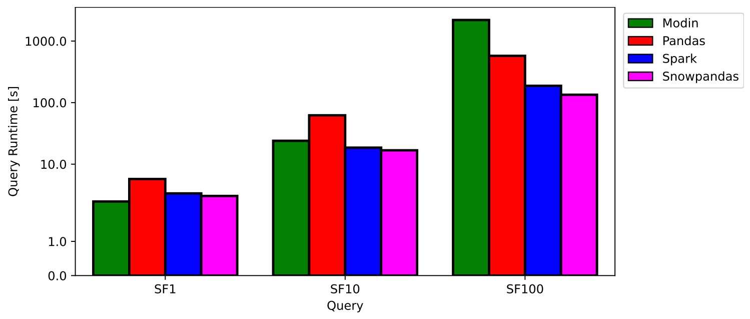


Figure A.14: Performance of the `sort` microbenchmark.

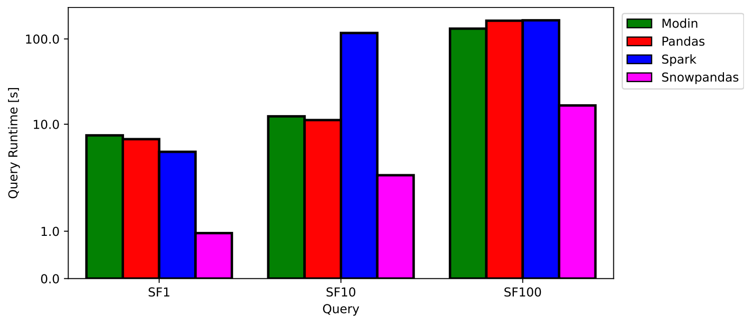


Figure A.15: Performance of the `bin_op` microbenchmark.

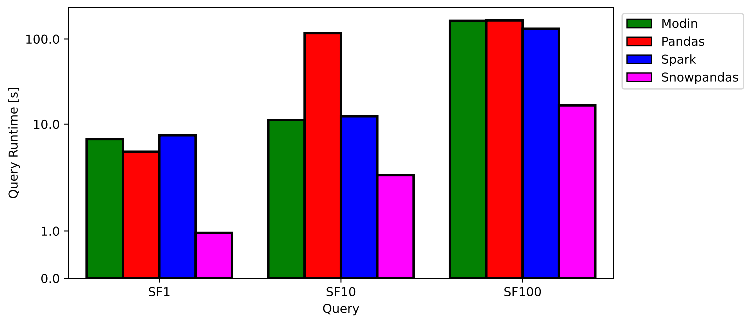


Figure A.16: Performance of the `selection` microbenchmark.

Bibliography

[1] Amazon EC2 - Cloud Compute Capacity - AWS — aws.amazon.com. https://aws.amazon.com/ec2/. Accessed 01-06-2024.

[2] Competitions documentation. https://www.kaggle.com/docs/notebooks. Accessed: 2024-05-18.

[3] Dask documentation — docs.dask.org. https://docs.dask.org/en/stable/generated/dask.dataframe.compute.html. Accessed 01-06-2024.

[4] Data types | snowflake documentation.

[5] Key concepts & architecture | snowflake documentation. https://docs.snowflake.com/en/user-guide/intro-key-concepts#cloud-services. Accessed: 2024-05-14.

[6] Koalas 1.8.2 documentation — koalas.readthedocs.io. https://koalas.readthedocs.io/en/latest/reference/api/databricks.koalas.DataFrame.to_pandas.html. Accessed 01-06-2024.

[7] Out-Of-Memory Prevention &x2014; Ray 2.23.0 — docs.ray.io. https://docs.ray.io/en/latest/ray-core/scheduling/ray-oom-prevention.html. Accessed 01-06-2024.

[8] Parquet — parquet.apache.org. https://parquet.apache.org/. Accessed 01-06-2024.

[9] The selenium browser automation project. https://www.selenium.dev/documentation/. Accessed: 2024-05-18.

[10] Spaceship Titanic — kaggle.com. https://www.kaggle.com/competitions/spaceship-titanic/overview. Accessed 01-06-2024.

[11] System architecture — modin 0.29.0+37.g1c0d9a6.dirty documentation. `https://modin.readthedocs.io/en/latest/development/architecture.html`. Accessed: 2024-05-14.

[12] Titanic - Machine Learning from Disaster — kaggle.com. `https://www.kaggle.com/competitions/titanic`. Accessed 01-06-2024.

[13] What is cloud computing? — microsoft azure. `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing`. Accessed: 2024-05-14.

[14] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Saqib Alam and Nianmin Yao. *The impact of preprocessing steps on the accuracy of machine learning algorithms in sentiment analysis*. Springer Science+Business Media, LLC, part of Springer Nature 2018, 2019.

[16] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiř'ı Šimša, and Chandramohan A. Thekkath. tf.data service: A case for disaggregating ml input data processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 358–375, New York, NY, USA, 2023. Association for Computing Machinery.

[17] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, 2007.

[18] Greg Boss, Padma Malladi, Dennis Quan, Linda Legregni, and Harold Hall. Cloud computing. *IBM white paper*, 321:224–231, 2007.

[19] C4rl05/V. GitHub - intel/hdk: A low-level execution library for analytic data processing. — github.com. `https://github.com/intel/hdk`. Accessed 01-06-2024.

[20] C4rl05/V. Spaceship Titanic Higher Score Endeavor — kaggle.com. `https://kaggle.com/code/cv13j0/spaceship-titanic-higher-score-endeavor`. Accessed 01-06-2024.

[21] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), dec 2009.

[22] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART*

Symposium on Principles of Database Systems, PODS '98, page 34–43, New York, NY, USA, 1998. Association for Computing Machinery.

[23] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[25] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[26] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. Evaluating query languages and systems for high-energy physics data. *Proc. VLDB Endow.*, 15(2):154–168, oct 2021.

[27] Dan Graur, Remo Röthlisberger, Adrian Jenny, Ghislain Fourny, Filip Drozdowski, Choden Konigsmark, Ingo Müller, and Gustavo Alonso. Addressing the nested data processing gap: Jsoniq queries on snowflake through snowpark. 2024. 40th IEEE International Conference on Data Engineering (ICDE 2024); Conference Location: Utrecht, Netherlands; Conference Date: May 13-17, 2024.

[28] Pasuvula Sai Kiran. Titanic spaceship feature selection catboost — kaggle.com. `https://kaggle.com/code/pasuvulasaikiran/titanic-spaceship-feature-selection-catboost`. Accessed 01-06-2024.

[29] Sotiris B Kotsiantis, Dimitris Kanellopoulos, and Panagiotis E Pintelas. Data preprocessing for supervised leaning. *International journal of computer science*, 1(2):111–117, 2006.

[30] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[31] Benedikt Martens, Marc Walterbusch, and Frank Teuteberg. Costing of cloud computing services: A total cost of ownership approach. In *2012 45th Hawaii International Conference on System Sciences*, pages 1563–1572, 2012.

[32] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[33] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[34] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

[35] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb), 2007.

[36] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *arXiv preprint arXiv:2001.00888*, 2020.

[37] Mason Proffitt, Ingo Müller, Dan Graur, Mat Adamec, Pieter David, Enrico Guiraud, and Sebastien Binet. iris-hep/adl-benchmarks-index: Adl functionality benchmarks index v0. 1. 10.5281/zenodo. 5131287, 2021.

[38] Kovács Péter. Fork of Starship — kaggle.com. `https://kaggle.com/code/kovcspter/fork-of-starship`. Accessed 01-06-2024.

[39] Matthew Rocklin et al. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, pages 126–132, 2015.

[40] Jimi Sanchez. A review of star schema benchmark, 2016.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[42] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*, pages 1042–1057, 2022.