KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Software Engineering for Distributed Systems

Second cycle, 30 credits

Resource-efficient and fast Point-in-Time joins for Apache Spark

Optimization of time travel operations for the creation of machine learning training datasets

KARL AXEL PETTERSSON

Stockholm, Sweden, 2022

Resource-efficient and fast Point-in-Time joins for Apache Spark

Optimization of time travel operations for the creation of machine learning training datasets

KARL AXEL PETTERSSON

Master's Programme, Software Engineering of Distributed Systems, 120 credits
Date: May 30, 2022

Supervisors: Sina Sheikholeslami, Moritz Meister
Examiner: Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: Resurseffektiva och snabba Point-in-Time joins i Apache Spark
Swedish subtitle: Optimering av tidsresningsoperationer för skapande av träningsdata för maskininlärningsmodeller

Abstract

A scenario in which modern machine learning models are trained is to make use of past data to be able to make predictions about the future. When working with multiple structured and time-labeled datasets, it has become a more common practice to make use of a join operator called the Point-in-Time join, or PIT join, to construct these datasets. The PIT join matches entries from the left dataset with entries of the right dataset where the matched entry is the row whose recorded event time is the closest to the left row's timestamp, out of all the right entries whose event time occurred before or at the same time of the left event time. This feature has long only been a part of time series data processing tools but has recently received a new wave of attention due to the rise of the popularity of feature stores. To be able to perform such an operation when dealing with a large amount of data, data engineers commonly turn to large-scale data processing tools, such as Apache Spark. However, Spark does not have a native implementation when performing these joins and there has not been a clear consensus by the community on how this should be achieved. This, along with previous implementations of the PIT join, raises the question: "How to perform fast and resource efficient Point-in-Time joins in Apache Spark?". To answer this question, three different algorithms have been developed and compared for performing a PIT join in Spark in terms of resource consumption and execution time. These algorithms were benchmarked using generated datasets using varying physical partitions and sorting structures. Furthermore, the scalability of the algorithms was tested by running the algorithms on Apache Spark clusters of varying sizes. The results received from the benchmarks showed that the best measurements were achieved by performing the join using *Early Stop Sort-Merge Join*, a modified version of the regular Sort-Merge Join native to Spark. The best performing datasets were the datasets that were sorted by timestamp and primary key, ascending or descending, using a suitable number of physical partitions. Using this new information gathered by this project, data engineers have been provided with general guidelines to optimize their data processing pipelines to be able to perform more resource-efficient and faster PIT joins.

Keywords

Apache Spark, Point-in-Time, ASOF, Join, Optimizations, Time travel

Sammanfattning

Ett vanligt scenario för maskininlärning är att träna modeller på tidigare observerad data för att för att ge förutsägelser om framtiden. När man jobbar med ett flertal strukturerade och tidsmärkta dataset har det blivit vanligare att använda sig av en join-operator som kallas Point-in-Time join, eller PIT join, för att konstruera dessa datauppsättningar. En PIT join matchar rader från det vänstra datasetet med rader i det högra datasetet där den matchade raden är den raden vars registrerade händelsetid är närmaste den vänstra raden händelsetid, av alla rader i det högra datasetet vars händelsetid inträffade före eller samtidigt som den vänstra händelsetiden. Denna funktionalitet har länge bara varit en del av datahanteringsverktyg för tidsbaserad data, men har nyligen fått en ökat popularitet på grund av det ökande intresset för feature stores. För att kunna utföra en sådan operation vid hantering av stora mängder data vänder sig data engineers vanligvis till storskaliga databehandlingsverktyg, såsom Apache Spark. Spark har dock ingen inbyggd implementation för denna join-operation, och det finns inte ett tydligt konsensus från Spark-rörelsen om hur det ska uppnås. Detta, tillsammans med de tidigare implementationerna av PIT joins, väcker frågan: ”Vad är det mest effektiva sättet att utföra en PIT join i Apache Spark?”. För att svara på denna fråga har tre olika algoritmer utvecklats och jämförts med hänsyn till resursförbrukning och exekveringstid. För att jämföra algoritmerna, exekverades de på genererade datauppsättningar med olika fysiska partitioner och sorteringstrukturer. Dessutom testades skalbarheten av algoritmerna genom att köra de på Spark-kluster av varierande storlek. Resultaten visade att de bästa mätvärdena uppnåddes genom att utföra operationen med algoritmen *early stop sort-merge join*, en modifierad version av den vanliga sort-merge join som är inbyggd i Spark, med en datauppsättning som är sorterad på tidsstämpel och primärnyckel, antingen stigande eller fallande. Fysisk partitionering av data kunde även ge bättre resultat, men det optimala antal fysiska partitioner kan variera beroende på datan i sig. Med hjälp av denna nya information som samlats in av detta projekt har data engineers försetts med allmänna riktlinjer för att optimera sina databehandlings-pipelines för att kunna utföra mer resurseffektiva och snabbare PIT joins.

Nyckelord

Apache Spark, Point-in-Time, ASOF, Join, Optimeringar, Tidsresning

Acknowledgments

I would like to especially thank Moritz Meister and the Hopsworks team for making this thesis project possible and helping out throughout the project. I would also like to thank Sina Sheikholeslami and Amir Payberah for being involved and guiding me through the thesis process.

Stockholm, May 2022
Karl Axel Pettersson

Contents

List of Figures

List of Tables

Listings

List of Algorithms

List of acronyms and abbreviations

AM	Application Master
API	Application Programming Interface
AST	Abstract Syntax Tree
DAG	Directed Acyclic Graph
FIFO	First In, First Out
HDFS	Hadoop Distributed File System
HiveQL	Hive Query Language
JVM	Java Virtual Machine
PIT	Point-in-Time
RDD	Resilient Distributed Dataset
SQL	Structured Query Language
UDF	User Defined Function

Chapter 1

Introduction

When constructing machine learning models, data scientists need, in many cases, to look at past examples of data to build a model for the future. A *feature* is a measurable attribute of the event you are trying to predict. The process of constructing meaningful features from raw data has long been a crucial part of machine learning and is important for creating well-performing models. Furthermore, as the number of features and data grows, the feature engineering process can become computationally expensive.

Feature stores are a new emerging technology that is used to combat these issues by providing an interface between raw data and machine learning models. A feature store can track the transformation of raw data into feature values, store, manage, and monitor the features themselves, and serve the features to be used within machine learning models.

When training a model on past data, we must be able to have a complete view of the state of the data at some specific time, that is, the value of the features at that point in the past. However, it is important that future feature values are not leaked into the training data and that the results precisely capture what is known about the system at the point in time when the prediction event occurred. If our data did not capture a correct view of the feature values at that time, we would train our model on scenarios that never actually happened, which could result in many flawed predictions. For building such a training dataset, a particular functionality called Point-in-Time (PIT) join is used [1]. These joins are an inexact temporal operator that merges rows from one collection of data with a row from another collection of data whose timestamp is before the left timestamp, but also closest to it. Because of how this join works, it is also referred to as an *ASOF join*, since we are observing the most recent values as of some particular timestamp. The implementation

of this join operator can be seen in the Pandas data analysis library [2] and the relational time series database Kdb+ [3].

1.1 Problem Statement

Apache Spark [4], which is a common framework used for large-scale data processing, currently does not have a native implementation to execute these PIT joins for structured data. It is possible to obtain correct results by using standard join operators through Spark SQL queries. However, these approaches can become less efficient when dealing with a large amount of data and multiple groups of joins, as it might explode the data into intermediary tables with an exponentially increasing number of rows and become computationally inefficient.

1.2 Research Question

The goal of the project is to propose, design, and evaluate different methods of executing PIT joins in the Spark ecosystem. These solutions should be compared to each other in terms of execution time, memory usage, and significant metrics. Hence, the research question to be answered is: *How to perform fast and resource efficient Point-in-Time joins in Apache Spark?*

1.3 Purpose and Goals

The purpose of this project is to provide a more efficient way to create training datasets for machine learning models to learn about past events. By optimizing this process and making it easier to use, data engineers working in Apache Spark are able to produce relevant datasets that will be used by data scientists in a faster and more efficient way. The data scientists will be able to receive the datasets faster and train their models, thus further improving the process.

These main goals of this project can be divided into the following tasks:

1. Implement an extension to Spark that exposes functionality that aims to execute PIT joins using different algorithms and methods.

2. Provide a comparison of the implementations and their benefits and drawbacks in comparison to each other.

1.4 Ethics and Sustainability

The findings of this research could be utilized to optimize existing data processing pipelines in terms of computational efficiency. Optimizing the operations used within the systems could prove beneficial for reducing the energy consumption of these systems, because of the reduced CPU time used for computations.

1.5 Research Methodology

This project will be carried out using an empirical approach [5]. First, algorithms will be proposed to execute PIT joins, which will be based on previous implementations and possible findings in the literature study. The theoretical performance of these algorithms in Apache Spark will be analyzed using a cost modeling similar to the model presented in [6]. Time and space complexity will be mostly focused on, while also taking note of the potential disk I/O operations and network transmissions, as these aspects will have a large effect on real-life performance. Furthermore, because some operations in Apache Spark performance are affected by how the input data is structured [7], possible ways to optimize the data itself will be investigated.

The implementations will be tested on different datasets of varying sizes and data distributions running on Spark clusters of different size. Because it is important to simulate real scenarios to account for the transmission cost between nodes, a real cluster of multiple nodes will be used.

Important metrics for the empirical evaluation will consider execution time, memory consumption, disk I/O, and transmission load, as all of these metrics affect the performance of the algorithm.

1.6 Delimitations

This project is not executed with the intention of adding functionality to the core Apache Spark library distribution. All modifications that will be made in this project will be accessible through an extension that contains predefined functions to allow the execution of the different algorithms. Although the implementations are not directly integrated with the Spark source code, low-level functionality can still be obtained with the use of APIs provided by the Spark distribution. More on how this is achieved will be described in future sections.

1.7 Outline

Chapter 2 provides relevant background information about Apache Spark and its internal workings, PIT joins and its relevance, as well as a review of relevant related work. Chapter 3 goes more in-depth on the methodology and method used to execute the project. Chapter 4 describes how the algorithms are implemented and the findings of the theoretical complexity analysis and what it says about the theoretical performance of the algorithms. In Chapter 5, the results obtained by executing the benchmarks on the implementations will be presented and analyzed. Finally, in Chapter 6, conclusions about the project results and the quality of the project will be discussed and possible future work within this area will be suggested.

Chapter 2

Background

This chapter provides background information on the internal workings of Apache Spark, relevant information on PIT joins, and previous relevant work within this topic area.

2.1 Joins

When working with strictly structured data in database systems, which are abstracted as tables, a common way to combine multiple sets of data is the SQL (Structured Query Language) join operator. In simple terms, this operation is executed using some algorithm, which combines the rows of two tables based on some *join conditions*; if more than two tables are to be joined, then this process is repeated. The join condition specifies which data from the left (first) table and the right (second) table should be used for match selection. These joining conditions can specify some logical operators. The joins using logical comparison of columns are commonly divided into *equi-joins* (matches of equal values) and *non-equi-joins* (matches with some other comparison operator, for example > and, <).

Joins can have different types depending on how the selection process should be made and what the results should consist of. The most commonly used join types are cross-, natural-, inner-, outer-, and self-join. These are briefly described in Table 2.1 [8].

2.1.1 Join Algorithms

Currently, there are three main general algorithms used for processing joins: *Sort-Merge join*, *Nested Loop*, and *Hash Join*.

Table 2.1: Common join types

Type	Description
Cross join	Combines all rows of left table with right table.
Natural join	Merges rows with same value of shared columns.
Inner join	Join that match rows based on comparison of common columns.
Left outer join	Return the inner join as well as the left rows without a match.
Right outer join	Return the inner join as well as the right rows without a match.
Full outer join	Return the inner join as well as both the left rows and right rows without a match.
Self join	Join which is executed on a table joined with itself.

In the Sort-Merge Join algorithm, the two tables that are to be joined are first sorted separately based on the columns that are used for the joining conditions. After the sorting has been completed, the resulting data can be obtained by a single scan through both sorted tables. The major downside of this algorithm is that it requires sorting, which, depending on the algorithm used, can become slow with increasing data size.

The Nested Loop Join works by using one table as the outer input table and the other as the inner input table. The outer loop iterates through the outer input table, whereas the inner loop iterates through the inner input table and searches for matches. This algorithm can be fast in pre-indexed and small datasets, but may perform badly if the data are not indexed and large.

The Hash Join algorithm does not require sorting of any of the tables and is executed by creating a hash table of matches from the left and right tables. The rows of the tables are hashed into a hash bucket based on the join key of the join condition. Only the values that are hashed into the same bucket are then compared and added to the results. Because this algorithm requires that the matching rows be hashed into the same bucket, it only works for equi-join conditions [9].

2.1.2 Point-in-Time Joins

The PIT join, also called the backward ASOF join, is not a common operator in database and data processing systems, mostly prevalent in time series

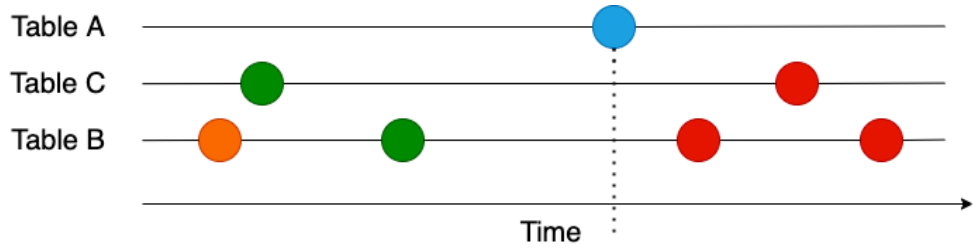


Figure 2.1: Visual demonstration of a Point-in-Time join of three tables

databases, such as QuasarDB[1] and Kdb+[2]; but the functionality is also included in other data processing tools such as JuliaDB.jl[3] and Pandas[4]. Essentially, it works by matching rows from the left and right tables based on timestamps, with an inexact time being allowed. If a timestamp in a row in the left table is not equal to any timestamp in the right table, it gets matched with the row with the most recent timestamp, which is less than or equal to the left timestamp.

The way this works is visualized in Figure 2.1, where each line represents the rows added to the table at certain points in time. Consider that the left table is Table A, which is joined with Tables C and B. Then consider the rows in Table A (the round shape colored blue) and try to find the rows in the other tables whose timestamp is the most recent row at that time; the matching rows from tables C and B are marked in green. The orange shapes are before the left timestamp but not the most recent, and the red shapes are after the left timestamp; therefore, they should not be prevalent in the joined result.

2.1.3 Importance in Feature Stores

With the increasing popularity of feature stores, a technology used to manage commonly used machine learning features, the demand for PIT joins has increased. In machine learning, features are a measurable piece of information; typically, in structured data collections, they are commonly represented as columns. A feature store makes it easier to reuse and organize

[1]QuasarDB SAS, 6. Select Available: https://doc.quasardb.net/master/queries/select.html. [Accessed: 2022-02-18]

[2]Kx Systems, Inc., asof Available: https://code.kx.com/q/ref/asof/. [Accessed: 2022-02-18]

[3]Julia Computing, Inc., API · JuliaDB.jl Available: https://juliadb.juliadata.org/stable/api. [Accessed: 2022-02-21]

[4]The Pandas Development Team, pandas.merge_asof — pandas 1.4.1 documentation Available: https://pandas.pydata.org/docs/reference/api/pandas.merge_asof.html. [Accessed: 2022-02-21]

features, saving data engineers the time to redevelop and specify new features [10]. Some of the most notable open-source feature stores to this day are Hopsworks[1] and Feast[2].

A common use case for machine learning models is to make predictions about the future. In order to obtain models that can achieve this goal, they must be trained on events that occurred in the past. For this purpose, it might be necessary to create training datasets that describe the state of feature values of an entity at a particular point in time in which an event occurred. For entities with many different features, the features may be spread across different data collections. Since it is important that the training data consist of all significant features of some entity, it is sometimes required to execute a PIT join to generate accurate training data that reflect reality, since these data collections could be updated at different frequencies.

2.2 Apache Spark

Apache Spark[3] is an open-source unified computing engine built to process large amounts of data in parallel among a cluster of computers. Spark provides libraries that make it possible to utilize it for programming languages such as Python, Scala, Java, and R, and provides functionality for analytical purposes and machine learning. Due to the unified design and broad range of relevant functionality for data engineers and data scientists, it has become especially popular for the pre-processing of stages of machine learning pipelines. It is within these pre-processing stages of machine learning that data used for training and testing is constructed [11].

2.2.1 Big Data

As the systems people use in their lives become more complex and advanced, there is a continuous challenge in the industry to ensure that the currently available hardware is able to handle the workload these systems require. In addition to this increase of complexity, a continuous increase in the amount of data collected from systems and people interacting for analytical and machine

[1] Logical Clocks AB, Hopsworks Feature Store with end-to-end ML pipeline. Available: https://www.hopsworks.ai/. [Accessed: 2022-03-07]

[2] Feast Authors, Feast: Feature Store for Machine Learning Available: https://feast.dev/. [Accessed: 2022-04-25]

[3] The Apache Software Foundation, Apache Spark™ - Unified Engine for large-scale data analytics Available: https://spark.apache.org/. [Accessed: 2022-01-25]

learning purposes has also been observed. In the past, this has been made possible by increasing the number of instructions per second that processors can handle. However, due to the hard limits of heat dissipation, this task became more difficult and hardware engineers required a new way to increase the speed of computers. This resulted in an increase in the use of parallelism in computers.

Instead of making each processor faster, more processors, or *cores*, were added to make it possible to execute computations in parallel with each other, making the processing speed faster. Furthermore, data collection technologies have not slowed down, and the cost of data storage is becoming cheaper each year [12]. As a result, organizations are collecting more and more data for monitoring and analytical purposes. Due to these trends, organizations require large amounts of parallel computations on large amounts of data, also known as *data-intensive computing*. To solve the challenges associated with data-intensive computing, new programming models and tools that specialize in this area were developed, Apache Spark being one of these solutions [11].

2.2.2 Architecture

Within a Spark Application, there are two types of processes that work together, the *driver process* and a set of *executor processes*. The responsibility of the driver is to maintain information about the Spark Application, respond to the user's input, analyze, distribute, and schedule tasks across the executors. One can think of the driver process as the entry point and the coordinator of the Spark Application. The executor's task is, as the name suggests, to execute the task assigned to it by the driver process and report its state to the driver process [11].

To manage the cluster of machines that composes a Spark Application, a cluster manager is used. Apache Spark includes a standalone cluster manager, but cluster managers such as Apache Mesos, Hadoop YARN, and Kubernetes can also be used for this purpose [13]. A high-level view of this architecture can be observed in the illustration in Figure 2.2.

2.2.3 Data Sources

Spark has support for six different "core" data sources, including CSV, JSON, Parquet, ORC, JDBC/ODBC connections, and plain-text files, while also having support for hundreds of external sources, managed by the community. When Spark reads the data, the developer may include a configuration on how

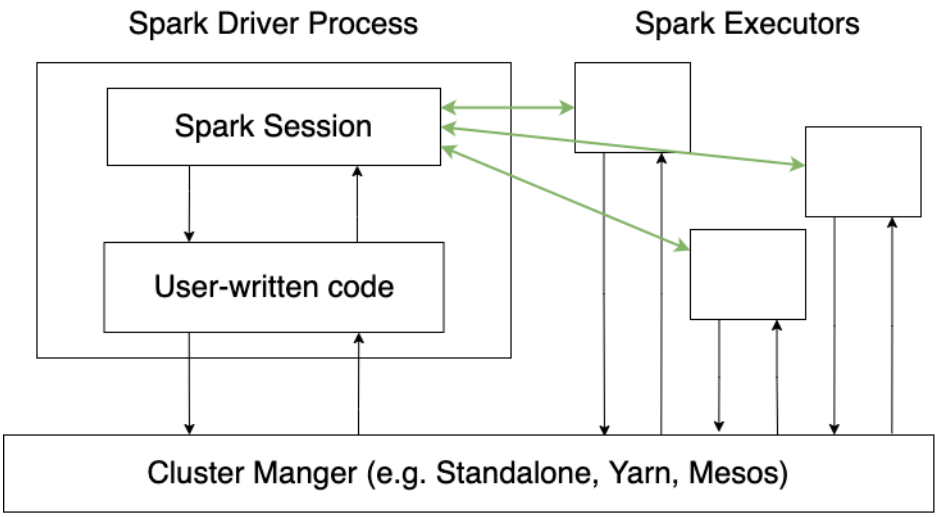


Figure 2.2: The architecture of a Spark Application

the data should be interpreted [11].

2.2.4 Data Structures

The core data structure concepts used in Spark are Datasets, DataFrames, SQL Tables, and Resilient Distributed Dataset (RDD). All these concepts represent collections of distributed data whose elements can be operated in parallel.

When working with structured data, the DataFrame API (Application Programming Interface) is the most widely used data collection abstraction. A DataFrame works conceptually similar to a SQL Table or spread-sheet; it consists of a table with rows and columns. In addition, it has metadata for columns that state the type of data with which it is populated, which is called a *schema*. Due to the tabular representation of a DataFrame, it can be constructed from a wide variety of data sources.

For working with strictly typed structured data, which require type-dependent operations, the Dataset API can be used. This API works similarly to the DataFrame API, since the DataFrame API is a Dataset organized into columns. A Dataset can be constructed from Java Virtual Machine (JVM) objects. Because a schema is used when operating on a DataFrame, Spark manipulates the `Row` object directly without the need for casting. However, when working with a typed Dataset, Spark converts the binary structure to a typed structure abstraction, which slows down the execution time of operations.

SQL tables work essentially like DataFrames, as they are structured collections of data on which queries are run. The biggest difference between DataFrames and tables is that DataFrames are defined in the context of the programming language, while tables are defined within the context of a database. For operations on tables in the database, the Spark client session has a connection to the database and exposes the entry point in which SQL queries can be provided and executed. Views can be created on top of tables and define a set of transformations that are convenient to reuse or organize query logic [11].

Lastly, the lowest-level data structure in Spark is the RDD, these are immutable, partitioned collections of data records. These records may contain any data structure, although some operations are limited to key-value RDD only. This API provides the developer with a lot of freedom, but the optimizations that are available in the Structured API are absent. Because of this, the RDD API is used mainly when manipulations that are not available at the higher level APIs are not available. Although these APIs differ from the way the developer interacts with them and the amount of integrated logic and optimizations, all Spark managed data compile to RDDs during low-level execution [14, 11].

2.3 Spark SQL

Spark SQL is Spark's module for working with structured data and is an important feature of the Spark ecosystem. It is a feature that lets developers query data from tables and views. This feature is very well unified with the DataFrame and Dataset API, which means that the data can be queried by using SQL statements. Furthermore, the same execution engine is used when using Spark SQL to query data from DataFrame or Datasets as when using their APIs directly.

Unlike the RDD API, Spark SQL includes information about the structure of the data, and planning is carried out to optimize the strategy to achieve the final result [15, 11].

2.3.1 Apache Hive

Apache Hive is a data warehouse solution which serves as a standard for querying large amounts of data in Apache Hadoop. It allows the use of SQL expressions for data stored in the distributed file storage HDFS (Hadoop Distributed File System) using the Hive Query Language (HiveQL).

Furthermore, Hive provides metadata structures on top of HDFS, allowing the use of *partitioning* and *bucketing* of the data. Partitions are used to divide a large table based on common values, creating smaller subsets of the data; this improves performance by only querying the partition of data which contains the query value. To create these partitions, Hive creates directories in HDFS that represent each partition of the data. Buckets work similarly to partitions in that they divide the data into smaller parts, but with some major differences. Instead of organizing data in different directories, buckets are organized into a user-specified number of files that represent *clusters* of the data [16].

Apache Tez

Historically, Hive has used the Hadoop MapReduce execution engine to perform Hive queries. However, in recent years, the default has changed to Apache Tez[1]. For the execution of the queries, the Apache Tez execution engine manages resources using YARN and distributes work, in a similar manner as Spark. A Tez application consists of one Application Master (AM) and multiple containers. The AM is a per-application controller that works similarly to the Spark master, it distributes work, handles requests, and aggregates results. Containers are launched using AM and are a unit of resource allocation that is used to run the execution on [17].

The Hive Metastore

Spark SQL has the ability to connect to a Hive metastore, the part of Hive that contains metadata about tables, as well as reading and writing data to Hive tables. Spark can utilize tables stored in Hive for high interoperability and use partitioning for performance benefits. Although the interoperability with Hive, tables bucketed using Hive's bucketing technique cannot be utilized in Spark; as Spark's technique for the bucketing of tables differ from Hive [18, 19, 11].

2.3.2 Job, Stages, and Tasks

For each action in Spark SQL, there should be one Spark *job*. The job in turn can be divided into multiple *stages*, which themselves can be divided into *tasks*. A Spark Application is able to run jobs in parallel, depending on the resource requirements and size of the cluster of the jobs. When running with default settings, Spark schedules jobs using the FIFO (First In, First Out)

[1]The Apache Software Foundation, Tez Available: https://tez.apache.org/. [Accessed: 2022-04-05]

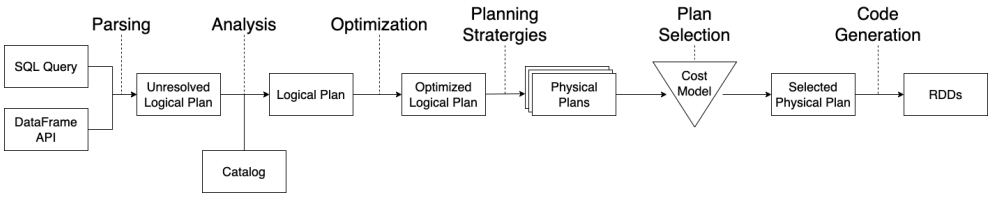


Figure 2.3: Spark's planning process

method, but if the job first in the queue does not require the entire cluster's resources, subsequent jobs may be able to start early.

Stages represents a collection of identical operations working on different subsets of data (tasks), depending on the shuffle partitioning. The stages can be represented by a Directed Acyclic Graph (DAG), where each stage can be of type *shuffle map* or *result*. A shuffle map stage produces input for another stage, while a result stage produces the output that is sent to the driver. Stages can be processed in parallel or sequentially, depending on the interdependence of the operations performed.

Tasks are the smallest unit of execution for a Spark job, representing a set of transformations that will be applied to some blocks of data. Similarly to stages, tasks can be represented as a DAG, and each task outputs a new RDD, which is the result of the previous RDD with the transformations applied. Tasks are executed within executors, and the number of tasks that exist in a stage depends on the number of shuffle partitions of the data [20, 11].

2.3.3 Planning and Strategies

Planning a series of transformations involves two steps, the *logical plan*, which describes *what* operations are performed, and the *physical plan*, which describes *how* these operations are performed. A high-level view of how the planning process works internally in Apache Spark can be observed in Figure 2.3. The steps presented in the illustration will be described in detail.

The planning starts with the computation of some relation which the developer has declared either by using a SQL query or by using the DataFrame API. If a SQL query has been used, then the Spark SQL parser processes it and returns an Abstract Syntax Tree (AST). Since the computed relation may include attribute references, it is required that these references are validated and resolved against the *catalog object*. The catalog object contains the metadata of the tables and columns so that the attributes used in the relation can be resolved and type information can be added. If the attributes cannot be resolved, for example, referring to a column that does not exist, the analyzer

will throw an error.

In the optimization phase, *rule-based optimizations* are applied to the resolved logical plan. A rule-based optimization is a transformation that modifies the logical plan into an improved logical plan, without modifying the end result of the plan. Some optimizations that can be applied are constant folding, predicate pushdown, projection pruning, null propagation, and expression simplification.

The strategy planning phase has the responsibility of transforming the logical plan into one or more physical plans using a *strategy*. For example, if the logical plan includes a step consisting of a join operation, the strategy would transform the logical join into the join algorithm(s) that suits the constraints of the join. After the physical plans are generated, it selects the best plan using a cost model.

Finally, using the selected physical plan, the Java bytecode is generated in a phase called *code generation*. Code generation is a feature that enables faster execution time since the interpreted expressions are pre-evaluated in the resulting bytecode and do not require additional interpretation. Additionally, starting from Spark 3.0, whole-stage code generation is also available. Whole-stage code generation further improves code generation by collapsing multiple trees of dependent physical operators into a single function [21, 22, 23].

2.3.4 Transformations

The transformations performed in Spark SQL are evaluated *lazily*; this means that the computations are executed exactly when a result is required by the user. The trigger to these computations is called an *action*; to name a few actions, there is `count`, which counts the number of rows within a DataFrame, `show`, which prints a specified amount of rows to the console, and any operation that writes data to storage [24].

Furthermore, instead of executing each transformation as it is written, Spark stacks all statements to build up the initial logical transformation plan, which consists of a high-level abstraction of the operations to be performed.

2.3.5 Shuffle

To make executors work on the same collection of data in parallel, it is necessary to partition the data into chunks that are assigned to executors, in an operation called *shuffling*. Effectively, this is a physical partitioning of the data. The use of this operation is common when dealing with key-based

transformations or operations that are executed on subsets of the data, for example, joining with equal keys.

When a shuffle is executed, the data is written to disk and transferred across the network, making it a costly operation to perform and can substantially increase the execution time of Spark jobs. However, this operation can be avoided if the data operated upon is bucketed, then the different partitions of the data can be divided among the executors by utilizing the structure of the buckets [11].

2.3.6 Joins

Spark SQL currently has support for seven join operations, as presented in [25]:

- Inner join - Select rows with matching values from both relations.
- Left (outer) join - Select the values of the left relation and the matched values of the right relation.
- Right (outer) join - Select the values from the right relation and the matched values from the left relation.
- Full join - Select all values from both relations, appending NULL to the values that do not have match.
- Cross join - Return the Cartesian product of the two relations.
- Semi join - Return values from the left side of the relation that has a match with the right relation.
- Anti join - Return values from the left relation that do not match with the right relation.

Depending on the type of join and the structure of the data, different join strategies are used. These are:

- Broadcast Hash Join
- Broadcast Nested Loop
- Cartesian Product
- Shuffled Hash Join

- Sort-Merge Join

By the usage of join hints, the developer may suggest a strategy to prioritize, supported ones are `BROADCAST`, `MERGE`, `SHUFFLE_HASH`, and `SHUFFLE_REPLICATE_NL`. However, providing a join hint does not guarantee that the strategy will be chosen, as the hinted strategy may not support the join type [26].

With these strategies in mind, as can be observed in [27], Spark internally considers the following prioritization when planning:

- If equi-join
  - If join-hint provided
    1. `BROADCAST` - Pick Broadcast Hash Join if the join type is supported.
    2. `MERGE` - Pick Sort-Merge Join if keys are sortable.
    3. `SHUFFLE_HASH` - Pick Shuffle Hash Join if the join type is supported.
    4. `SHUFFLE_REPLICATE_NL` - Pick Cartesian product if an inner-like join.
  - If not join hint provided
    1. Pick Broadcast Hash Join if one side is small enough to broadcast and the stated join type is supported.
    2. Pick shuffle Hash Join if one side is small enough to create a local hash map.
    3. Pick Sort-Merge Join if the join keys are sortable.
    4. Pick Cartesian Product if the join type is inner-like.
    5. Pick Broadcast Nested Loop.
- If non-equi-join
  - If join-hint provided
    1. `BROADCAST` - Pick Broadcast Nested Loop.
    2. `SHUFFLE_REPLICATE_NL` - Pick Cartesian product if the join type is inner-like.
  - If not join hint provided
    1. Pick Broadcast Nested Loop if one side is small enough to broadcast.
    2. Pick Cartesian product if the join type is inner-like.
    3. Pick Broadcast Nested Loop.

2.3.7 Windowing

Aggregation of data in Spark SQL is commonly achieved using *window functions*. A window function calculates *window frames*, which are defined using a reference to the current row of data. In practice, this means that a single row can exist in any number of frames, depending on the specifications for the window. A common use case for these operations is rolling averages, where the window slides across the data when aggregating the results. A window specification can specify different partitions, or groups, of the data by using the `partitionBy` statement. In addition, ordering of data within a window frame using the statement `orderBy`. When defining the number of rows each window frame should contain, one can use the `rowsBetween` statement, where the input is the number of rows before and after the current row should be included.

When a window specification is defined, an aggregation function can be specified to aggregate the rows within a window frame to an output. Some of the aggregation function that are supported by Spark are: `max` and `min`, obtain the max value of some column in the frame, `sum`, sum the values of a column, and `avg`, the average of the values of a column. In addition to these aggregation functions, there are also ranking and analytical functions such as: `rank`, the rank of the rows within a window partition based on the ordering, `row_number`, the number of rows within a window partition.

2.4 Related Work

In this section, different implementations of the PIT join operator are presented; both that are developed on top of Spark and for other libraries.

2.4.1 Existing Spark Point-in-Time Implementations

There have been previous implementations for exposing functionality in Spark to make PIT-joins possible; the most significant ones being Databrick's time series utility library *tempo*[1], and Two Sigma's *Flint*[2].

The tempo library implements a PIT-join by taking the union of the left and right DataFrames, prefixing the table's column names with a specified

[1]Databricks Inc., tempo - Time Series Utilities for Data Teams Using Databricks Available: https://github.com/databrickslabs/tempo. [Accessed: 2022-01-31]

[2]Two Sigma LCC, Flint: A Time Series Library for Apache Spark Available: https://github.com/twosigma/flint. [Accessed: 2022-02-18]

string. To determine which table a row is from, another column is added as the table identifier `rec_ind`. A table-agnostic timestamp `combined_ts` is constructed for each row, which, together with the table identifier `rec_ind`, is used to sort the data. If a row in the left table has the same timestamp as any row in the right table, the left row is ordered below the right row. The final table is calculated by using a Spark window and for each row from the left table, spanning the window frames from the beginning of the table up to the current row, and then merging that row with the *last* non-null values of the right columns, within that window partition. Last, in this case, lowest ordered. The main parts of this implementation, written in pseudocode, can be observed in Algorithm 1 [28].

While tempo utilizes higher-order Spark SQL functionality to make the transformations possible, Flint applies the logic on the RDD level. This is done using a custom RDD implementation `OrderedRDD`, which is a RDD whose data are known to be ordered; this is useful when working with join operators, since the data do not need to be re-sorted. Partitions within an `OrderedRDD` are defined as splits of independent ranges of key-value pairs, defined as open-closed ranges. In the case of joining operations, both RDDs partitions are ordered in ascending order, and the partition ranges from the left are applied to the right RDD, with an included tolerance (for example, 1 day), creating overlapping partitions for the right RDD. For each of the entries in a partition of the left RDD, searching for a match in the right RDD's partition is executed using a peekable iterator, which searches for the entry in the right RDD that has the highest key, which also is less than or equal to the right key.

By utilizing a peekable iterator, the algorithm looks at the next entry's key and compares it to the right key; if the right key is greater, the last value seen is the correct match, since the sorting is in ascending order. Because the partitions applied to the right RDD are the left partitions with the addition of a tolerance interval, this algorithm cannot work without a manually specified tolerance input. A benefit of this approach is that it is highly parallelizable, as the partitions can be worked on independently within Spark. However, the downside of this process is that the generation of splits within each RDD is done by sampling the data, which could be expensive. Furthermore, if the specified tolerance is very large, the number of iterations and comparisons may increase drastically, depending on the data [29].

2.4.2 Non-Spark Point-in-Time Implementations

Apart from the existing implementations made on top of Spark, the source code for the data processing tools Pandas and JuliaDB.jl can provide a more general view on how PIT joins may be implemented.

The implementation used by Pandas can be obtained by analyzing the function `asof_join_backward`, which is used to determine which right row each of the left rows maps to. Pandas then uses these index mappings to create a new Pandas DataFrame. The algorithm for obtaining the index mapping, assuming that no tolerance is allowed (maximum difference between the left and right timestamps) and that exact values are allowed, is presented in Algorithm 2 [30].

The algorithm used by JuliaDB.jl is similar to that used in Pandas, with the only difference that it only allows for outer-left joins, meaning that if a left row does not match with any of the rows of the right table, the left row will still be added to the results, with all column values of the right table set to `null` [31].

Algorithm 1: PIT join implementation Tempo

Data: Left and right DataFrame, $left$ and $right$. Column prefixes, $leftPrefix$ and $rightPrefix$

begin

$leftIndexed \leftarrow left.withColumn(rec\_ind, 1)$
$rightIndexed \leftarrow right.withColumn(rec\_ind, -1)$

$prefixedLeft \leftarrow leftIndexed.prefixColumns(leftPrefix)$
$prefixedRight \leftarrow$
$rightIndexed.prefixColumns(rightPrefix)$

$combined \leftarrow prefixedLeft.union(prefixedRight)$
$combinedWithTS \leftarrow combined$
$.withColumn(ts, left.ts \text{ or } right.ts)$

$windowSpec \leftarrow Window$
$.orderBy(combinedWithTS.ts, rec\_ind)$
$.rowsBetween(-Int.MaxValue, currentRow)$

$rightColumns \leftarrow prefixedRight.observationColumns +$
$prefixedRight.tsColumn$

$result \leftarrow rightColumns.foldLeft(combined)\{$
$(dataFrame, column) \rightarrow$
$dataFrame.withColumn(column,$
$last(column, ignoreNull \leftarrow true)$
$.over(windowSpec)$
$)$
$.filter(leftTS \text{ is not } null)$
$\}$
return $result$

Algorithm 2: Backward ASOF join Pandas

Data: Ordered arrays of left and right values, $leftValues$ and $rightValues$.
Result: Indexed mapping.

```
1  begin
2      leftIndexer[0..leftValues.length]
3      rightIndexer[0..rightValues.length]
4      rightPos ← 0
5      for leftPos ← 0 to leftValues.length do
6          if rightPos < 0 then
7              rightPos ← 0
           /* Try to find the last right value that
              is smaller or equal to left value   */
8          while rightPos < rightValues.size and
            rightValues[rightPos] <= leftValue[leftPos] do
9              rightPos ← rightPos + 1
10         rightPos ← rightPos − 1
           /* Store the mappings in the indexers  */
11         leftIndexer[leftPos] ← leftPos
12         rightIndexer[leftPos] ← rightPos
13     return leftIndexer, rightIndexer
```

Chapter 3

Method

This chapter aims to present the research methodology and evaluation methodology of the different implementations and optimizations made on the PIT joining process. To be able to answer the research question *"How to perform fast and resource efficient Point-in-Time joins in Apache Spark?"*, it is important to specify the empirical and research methods used. The comparison of these processes must provide reliable data that can be used to draw conclusions.

3.1 Research Process

The research process is divided into seven steps that can be observed in Figure 3.1. The first step of the process is the selection of algorithms that will be evaluated; this step will be partially based on previous known implementations of PIT algorithms, as presented in Section 2.4, as well as the implementation of join algorithms already present in Spark, as presented in Section 2.3.6. These algorithms will be analyzed using a cost analysis model that can identify their weaknesses and potential bottlenecks. From the cost analysis, optimizations to the algorithm can be identified; if the optimizations affect the cost analysis in any way, a new cost analysis is made with these optimizations taken into account. When the main possible optimizations are identified, the algorithms will be implemented in Spark. All algorithms will be deployed in a Spark cluster and, using and not using the optimizations, will be run on the generated test data. The metrics collected from the experiments will then proceed to be processed and analyzed to be able to draw conclusions about the different implementations and optimizations.

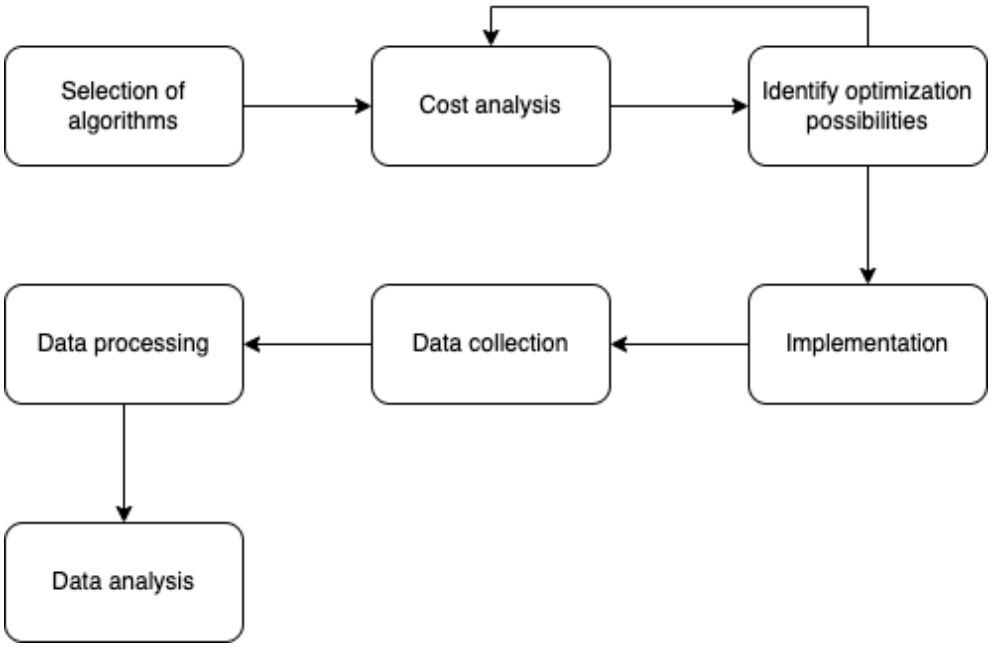


Figure 3.1: Research process

3.2 Cost Analysis Model

The cost analysis model will be heavily based on the cost model for the join operator as presented by Lian and Zhang in [6]. This cost model takes into account the internal workings of Spark and the structure of the cluster on which Spark is deployed. In addition to space and time complexity, this model accounts for network transmission cost, disk operation, data size, number of cluster nodes, shuffle buffer size, and the average size of a row of data. However, since this cost model focuses primarily on built-in join algorithms, it will have to be extended to account for the processes that will be implemented in this project.

3.3 Experimental design

The experiments will be conducted by running the algorithm using data of varying properties on a cluster of varying sizes. Further details of the data and the execution of the experiments will be provided in this section.

Table 3.1: Example of a row in the dataset

ID	Timestamp	Value / Label
1	10	”Some value”

3.3.1 Datasets

The datasets used in the experiments will aim to simulate possible datasets in which a PIT join may be used. In particular, the data are designed to replicate timed feature values along with a prediction target that will be joined, to form input data for a machine learning model. In Table 3.1 an example of a row from these datasets can be observed. The timestamp value is a non-negative integer, which represents the elapsed time from some starting point; for simplicity sake, the starting point is always zero and represents seconds. In the left table, each entry will consist of a unique ID, along with a label (the prediction target for that ID at the given timestamp). Each unique ID in the left table will be assigned to an arbitrary number of rows in the right table, representing the feature update events for that entity.

Significant variables

The significant variables identified are as follows:

- Maximum timestamp value
- Number of unique IDs, for example, number of users
- Distribution of timestamps for entries in the left table
- Number of entries in the right table that correspond to an entry in left table
- Distribution of timestamps for entries in the right table

Generation of data

When generating the datasets, the maximum timestamp will be a configurable variable that determines the interval of the recording of events, for example, six months, one year, or two years. For the chosen maximum timestamp, multiple datasets with different numbers of unique IDs will be generated, essentially determining the size of the left dataset. The target event time for each of the rows in the left dataset will be chosen uniformly at random.

The number of events per ID generated is determined by using a normal distribution. Event timestamps will use two different sampling methods: normal distribution, to simulate special events, and uniform distribution, to simulate random updates over time. For the normal distribution, the configuration values are chosen uniformly at random, with the mean chosen between zero and the maximum timestamp, and the standard deviation between five and 15 days.

The number of unique IDs that will be generated are 10,000, 100,000, 1,000,000, and 10,000,000, which will be generated with a timestamp interval of one year, i.e., the maximum timestamp is $60 \cdot 60 \cdot 24 \cdot 365$. Furthermore, two different configurations for the normal distribution used to decide the number of feature updates are used, one where the mean is 20 and a standard deviation of two and one with the mean of 80 and a standard deviation of eight. Furthermore, there is one configuration with a uniform distribution of feature event timestamps and one with a normal distribution. To avoid generating many different permutations of the configurations, each configuration is chosen uniformly at random for each user id.

The library to be used for generation is `numpy.random`[1] using a set seed for the pseudo random number generator.

Data variations

Since data organization can affect the performance of data-intensive tasks, different variations of the generated datasets will be used. In addition to the direct output of the data generation program, the data will also be stored using three different sorting orders: ascending according to the ID and timestamp, descending according to the ID and timestamp, and randomized. To also test the performance on pre-bucketed data, which reduces the performance impact of the shuffle operation, as described in Section 2.3.5, the dataset sorted in ascending order will be bucketed into different number of buckets and bucketed based on the ID of each row. The number of buckets that will be tested is 20, 40, 80, and 160 buckets. Since both tables contain the same number of IDs, the resulting buckets of the left and right tables should contain the same IDs. Furthermore, for all experiments, the shuffle partition will be set to 200 partitions, when not utilizing bucketing, the default used in Spark.

[1] Numpy Developers, Random sampling (numpy.random) — NumPy v1.22 Manual Available: https://numpy.org/doc/1.22/reference/random/index.html. [Accessed: 2022-02-28]

Data format and storage

The datasets will be formatted using the *Apache Parquet* columnar storage format[1] and will be stored using the *Amazon S3* service[2]. These formatting and storage options were chosen because they are some of the most common storage options used when working with Spark [32]. Since bucketed tables require a Hive metastore to be able to store and load the metadata concerning bucketing partitions, which cannot be stored in S3. Because of this, *HopsFS*[3] will be used as a metastore.

3.3.2 Hardware

As mentioned above, it is important that the hardware consists of a cluster of nodes, as this is the most common way for Spark deployment [13]. The experiments will run in clusters of one, two, and three nodes, each hosting two to three executors each. This will provide relevant benchmarks that can be analyzed to determine how the implementation scales depending on the cluster size. The machines used are provided using *Amazon EC2* instances, utilizing the *m5.xlarge* instance type. These instances utilize two physical processor cores, each with a maximum clock speed of 2.5 gigahertz and two threads (four vCPUs in total), 16 gigabytes of memory, up to 10 gigabits per second of network bandwidth, and up to 4,750 megabits per second of disk bandwidth[4].

Each of the worker instances will occupy two to three executors each; where each executor will utilize two gigabytes of memory and one virtual core. The driver process for the Spark Application will also need to reside in a worker node and will also require gigabytes of memory and one virtual core and will reside in one of the worker nodes.

[1]The Apache Software Foundation, Apache Parquet Available: https://parquet.apache.org/. [Accessed: 2022-02-24]

[2]Amazon Web Services, Inc., Cloud Object Storage – Amazon S3 Available: https://aws.amazon.com/s3/. [Accessed: 2022-02-24]

[3]Logical Clocks AB, HopsFS Available: https://www.logicalclocks.com/products/hopsfs. [Accessed: 2022-03-07]

[4]Amazon Web Services, Inc., Amazon EC2 M5 Instances - general purpose compute workloads Available: https://aws.amazon.com/ec2/instance-types/m5/. [Accessed: 2022-03-07]

3.3.3 Software

To make the configuration of the Spark cluster more straightforward and enable the use of HopsFS, the Hopsworks.ai[1] managed platform will be used. By using Hopsworks, the cluster can be scaled using different numbers of executor nodes between experiments and run a Jupyter server, which will be used to initiate the Spark jobs.

Data collection will be achieved using the library *SparkMeasure*[2]. This library adds functionality to be able to easily receive aggregated metrics for Spark stages and tasks that can then be used for analytical purposes. Some of the metrics that will be most notable for the purpose of this project are execution time, result size (bytes sent from executor to driver), reads and writes in memory, and disk writes and reads [33]. The complete list of the specific metrics that will be collected and examined, together with their purpose, can be seen in Table 3.2.

Table 3.2: Metrics to be examined

Name	Description	Purpose
Elapsed time	The total duration of time that is takes to retrieve the results	Examine how the algorithms scale based on time
Peak execution memory	Accumulative peak memory used by internal data structures when handling tasks	Examine the total memory utilization
Shuffle bytes written	Bytes written to disk as a part of data shuffle	Examine the quantity of shuffles and disk operations
Memory data spill	The unserialized form of spilled data as it exists in memory	Examine the quanitity of spilled data
Disk data spill	The serialized form of spilled data as it exists in disk	Examine the quantity of spilled data

[1]Logical Clocks AB, Hopsworks.ai Available: https://docs.hopsworks.ai/hopsworks-cloud/latest/. [Accessed: 2022-03-07]

[2]Canali, Luca, sparkMeasure Available: https://github.com/LucaCanali/sparkMeasure. [Accessed: 2022-02-23]

3.3.4 Hive Setup

To better show the effect of using this functionality in Spark, a comparison with the widely popular Apache Hive data warehouse solution will be included. Similarly to Apache Spark, Apache Hive allows the querying of large amounts of column-based data using SQL-like syntax, as presented in Section 2.3.1. For the execution of Hive queries, the Tez execution engine will be used. For these experiments, Tez will use three physical nodes and AM will be configured to use one vCPU and 2048 megabytes of memory, while the containers will use one vCPU and 2048 megabytes of memory each.

3.4 Reliability and Validity of the Data

In this section, the method used for data collection and analysis will be evaluated in terms of validity and reliability. It is important that the data recovered using the presented method is reliable and that the analysis is executed in a meaningful and accurate way.

3.4.1 Data Validity

To determine whether the work, which is derived from the executors, is the expected work, it is important to examine Spark *physical plan*, discussed in Section 2.3.3. Using the Spark SQL `EXPLAIN` statement, it is possible to obtain the physical plan of a series of transformations that Spark will apply to the input data. It is important that the physical plan is examined before retrieving any experiment results, so that the output metrics can be related to a series of physical operations performed by Spark.

3.4.2 Reliability of Data

SparkMeasure reliably extracts data from the *Spark Listener API*, which transports data from the executors to the driver; these listeners are used internally for the *Spark Web UI* and the history server, both provided by the standard Spark distribution. Using this library, a reliable view of how work is derived within executors can be achieved [34].

Since Spark transformations are evaluated lazily, it may be difficult to entirely isolate the work of the algorithm into a single Spark job. To try to reduce the impact of this, the datasets will persist in memory using the DataFrame `persist()` operation. Since it is not possible for the scope of

this project to account for all possible deviations in possible datasets that these algorithms can use, the results cannot be applied to every use case.

3.4.3 Evaluation Method

The method used to evaluate the results will be mainly comparative. Meaning that the results from each of the algorithms will be compared to each other based on the individual experiment metrics as well as scalability as the data size and the number of nodes grow. For analytical purposes, the mean of the experiments will be used for evaluation while recognizing potential deviations observed for different iterations of the same experiments. The reliability of the data will be analyzed by investigating how the result deviated between iterations of the experiments, aiming to be as small as possible.

Chapter 4

Implementation

This chapter will present details on the implementation of the algorithms used to obtain PIT datasets, as well as a cost analysis of the processes.

4.1 Algorithm Selection

On the basis of previous work, three different implementations of the PIT join operation were decided. All three of these algorithms differ in terms of the implementation complexity and the amount of customization required for SparkSQL. The first one is called the *Exploding PIT join*, which is the most naive approach of these three, whose implementation requires simple operations available in SparkSQL. The second one is called the *Union PIT Join*, whose implementation is highly based on the implementation of the join operator in the Tempo library, as presented in Section 2.4.1. The last, called *Early Stop Sort-Merge*, is an implementation that exposes a new low-level join operator to Spark, highly inspired by the implementation in the Pandas library, as presented in Section 2.4.2.

4.1.1 Exploding Point-in-Time Join

The Exploding PIT join can be seen to be the easiest one to implement in SparkSQL, as it consists of only four steps. The following steps assume that the datasets contain event timestamps and that the rows also require to be matched by ID:

1. Inner join of left and right dataset on the condition that `left.ts >= right.ts`, other equality conditions may also be used in addition with this, for example equality of IDs.

Listing 4.1: HiveQL query for a Point-in-Time join

```
1  SELECT id, target_ts, label, value, feature_ts FROM (
2      SELECT id, target_ts, label, value, feature_ts,
           ROW_NUMBER() OVER (PARTITION BY id, target_ts
           ORDER BY feature_ts) rn FROM (
3           SELECT {left}.id as id, {left}.ts as target_ts,
                {left}.label as label, {right}.value as value,
                {right}.ts as feature_ts
4           FROM {left} {left} INNER JOIN {right} {right}
5           ON {left}.id = {right}.id AND {left}.ts >=
                {right}.ts
6      ) combined
7  ) windowed
8  WHERE rn = 1
```

2. Create window specification, partitioning the data using `left.ts` and ordering the data in descending order based on the right table's timestamp. If equality condition is used in previous step, those columns should also be used for partitioning.

3. Rank each of the rows in the combined dataset based on their row number within each window frame.

4. Filter the rows whose row number is not equal to `1`.

Essentially what this algorithm does, it generates *all* possible PIT candidates by performing the first inner join, and then only picking the candidate where the right timestamp is as close as possible to the right timestamp. The name *exploding* is used since the generation of all possible candidates results in an exponentially increasing intermediate table of the candidates as the size of the left or right table increases. The Scala implementation of this algorithm with Spark can be observed in Appendix A.1.

Hive Query

For this implementation, an additional version was created using HiveQL, as presented in Listing 4.1. This query will be used to benchmark the difference in performance when comparing the Hive execution with the Spark execution.

4.1.2 Union Point-in-Time Join

As mentioned above, the Union PIT join essentially works as the join implemented in *Tempo*, with minor modifications. The steps included in this algorithm are the following:

1. Prefix the columns of the right table, optionally prefix the left columns.
2. Add a index column to both tables, set it to 1 for the left and 0 for the right.
3. Take the union of both tables.
4. Add add global timestamp column, with the value being `left.ts` or `right.ts`, depending which is defined for each row.
5. Create a window specification that defines the following configuration:
    - Order by the global timestamp column and the table index column, in ascending order, lower timestamp should be at the top and rows from the right table should be above the ones from the left table.
    - For each window frame, it should select the current row and *all* the preceding rows.
    - Optionally define partitioning columns, for example, ID.
6. For each column belonging to the right table in the combined table, apply the windowing specification and take the last non-null value of the column.
7. Filter away the rows where the left values are `null`, these are the results by applying the windowing function on rows from the right table.

An illustration of how the selection process is executed within a window frame can be observed in Figure 4.1. The Scala implementation of this algorithm with Spark can be observed in Appendix A.2.

4.1.3 Early Stop Sort-Merge Join

Early Stop Sort-Merge is a modified low-level joining mechanism designed specifically for Spark. The logic behind this mechanism is inspired by the implementation of the `merge_asof` in Pandas, as presented in Section 2.4.2. The abstract functionality and workings of this join mechanism are as follows:

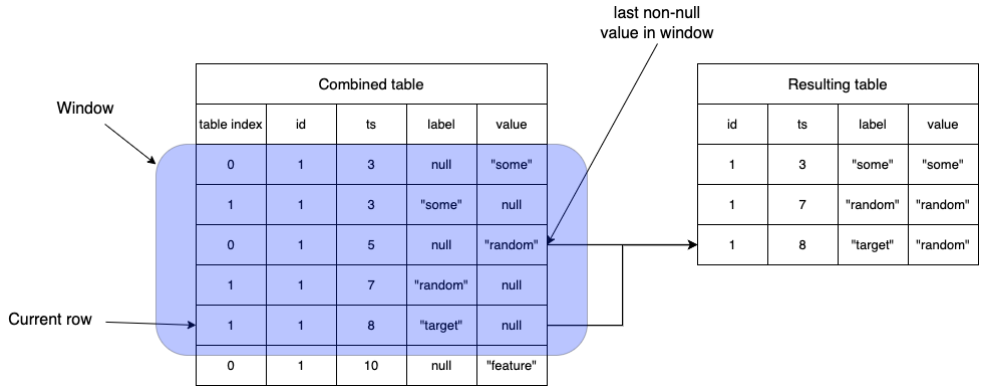


Figure 4.1: Point-in-Time merge using window frames

1. Determine the left and right columns should be used for the PIT predicate, `left.ts` and `right.ts`.
   - Also determine one or more equality conditions, these will be used for partitioning of the data.
   - Other non-equality conditions are not allowed.
2. Partition (shuffle) the left and right dataset according to the columns used for the equality condition.
3. Sort both tables in descending order, based on the columns `left.ts` and `right.ts`.
4. Using the underlying RDDs of each pair of table partitions from the left and right table:
   (a) Select the next RDD of the left and right iterator.
   (b) Until the end of either the left or right iterators are reached, to the following:
      - If `left.ts < right.ts` or the right partitioning column has a greater value than the left one, select the next right RDD.
      - If the left partitioning column is greater than the right partitioning column, select the next left RDD.
      - Otherwise, join the two RDDs and select the next left RDD.

Considering that this algorithm directly affects how the particular join is performed on the RDD level, it is not sufficient to implement it using the Spark SQL API. Because of this, the implementation has been made

Listing 4.2: Logical plan for the Point-in-Time join

```
1 case class PITJoin(
2     left: LogicalPlan,
3     right: LogicalPlan,
4     pitCondition: Expression,
5     condition: Option[Expression]
6 ) extends BinaryNode
```

using custom built logical and physical plans, extending on the existing implementation of the Sort-Merge Join, as well as a custom join strategy for selecting the physical plan. This is made possible by utilizing the `ExperimentalMethods` instance of the `SparkSession` object. This makes it possible to inject custom strategies and optimizations of logical plans that can be applied to the planning process at runtime [35]. The optimizer injects the logical plan by substituting a join utilizing a registered UDF (User Defined Function), registered under the name `PIT`; the logical plan can be observed in Listing 4.2. Using a registered UDF, the functionality is available using both the DataFrame API and the SQL interpreter.

4.1.4 Source code

The source code for these implementations are publicly available on Github[1]. The Github repository also contains instructions for how to utilize the implementations for other Spark projects.

4.2 Execution Plans

The execution plans for all algorithms can be observed in Appendix B. What can be observed is that the Exploding PIT join could take on different plans, due to the selection of the joining technique. In particular, when any of the tables is small enough to fit within the broadcasting limit, the Broadcast Hash Join technique can be used, which does not utilize the shuffle operation. However, when none of the tables are small enough to be broadcasted, both datasets are required to be shuffled, possibly impacting the performance.

[1] Pettersson, Axel, Spark PIT: Utility library for Point-in-Time joins in Apache Spark Available: https://github.com/Ackuq/spark-pit. [Accessed: 2022-05-12]

Table 4.1: Significant variables for cost analysis

Symbol	Definition
O	Time complexity
S	Space complexity
P	Network cost
D	Disk I/O cost
L	Left table
R	Right table
$R_{columns}$	Amount of columns in right table
N	Number of nodes
$Buffer_{shuffle}$	Buffer size of shuffle operator
K	Row selectivity

Considering the Union PIT join, its execution plan is heavily affected by the number of columns that are used in the right table, since each column requires its own windowing operation. Furthermore, when the unionized dataset is large, each of these windowing operations may require a shuffle before execution. This could affect the performance of this algorithm as more columns are added.

The Early Stop Sort-Merge Join has an execution plan similar to that of the Spark native Sort-Merge Join. Before performing the joining algorithm, both datasets are sorted and possibly shuffled, depending on if the datasets are pre-shuffled or not.

4.3 Cost Analysis

As aforementioned, the cost analysis is based on previous work by Lian and Zhang in [6] and its purpose is to better understand the working of the algorirthms and their theoritical performance. For this analysis, eight significant variables have been identified, which can be observed in Table 4.1. The time and space complexity are represented by O and S respectively, and in addition to these complexities, the cost of network and disk I/O will also be considered in the analysis, represented by P and D respectively. The reason for not only settling on time and space complexity is because that network and disk I/O are time consuming operations which may affect the overall performance of a Spark job. Furthermore, N represents the number of nodes running executors and K represents the row selectivity of a join operation, which is the fraction of rows in each table that match the joining condition.

4.3.1 Common Operators

Some operations performed by the proposed processes are shared, such as the shuffle operation; these operations will be considered in this section.

Shuffle Operator

As described in [6], there are two phases when executing a shuffle operation, the *map phase* and the *reduce phase*. In the map phase, the mapper nodes receive different partitions of the data (if any) and generate a number of buckets, that is, physical partitions of the data that will be mapped to different executors. In the reduce phase, the output of the mapper nodes is reduced and sent to the subsequent task. Derived from [6], the time complexity can be observed in Equation (4.1), the space complexity in Equation (4.2), the cost of the network in Equation (4.3), and the cost of the disk I/O in Equation (4.4).

$$O_{shuffle}(T) = |T| \times \log(|T|) \tag{4.1}$$

$$S_{shuffle}(T) = Buffer_{size} \tag{4.2}$$

$$P_{shuffle}(T) = |T| \tag{4.3}$$

$$D_{shuffle}(T) = \max(|T| - Buffer_{size}) \times N, 0) + |T| \tag{4.4}$$

Hash Join

The Hash Join operator is used for the implementation of the Broadcast Hash Join and the Shuffled Hash Join operator. It works by hashing the join keys of the left and right tables and adding them to an in-memory hash table. Rows whose primary key hash to the hash table entry are then compared and finally merged to create the resulting rows. As derived from [6], the time and space complexity can be observed in Equations (4.5) and (4.6), respectively.

$$O_{HashJoin}(L, R) = \frac{|L| + |R|}{N} \tag{4.5}$$

$$S_{HashJoin}(L, R) = \frac{|L| + |R|}{N} \tag{4.6}$$

Broadcast Hash Join

This variation of the Hash Join algorithm is used when at least one of the tables are small enough to be broadcasted to the worker nodes. The broadcasted table is shared by all worker nodes while the larger table is shuffled. As derived

from [6], the time and space complexity can be observed in Equations (4.7) and (4.8), and the network and disk I/O cost can be observed in Equations (4.9) and (4.10).

$$O_{BHJoin}(L, R) = O_{HashJoin}(L, R) \tag{4.7}$$

$$S_{BHJoin}(L, R) = S_{HashJoin}(L, R) + |L| \tag{4.8}$$

$$P_{BHJoin}(L, R) = N \times K \times |L| \tag{4.9}$$

$$D_{BHJoin}(L, R) = D_{shuffle}(R) \tag{4.10}$$

Shuffle Hash Join

The Shuffle Hash Join includes shuffling of both the left and right tables and performs a regular Hash Join within each shuffle partition. As derived from [6], the time and space complexity can be observed in Equations (4.11) and (4.12), and the cost of network and disk I/O can be observed in Equations (4.13) and (4.14).

$$O_{SHJoin}(L, R) = O_{shuffle}(L) + O_{shuffle}(R) + O_{HashJoin}(L, R) \tag{4.11}$$

$$S_{SHJoin}(L, R) = S_{shuffle}(L) + S_{shuffle}(R) + S_{HashJoin}(L, R) \tag{4.12}$$

$$P_{SHJoin}(L, R) = K \times |L| \times N + K \times |R| \times N \tag{4.13}$$

$$D_{SHJoin}(L, R) = D_{shuffle}(L) + D_{shuffle}(R) \tag{4.14}$$

Sort-Merge Join

The Sort-Merge Join consists of three steps, shuffle of the data based on the join key, sorting of the data within the shuffle partitions, and lastly, traversing of the data within each partition to find matching rows. As derived from [6], the time and space complexity can be observed in Equations (4.15) and (4.16), and the network and disk I/O cast can be observed in Equations (4.17) and (4.18).

$$O_{SMJoin}(L, R) = O_{shuffle}(L) + O_{shuffle}(R) + \frac{|L| + |R|}{N} \tag{4.15}$$

$$S_{SMJoin}(L, R) = S_{shuffle}(L) + S_{shuffle}(R) \tag{4.16}$$

$$P_{SMJoin}(L, R) = P_{shuffle}(L) + P_{shuffle}(R) \tag{4.17}$$

$$D_{SMJoin}(L, R) = D_{shuffle}(L) + D_{shuffle}(R) \tag{4.18}$$

Sorting

By default, Spark uses Timsort [36] for sorting in Spark SQL, where, in the case of shuffled data, sorting is performed on each shuffle partition. Hence, the average time complexity for this process can be observed in Equation (4.19).

$$O_{shuffle}(T) = \frac{|T| \times \log(|T|)}{N} \tag{4.19}$$

4.3.2 Exploding Point-in-Time Join Operator

As mentioned in Section 4.1.1, the steps required to execute this process are joining, sorting, and aggregating. The result of the joining operation, J, has an exponentially increasing size, since the joining criteria match all rows in the right table where the timestamp of the left table is greater and the joining keys match. The equation for the size of J can be observed in Equation (4.20).

$$|J| = |L| \times |R| \times K \tag{4.20}$$

The aggregation step is applied to each window partition in the joined table, where the number of window partitions equals the number of unique join keys in the left table. Hence, it will have a time complexity of $\frac{L}{N}$.

Taking into account the best-case scenario, when the left table is small enough to be broadcasted, no shuffle operations are required. The resulting space and time complexity can be observed in Equations (4.21) and (4.22), as well as the network and disk I/O cost observed in Equations (4.23) and (4.24).

$$O_{ExplodingBest}(L, R) = O_{BHJoin}(L, R) + O_{sort}(J) + \frac{L}{N} \tag{4.21}$$

$$S_{ExplodingBest}(L, R) = S_{BHJoin}(L, R) \tag{4.22}$$

$$P_{ExplodingBest}(L, R) = P_{BHJoin}(L, R) \tag{4.23}$$

$$D_{ExplodingBest}(L, R) = D_{BHJoin}(L, R) \tag{4.24}$$

In the worst case, when neither dataset is small enough to be broadcasted, both tables require shuffling as well as the resulting table. Considering that the operator used is the Sort-Merge Join operator, the resulting space and time complexity can be observed in Equations (4.25) and (4.26), as well as

the network and disk I/O cost observed in Equations (4.27) and (4.28).

$$O_{ExplodingWorst}(L,R) = O_{SMJoin}(L,R) + O_{shuffle}(J) + O_{sort}(J) + \frac{L}{N} \tag{4.25}$$

$$S_{ExplodingWorst}(L,R) = S_{SMJoin}(L,R) + S_{shuffle}(J) \tag{4.26}$$

$$P_{ExplodingWorst}(L,R) = P_{SMJoin}(L,R) + P_{shuffle}(J) \tag{4.27}$$

$$D_{ExplodingWorst}(L,R) = D_{SMJoin}(L,R) + D_{shuffle}(J) \tag{4.28}$$

4.3.3 Union Point-in-Time Join

When executing the Union Point-in-Time join, it performs a union step and then a number of repeated windowing operations to populate the right column values of each row from the left table. Each of these window specifications requires a shuffle of partitions and sorting of the data, as defined by the window specification. Due to these repeated windowing operations, the number of columns in the right table can affect the performance of this process. The union process itself requires scanning both tables to create the resulting table. The resulting space and time complexity can be observed in Equations (4.29) and (4.30), as well as the network and disk I/O cost observed in Equations (4.31) and (4.32).

$$O_{UnionJoin}(L,R) = \frac{|L \cup R|}{N} + (O_{shuffle}(L \cup R) + O_{sort}(L \cup R)) \times R_{columns} \tag{4.29}$$

$$S_{UnionJoin}(L,R) = S_{shuffle}(L \cup R) \times R_{columns} \tag{4.30}$$

$$P_{UnionJoin}(L,R) = P_{shuffle}(L \cup R) \times R_{columns} \tag{4.31}$$

$$D_{UnionJoin}(L,R) = D_{shuffle}(L \cup R) \times R_{columns} \tag{4.32}$$

4.3.4 Early Stop Sort-Merge Point-in-Time Join

The Early Stop Sort-Merge process is executed by first performing a shuffle of both the left and right tables, sorting the data, and then performing a scan of both tables to find matches. Therefore, the cost and complexity are the same as for the native Spark Sort-Merge Join operator and can be observed in Equations (4.33), (4.34), (4.35) and (4.36).

$$O_{ESSMJoin}(L, R) = O_{SMJoin}(L, R) \tag{4.33}$$

$$S_{ESMJoin}(L, R) = S_{SMJoin}(L, R) \tag{4.34}$$

$$P_{ESSMJoin}(L, R) = P_{SMJoin}(L, R) \tag{4.35}$$

$$D_{ESSMJoin}(L, R) = D_{SMJoin}(L, R) \tag{4.36}$$

Chapter 5

Results and Analysis

In this chapter, the results of the experiments will be presented and briefly discussed. The main metrics examined from the test results are the total time elapsed to execute the algorithm, the peak execution memory, the shuffle bytes written, the disk and the memory data spill.

5.1 Major Results

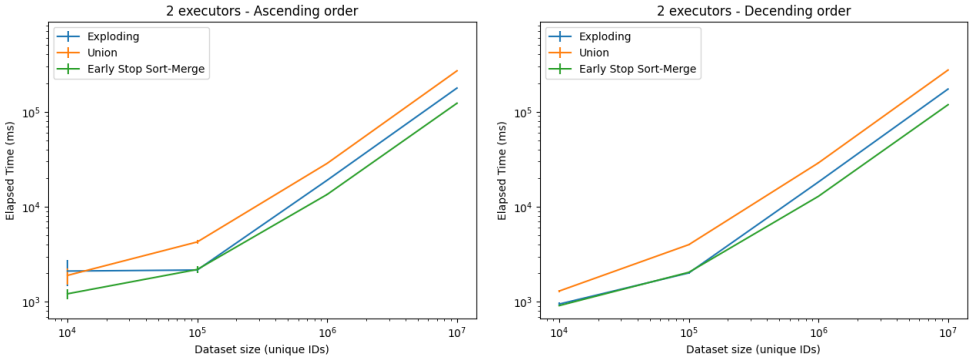
In this section, the main results achieved by the execution of the experiments will be presented, and different subsections will discuss how the algorithms perform when scaling attributes, such as the size of the dataset, the number of buckets, and the number of executors.

5.1.1 Dataset Size

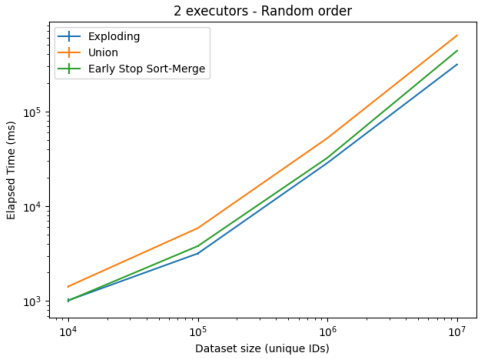
This subsection will focus on how the different algorithms perform when changing the size of the dataset used to perform the PIT join using the implemented algorithms. The results presented are from running the algorithm on a cluster containing two executors and will serve as a baseline to compare how the algorithms perform when using bucketing.

Elapsed Time

When the dataset grew, all algorithms followed a near-linear growth over time, but with different slopes. In Figure 5.1, the growth of the time elapsed as the datasets grow can be observed. The Union algorithm performs worst in all cases, and the Early Stop Sort-Merge algorithm performs best in all cases except for the randomly sorted tables, where the Exploding algorithm



(a) Elapsed time - Ascending order (b) Elapsed time - Descending order



(c) Elapsed time - Random order

Figure 5.1: Elapsed time using two executors

performs best. Whether the dataset was sorted in ascending or descending order did not significantly affect the elapsed time between the experiments; however, randomly sorted data negatively affected all algorithms. The slope and intercept values, together with the respective p-value and standard error, using the linear least-squares regression model found in Equation (5.1), can be observed in Table 5.1. In the table, the Early Stop Sort-Merge is abbreviated with *ESSM*. From these values, it can be observed that the Early Stop Sort-Merge can achieve a speedup of around 2.0 to 2.5 times, compared to the Union algorithm for any dataset, and can achieve a speedup of around 1.45 times compared to the Exploding algorithm for sorted datasets. For the randomly ordered dataset, the Early Stop Sort-Merge Join received a slowdown of about 0.71 times compared to the Exploding algorithm.

$$\hat{Y} = \beta X + \alpha \tag{5.1}$$

Table 5.1: Elapsed time as dataset grows

Dataset	Algorithm	Slope	Intercept	p-value	SE
sort-asc	Exploding	1.757×10^{-2}	1.587×10^{3}	6.286×10^{-5}	1.393×10^{-4}
sort-asc	Union	2.674×10^{-2}	1.791×10^{3}	5.741×10^{-7}	2.026×10^{-5}
sort-asc	ESSM	1.218×10^{-2}	1.143×10^{3}	2.451×10^{-6}	1.906×10^{-5}
sort-desc	Exploding	1.725×10^{-2}	7.025×10^{2}	7.246×10^{-6}	4.643×10^{-5}
sort-desc	Union	2.730×10^{-2}	1.313×10^{3}	2.005×10^{-6}	3.866×10^{-5}
sort-desc	ESSM	1.174×10^{-2}	9.301×10^{2}	3.506×10^{-6}	2.198×10^{-5}
sort-rand	Exploding	3.124×10^{-2}	-5.071×10^{2}	4.235×10^{-5}	2.033×10^{-4}
sort-rand	Union	6.363×10^{-2}	-3.411×10^{3}	1.560×10^{-4}	7.948×10^{-4}
sort-rand	ESSM	4.397×10^{-2}	-3.591×10^{3}	3.299×10^{-4}	7.988×10^{-4}

Memory Consumption

The results achieved by collecting data on memory consumption can be observed in Figure 5.2, showing how the maximum memory usage changes, and in Figure 5.3, showing how many shuffle data bytes are written. Taking into account the maximum memory consumption for relatively small datasets, where the left dataset has fewer than one million unique IDs, the Exploding algorithm uses less memory. This is possibly due to the change in the joining technique chosen by the Spark strategy. Spark will use a Broadcast Hash Join when one dataset is small enough to be broadcasted. For the larger datasets, the memory consumption grew significantly, resulting in a usage of about 81.6 gigabytes for the datasets of the largest datasets; about 1.85 times more than that of the Early Stop Sort-Merge, which peaked at about 44.1 gigabytes of memory usage. Furthermore, for the number of shuffle bytes written, the Exploding algorithm and the Early Stop Sort-Merge used about the same amount as any of the datasets grew, both using less than that of the Union algorithm. Furthermore, the randomized ordered dataset resulted in an increase in shuffle bytes written for all algorithms, whereas the ascending and descending order datasets achieved about the same. Exact data from the experiments using the ascending order sorted datasets can be observed in Table 5.2.

5.1.2 Bucketing

The bucketing was performed on the dataset with the data sorted in ascending order (`sort-asc`); using 20, 40, 80, and 160 buckets to bucket the data.

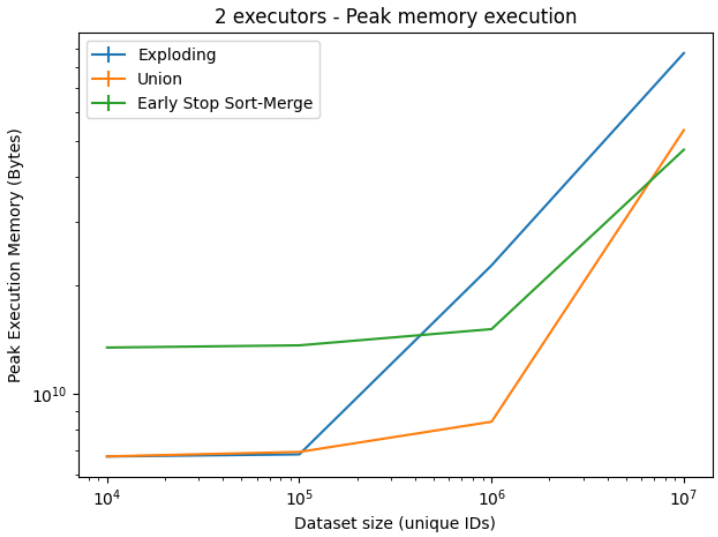


Figure 5.2: Peak memory consumption using two executors

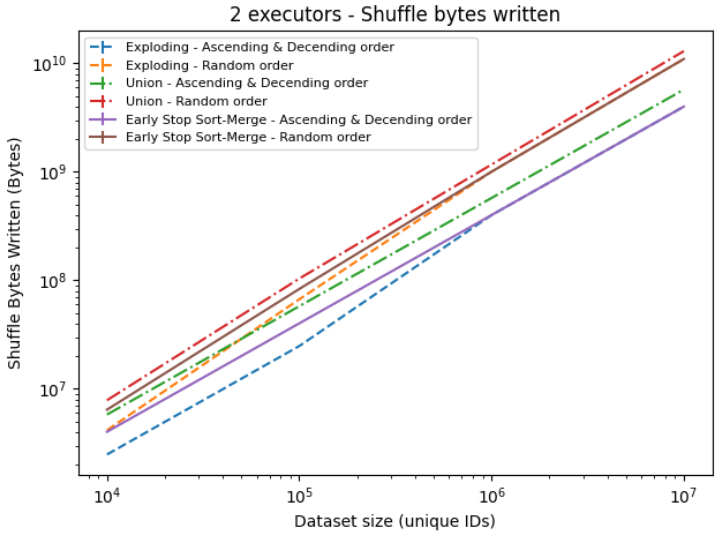


Figure 5.3: Shuffle bytes written using two executors

Table 5.2: Memory consumption two executors - ascending order

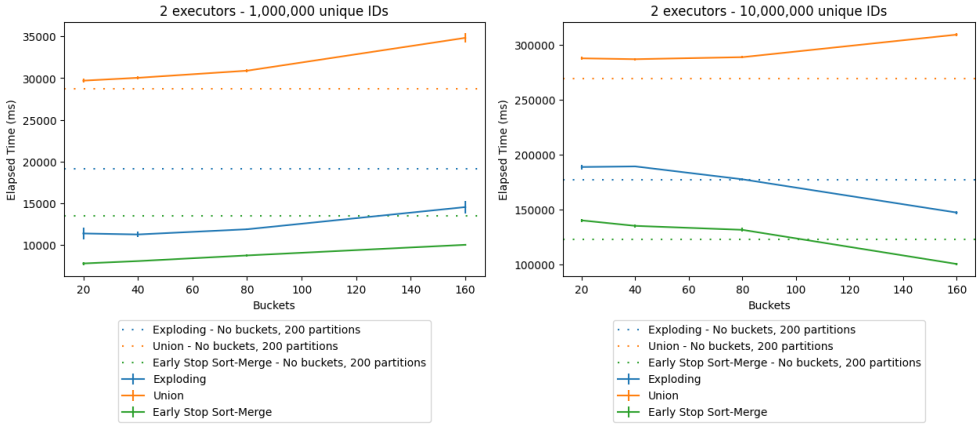
Dataset size	Algorithm	Peak memory consumption (GB)	Shuffle Bytes Written (MB)
10,000	Exploding	6.26	2.31×10^{-3}
10,000	Union	6.26	5.39×10^{-3}
10,000	ESSM	12.52	3.74×10^{-3}
100,000	Exploding	6.35	2.31×10^{-2}
100,000	Union	6.45	5.38×10^{-2}
100,000	ESSM	12.71	3.37×10^{-2}
1,000,000	Exploding	21.14	3.72×10^{-1}
1,000,000	Union	7.81	5.35×10^{-1}
1,000,000	ESSM	14.09	3.72×10^{-1}
10,000,000	Exploding	81.64	3.72
10,000,000	Union	50.00	5.34
10,000,000	ESSM	44.14	3.72

Elapsed Time

The results of the joining operations on the bucketed data of the two largest datasets (10 million and one million unique IDs) can be observed in Figure 5.4; reference lines mark the time elapsed for the respective unbucketed dataset. No matter how many buckets, the Union performed slightly worse and became increasingly slower as the number of buckets grew and never achieved the unbucketed performance. This is possibly due to the Union operation not performing a shuffle before performing the operation, but later performing a shuffle operation on the unionized data. However, for the other algorithms, the performance when using the smaller datasets got worse as the number of buckets grew, but improved for the larger datasets, exceeding the performance of the unbucketed data. Numerous factors can affect this; for example, if the bucket partitions are too large, memory could be spilled and costly disk operations could be introduced.

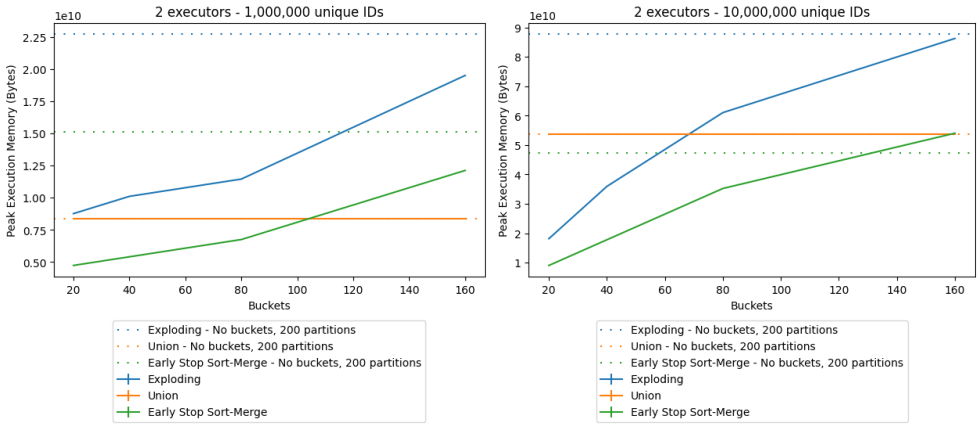
Memory Consumption

The peak execution memory when performing the algorithm on the data of the two largest datasets can be observed in Figure 5.5. Although the Union algorithm has the same peak execution memory, both the Exploding and Early Stop Sort-Merge algorithms increased in execution memory as the number of buckets grew; this could be a result of the increase in parallelism as the number of buckets increased. Furthermore, both the Exploding algorithm and the Early



(a) 1,000,000 unique IDs (b) 10,000,000 unique IDs

Figure 5.4: Elapsed time as number of buckets grow



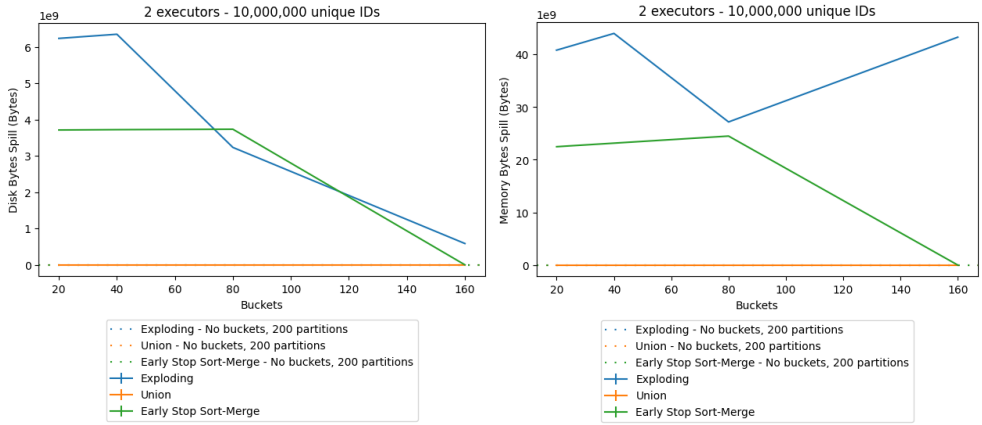
(a) 1,000,000 unique IDs (b) 10,000,000 unique IDs

Figure 5.5: Memory consumption as number of buckets increase

Stop Sort-Merge had no shuffle bytes written or read during the execution, but the Union algorithm had the same number as for the unbucketed dataset.

Spilled Data

Data spillage did not occur until the largest dataset (10 million unique IDs) was used. Data spilling occurred for both the Exploding and Early Stop Sort-Merge algorithms, as shown in Figure 5.6. Spark distinguishes two metrics for data spillage, disk spill, and memory spill; disk spill is the size of the data that gets spilled and written to disk (serialized data), while memory spill is the size



(a) Disk bytes spill (b) Memory bytes spill

Figure 5.6: Spilled data as number of buckets increase

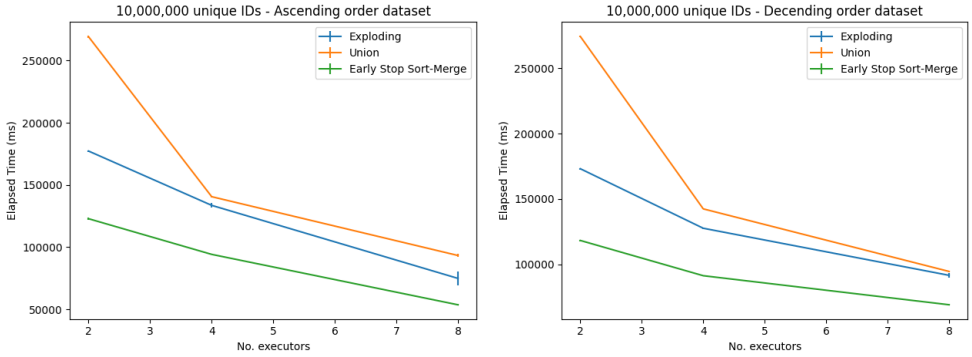
of the data as it exists in memory (deserialized data). However, as the number of buckets increased, the amount of data spilled decreased. The Exploding algorithm disk spill peaked around 5.91 gigabytes when using 40 buckets, and the memory spill peaked around 40.97 gigabytes also when using 40 buckets. Furthermore, the amount of spilled disk data reached a low of 0.546 gigabytes when using 160 buckets, although the memory spill increased in this configuration to 40.31 gigabytes, compared to the 25.31 gigabytes spilled when using 80 buckets. For the Early Stop Sort-Merge algorithm, the disk and memory spill peaked at 3.48 and 22.81 gigabytes, respectively, when using 80 buckets. When the number of buckets increased to 160, no data was spilled using the Early Stop Sort-Merge algorithm.

5.1.3 Increasing Executors

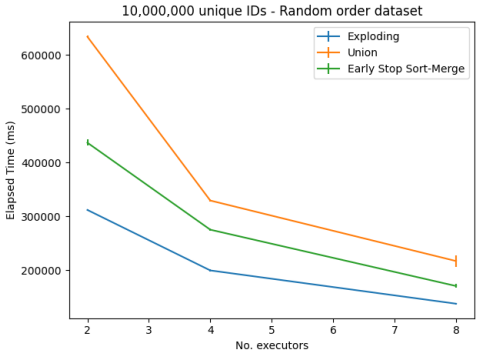
When the number of executors increases, only the elapsed time is affected. The quantity of data to process is the same amount, so the memory consumption of the algorithms remained the same.

Elapsed time

When increasing the number of executors in the cluster, the time elapsed to perform the Spark jobs decreased substantially, for all algorithms, as can be observed in Figure 5.7, where the results are shown for the largest datasets. Applying the speedup formula seen in Equation (5.2), where n is the number of executors, the algorithm that achieved the highest speedup



(a) Elapsed time, ascending order (b) Elapsed time, descending order



(c) Elapsed time, random order

Figure 5.7: Elapsed time as number of executors increase

was the Union algorithm; achieving a speedup of about 2.90 when using eight executors, for all large datasets. Both the Exploding and the Early Stop Sort-Merge algorithms achieved similar speedup and deviated similarly between experiments. With the Exploding algorithm achieving a speedup between 1.89 and 2.36 when utilizing eight executors, and the Early Stop Sort-Merge achieving between 1.71 and 2.56 speedup. The results of the speedup calculations can be found in Table 5.3.

$$S(n) = \frac{elapsedTime(2)}{elapsedTime(n)} \tag{5.2}$$

5.1.4 Comparison with Apache Hive

The results gathered by executing the Exploding algorithm using the Fez execution engine can be observed in Figure 5.8. The query execution of the

Table 5.3: Speedup - 10,000,000 unique IDs

Algorithm	Dataset	Speedup (4 executors)	Speedup (8 executors)
Exploding	sort-asc	1.33	2.37
Exploding	sort-desc	1.36	1.89
Exploding	sort-rand	1.56	2.26
Union	sort-asc	1.91	2.89
Union	sort-desc	1.93	2.90
Union	sort-rand	1.92	2.92
ESSM	sort-asc	1.30	2.29
ESSM	sort-desc	1.30	1.71
ESSM	sort-rand	1.59	2.56

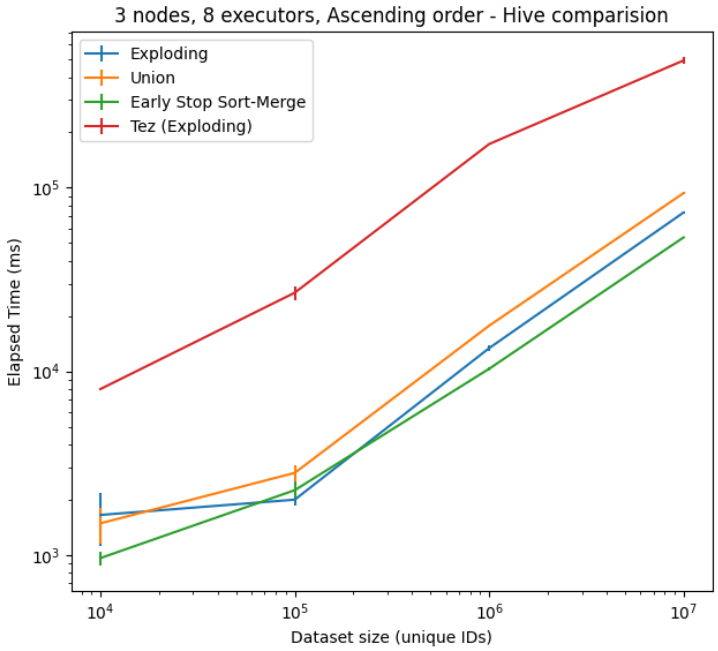


Figure 5.8: Elapsed time comparison, Spark implementations vs. Hive query

Hive query took a lot longer than any of the Spark implementations when running on a cluster of three nodes, with Fez reaching about 482 seconds for execution of the largest dataset, compared to 123 seconds achieved by the Early Stop Sort-Merge algorithm running on Spark.

5.1.5 Raw Data

The raw observed data retrieved by running the experiments is available on Github[1]. The repository also contains Python scripts that were used for the generation of the graphs seen in this report.

5.2 Validity of Results

Since these results only capture the performance of algorithms for specific types of datasets, they cannot be applied to every possible permutation or configuration of the dataset. However, these results provide information on the general performance of the algorithms.

The desired distribution of the results from the experiments would be to have a standard deviation equal to zero, meaning that all the runs result in the exact same observations. Of course, there were deviations in the observed values. As expected, the standard deviation became relatively small compared to the mean value observed as the size of the dataset grew. These deviations could be the result of background tasks or garbage collection. In general, the elapsed time metric received a higher relative standard deviation compared to the other metrics.

In general, the elapsed time standard deviation for the larger datasets remained around 1% of the observed mean; with the datasets using bucketing and more executors being affected more and could deviate with approximately 4% of the observed mean.

5.3 Summary

For the largest dataset (10 million unique IDs), early stop sorting was the best algorithm to efficiently use assigned resources. When choosing the Early Stop Sort-Merge algorithm, the observations show a decrease of 46% of the peak execution memory compared to the Exploding algorithm and a decrease of approximately 12% compared to the Union algorithm. Furthermore, there was a decrease in the elapsed time of approximately 32% compared to the Exploding algorithm for sorted data and 56% compared to the Union algorithm. An exception to this was when the data were sorted in a random order, in this case the Exploding algorithm was proving to be more resilient,

[1] Pettersson, Axel, Spark PIT Data Analysis Available: https://github.com/Ackuq/spark-pit-data-analysis. [Accessed: 2022-05-12]

where the result showed a speedup of approximately 40% when choosing the Exploding algorithm over the Early Stop Sort-Merge; the peak execution memory remained the same for the random ordered dataset.

For the smaller datasets, specifically those with fewer than one million unique IDs, the Early Stop Sort-Merge did not outperform the others by any significant amount. In terms of elapsed time, it performed roughly as well as the Exploding algorithm. In terms of execution memory, the Early Stop Sort-Merge achieved the highest of the three, with an increase of about 100% compared to both algorithms, showing a probable greater amount of overhead.

Furthermore, while the Exploding and Early Stop Sort-Merge showed an improvement when introducing the bucketed datasets, the Union algorithm only performed worse as the number of buckets increased. For the Exploding and Early Stop Sort-Merge, increasing the number of buckets decreased the performance on the smaller datasets, while increasing the performance of the largest dataset. Furthermore, increasing the number of buckets also increased the peak execution memory of the Exploding and Early Stop Sort-Merge algorithm, possibly due to an increase in the parallelism. A decrease in data disk spillage could also be observed in these algorithms as the number of buckets increased.

Chapter 6

Conclusions and Future Work

This chapter presents conclusions and reflections based on the results achieved from this project. As the research question in this work states, the goal is to find out an efficient way to execute a PIT join in terms of time and resources. To answer this question, the results presented in the previous chapter will be reflected upon to present recommendations on how data engineers should go about integrating such a join functionality into their data pipeline.

6.1 Conclusions

By analyzing the results presented in the previous chapters, the following conclusions can be inferred:

- Early Stop Sort-Merge algorithm preferred in terms of memory consumption and elapsed time for very large datasets
- For smaller datasets, the Early Stop Sort-Merge and Exploding algorithms performed equally well in terms of elapsed time, but Exploding performs better in terms of memory consumption due to less overhead
- Number of buckets can increase or decrease the elapsed time, as well as introduce the possibility of disk spillage and increased memory consumption
- Exploding and Early Stop Sort-Merge scales equally well as the number of executors increases

- The Union algorithm receives a large decrease in elapsed time when initially increasing executors, but after a threshold, the decrease slows down
- Having the data stored sorted decreases elapsed time significantly
- Specific sort order of the data is not that significant

Of course, there is no silver bullet that can answer this question in general terms; however, based on these conclusions, the most beneficial algorithm for the observed use cases is the Early Stop Sort-Merge if control over the sorting of the data is possible. However, due to the memory overhead, it could perform worse if the amount of memory available to the executors is small and the datasets used are small. Furthermore, both the number of buckets and the scaling of the number of executors are recommended to be able to handle very large amounts of data. However, as the results show, one has to be careful when determining the number of buckets used if such an optimization is chosen; if too few buckets are used, there is a risk of data spillage and a lower degree of parallelism, whereas too many may be inefficient for lower volumes of data. Unfortunately, there is no universal formula to determine the most optimized number of buckets to use, so the best way to achieve this is by empirical comparisons of datasets.

Considering the elapsed time by executing a PIT join in Apache Hive using Tez as an execution engine compared to Spark, one can see that the decrease in the elapsed time is very significant. Nevertheless, there could be other considerations for choosing executing the queries with Tez over Spark, for example if the data is way too large to be able to fit into memory.

Limitations

Due to the time constraints of this project, the number of experiments had to be limited. Possible interesting scenarios that were not tested were cases where the left dataset contains more entries than the right dataset. Introducing datasets that use controlled distributions of data instead of replicating a more general use case could gather information on how the algorithms perform under specific circumstances.

6.2 Future Work

This section aims to present considerations for future work within the area that this project covered.

6.2.1 Partitioning of Datasets Without Primary Key

The algorithms developed throughout this project focused mainly on datasets where each row contains a primary key and an event timestamp. Specifically, in each of the algorithms, the primary key was used for the partitioning of the data. For datasets where there is no primary key, the datasets would be contained in a single partition, significantly reducing the parallelism of the algorithms. To remedy an increase in the degree of parallelism when processing these datasets, an additional technique is required to partition them. A potential solution to this is seen in the Two Sigma Flint project, as described in Section 2.4.1. In Flint, both the left and right datasets are partitioned using date ranges, which are then merged when performing then join, with the partitioning interval of the right table having an padding of the desired tolerance (maximum time gap for valid pair of rows), for example, one day. However, since this solution requires the use of tolerance, it might not be applicable for all use cases. Furthermore, the Flint solution uses data sampling to detect partition intervals; introducing stored data about the timestamp distribution of the data could also reduce the amount of sampling performed when calculating these partition intervals, improving performance.

A possible solution to be able to perform partitioning without the need of a primary key or tolerance could be to partition the left and right datasets based on a sample of closed-open intervals extracted from the right dataset. Here, each interval is created between two event times that is known to exist in the right dataset. For example, if the right dataset has rows with event times one, three, five and sevent, possible intervals would be $[1, 3)$, $[3, 5)$, $[5, 7)$, and $[7, \infty)$. Considering that a left row must be matched with a row whose timestamp is less than or equal to its own timestamp, a left row partitioned using the intervals extracted from the right dataset, has a match that must exist within that partition, if a match exists. However, this solution would still require sampling of the right dataset and potentially could be ineffective for skewed datasets.

6.2.2 Formula for Estimating Number of Bucket

As the results of this project have shown, bucketing of data can greatly improve the execution time of the algorithms. However, there is no universal formula to estimate the number of buckets used. There are many factors to take into account when estimating this, for example, the size of the datasets, the number of executors in the Spark applications, and the skewness of the data. This project has shown that the number of buckets can either worsen or improve the performance of algorithms as the number of buckets increases, depending on the size of the data. A more thorough investigation of how to estimate the optimized number of buckets for a dataset, considering the aforementioned attributes, could allow data engineers to swiftly speed up their data pipelines.

References

[1] A. Kumar, *Practical Full Stack Machine Learning: A Guide to Build Reliable, Reusable, and Production-Ready Full Stack ML Solutions*. BPB Publications, Nov. 2021. ISBN 978-93-91030-42-1 [Page 1.]

[2] S. Molin and K. Jee, *Hands-On Data Analysis with Pandas: A Python Data Science Handbook for Data Collection, Wrangling, Analysis, and Visualization, 2nd Edition*. Birmingham: Packt Publishing, Limited, 2021. ISBN 978-1-80056-345-2 [Page 2.]

[3] G. Aris Galiotos, B. Paul A Bilokon, N. Jan Novotny, and D. Frederic Deleze, *Machine Learning and Big Data with kdb+/q*, ser. Wiley finance. Wiley, 2019. ISBN 978-1-119-40475-0 [Page 2.]

[4] The Apache Software Foundation, "Apache Spark™ - Unified Engine for large-scale data analytics." [Online]. Available: https://spark.apache.org/ [Accessed: 2022-01-25] [Page 2.]

[5] P. Bock, *Getting It Right: R&d Methods for Science and Engineering*. San Diego: Elsevier Science & Technology, 2020. ISBN 978-0-12-816165-4 Book Title: Getting It Right. [Page 3.]

[6] X. Lian and T. Zhang, "The Optimization of Cost-Model for Join Operator on Spark SQL Platform," *MATEC Web of Conferences*, vol. 173, p. 01015, 2018. doi: 10.1051/matecconf/201817301015. [Online]. Available: https://www.matec-conferences.org/10.1051/matecconf/201817301015 [Accessed: 2022-02-07] [Pages 3, 24, 36, 37, and 38.]

[7] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. Sebastopol: O'Reilly Media, Incorporated, 2017. ISBN 978-1-4919-4320-5 [Page 3.]

[8] Chris Fehily, *SQL: Visual QuickStart Guide*. Peachpit Press, 2008. ISBN 978-0-321-58406-9 [Page 5.]

[9] Microsoft Corporation, "Joins (SQL Server)," Sep. 2021. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins [Accessed: 2022-02-18] [Page 6.]

[10] G. Regunath, "A Beginner's Guide To Understanding Feature Stores," Jan. 2022. [Online]. Available: https://www.advancinganalytics.co.uk/blog/2022/1/13/a-beginners-guide-to-understanding-feature-stores [Accessed: 2022-03-24] [Page 8.]

[11] M. Zaharia and B. Chambers, *Spark: The Definintive Guide*. O'Reilly, 2018. ISBN 978-1-4919-1221-8 [Pages 8, 9, 10, 11, 12, 13, and 15.]

[12] L. Mearian, "Data storage goes from $1M to 2 cents per gigabyte," Mar. 2017. [Online]. Available: https://www.computerworld.com/article/3182207/cw50-data-storage-goes-from-1m-to-2-cents-per-gigabyte.html [Accessed: 2022-02-10] [Page 9.]

[13] The Apache Software Foundation, "Cluster Mode Overview - Spark 3.2.1 Documentation." [Online]. Available: https://spark.apache.org/docs/3.2.1/cluster-overview.html [Accessed: 2022-02-10] [Pages 9 and 27.]

[14] ——, "RDD Programming Guide - Spark 3.2.1 Documentation." [Online]. Available: https://spark.apache.org/docs/3.2.1/rdd-programming-guide.html [Accessed: 2022-02-10] [Page 11.]

[15] ——, "Spark SQL and DataFrames - Spark 3.2.1 Documentation." [Online]. Available: https://spark.apache.org/docs/3.2.1/sql-programming-guide.html [Accessed: 2022-02-14] [Page 11.]

[16] D. Du, *Apache Hive Essentials: Essential Techniques to Help You Process, and Get Unique Insights from, Big Data*, 2nd ed. Birmingham: Packt Publishing, Limited, Jun. 2018. ISBN 978-1-78899-509-2 [Page 12.]

[17] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne Victoria Australia: ACM, May 2015. doi: 10.1145/2723372.2742790. ISBN 978-1-4503-2758-9 pp. 1357–1369. [Online]. Available: https://dl.acm.org/doi/10.1145/2723372.2742790 [Accessed: 2022-04-05] [Page 12.]

[18] The Apache Software Foundation, “Hive Tables - Spark 3.2.1 Documentation.” [Online]. Available: https://spark.apache.org/docs/3.2.1/sql-data-sources-hive-tables.html [Accessed: 2022-02-14] [Page 12.]

[19] J. Laskowski, “Bucketing - The Internals of Spark SQL.” [Online]. Available: https://books.japila.pl/spark-sql-internals/bucketing/?h=bucketing [Accessed: 2022-02-14] [Page 12.]

[20] B. Konieczny, “Jobs, stages and tasks,” Apr. 2017. [Online]. Available: https://www.waitingforcode.com/apache-spark/jobs-stages-tasks/read [Accessed: 2022-02-14] [Page 13.]

[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, “Spark SQL: Relational Data Processing in Spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne Victoria Australia: ACM, May 2015. doi: 10.1145/2723372.2742797. ISBN 978-1-4503-2758-9 pp. 1383–1394. [Online]. Available: https://dl.acm.org/doi/10.1145/2723372.2742797 [Accessed: 2022-02-07] [Page 14.]

[22] L. Leturgez, “Spark’s Logical and Physical plans … When, Why, How and Beyond.” Oct. 2021. [Online]. Available: https://medium.com/datalex/sparks-logical-and-physical-plans-when-why-how-and-beyond-8cd1947b605a [Accessed: 2022-02-16] [Page 14.]

[23] J. Laskowski, “Whole-Stage CodeGen - The Internals of Spark SQL.” [Online]. Available: https://books.japila.pl/spark-sql-internals/whole-stage-code-generation/?h=whole+stage+code+generation [Accessed: 2022-02-16] [Page 14.]

[24] ——, “Actions - The Internals of Spark SQL.” [Online]. Available: https://books.japila.pl/spark-sql-internals/spark-sql-Dataset-actions/ [Accessed: 2022-02-16] [Page 14.]

[25] The Apache Software Foundation, “JOIN - Spark 3.2.1 Documentation.” [Online]. Available: https://spark.apache.org/docs/3.2.1/sql-ref-syntax-qry-select-join.html [Accessed: 2022-02-16] [Page 15.]

[26] ——, “Hints - Spark 3.2.1 Documentation.” [Online]. Available: https://spark.apache.org/docs/3.2.1/sql-ref-syntax-qry-select-hints.html [Accessed: 2022-02-16] [Page 16.]

[27] ——, "SparkStrategies.scala," Feb. 2022, original-date: 2014-02-25T08:00:08Z. [Online]. Available: https://github.com/apache/spark/blob/5976a0655e78b1d5666d8b2b5d459eafc6319547/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala [Accessed: 2022-02-16] [Page 16.]

[28] Databricks Inc., "asofJoin.scala," Feb. 2022, original-date: 2020-07-14T15:43:11Z. [Online]. Available: https://github.com/databrickslabs/tempo/blob/e05e45545af3e53dfd8caf548697b5ccf281b2dc/scala/tempo/src/main/scala/com/databrickslabs/tempo/asofJoin.scala [Accessed: 2022-02-18] [Page 18.]

[29] Two Sigma LCC, "flint/LeftJoin.scala," Feb. 2022, original-date: 2016-10-19T17:44:15Z. [Online]. Available: https://github.com/twosigma/flint/blob/e4d0abae03e619cc6f52fb212d33977bf5aceb52/src/main/scala/com/twosigma/flint/rdd/function/join/LeftJoin.scala [Accessed: 2022-02-21] [Page 18.]

[30] The Pandas Development Team, "pandas/join.pyx," Feb. 2022, original-date: 2010-08-24T01:37:33Z. [Online]. Available: https://github.com/pandas-dev/pandas/blob/226f1c2b8ad6fafd3c268459f3f745191591bb89/pandas/_libs/join.pyx [Accessed: 2022-02-21] [Page 19.]

[31] Julia Computing, Inc., "IndexedTables.jl/join.jl," Feb. 2022, original-date: 2016-03-10T23:09:06Z. [Online]. Available: https://github.com/JuliaData/IndexedTables.jl/blob/2e97b488e24b9069cca5449b6e95dc9690ab19a1/src/join.jl [Accessed: 2022-02-21] [Page 19.]

[32] The Apache Software Foundation, "Integration with Cloud Infrastructures - Spark 3.2.1 Documentation." [Online]. Available: https://spark.apache.org/docs/3.2.1/cloud-integration.html [Accessed: 2022-02-24] [Page 27.]

[33] ——, "Monitoring and Instrumentation - Spark 3.2.1 Documentation." [Online]. Available: https://spark.apache.org/docs/3.2.1/monitoring.html [Accessed: 2022-02-23] [Page 28.]

[34] L. Canali, "sparkMeasure," Feb. 2022, original-date: 2017-03-16T20:53:26Z. [Online]. Available: https://github.com/LucaCanali/sparkMeasure [Accessed: 2022-02-23] [Page 29.]

[35] The Apache Software Foundation, “ExperimentalMethods (Spark 3.2.1 JavaDoc).” [Online]. Available: https://spark.apache.org/docs/3.2.1/api/java/org/apache/spark/sql/ExperimentalMethods.html [Accessed: 2022-03-01] [Page 35.]

[36] S. Buss and A. Knop, “Strategies for Stable Merge Sorting,” in *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, ser. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 1272–1290. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975482.78 [Accessed: 2022-03-08] [Page 39.]

Appendix A

Scala code

A.1 Exploding Point-in-Time Scala implementation

The implementation for the exploding PIT join, as described in Section 4.1.1, can be observed in Listing A.1. This implementation assumes the use of DataFrames.

A.2 Union Point-in-Time Scala implementation

The implementation for the union PIT join, as described in Section 4.1.2 can be observed, in Listing A.2. This implementation assumes the use of DataFrames.

Listing A.1: Scala implementation exploding Point-in-Time

```
def join(
    left: DataFrame,
    right: DataFrame,
    leftTSColumn: Column,
    rightTSColumn: Column,
    partitionCols: Seq[(Column, Column)] = Seq()
): DataFrame = {
  // Create the equality conditions of the partitioning column
  val partitionConditions =
    partitionCols.map(col => col._1 === col._2)
  // Combine the partitioning conditions with the PIT condition
  val joinConditions =
    partitionConditions :+ (leftTSColumn >= rightTSColumn)
  // Reduce the sequence of conditions to a single one
  val joinCondition =
    joinConditions.reduce((current, previous) =>
      current.and(previous))
  // Join on conditions that left.ts >= right.ts and belongs to same partition
  val combined = left.join(
    right,
    joinCondition
  )
  // Partition each window using the partitioning columns of the left DataFrame
  val windowPartitionCols = partitionCols.map(_._1) :+ leftTSColumn

  // Create the Window specification
  val windowSpec =
    Window
      .partitionBy(windowPartitionCols: _*)
      .orderBy(rightTSColumn.desc)

  combined
    // Take only the row with the highest timestamps within each window frame
    .withColumn("rn", row_number().over(windowSpec))
    .where(col("rn") === 1)
    .drop("rn")
}
```

Listing A.2: Scala implementation union Point-in-Time join

```
def join(left: DataFrame, right: DataFrame, leftTSColumn:
    String = "ts", rightTSColumn: String = "ts",
    leftPrefix: Option[String] = None, rightPrefix:
    String, partitionCols: Seq[String] = Seq()
): DataFrame = {
  // Rename the columns in left and right dataframes
  val leftPrefixed = leftPrefix match {
    case Some(lp) => prefixDF(left, lp, partitionCols)
    case None     => left
  }
  val rightPrefixed = prefixDF(right, rightPrefix,
      partitionCols)
  // Timestamp columns
  val leftTS = leftPrefix match {
    case Some(p) => p ++ leftTSColumn
    case None    => leftTSColumn
  }
  val rightTS = rightPrefix ++ rightTSColumn
  val leftPrefixedAllColumns = addColumns(
    leftPrefixed.withColumn(DF_INDEX_COLUMN, lit(1)),
    rightPrefixed.columns.filter(!partitionCols.contains(_)))
  val rightPrefixedAllColumns = addColumns(
      rightPrefixed.withColumn(DF_INDEX_COLUMN, lit(0)),
      leftPrefixed.columns.filter(!partitionCols.contains(_))
  )
  val combined = leftPrefixedAllColumns
    .unionByName(rightPrefixedAllColumns)
  val combinedTS =
     combined.withColumn(COMBINED_TS_COLUMN,
     coalesce(combined(leftTS), combined(rightTS)))

  val windowSpec = Window
    .orderBy(COMBINED_TS_COLUMN, DF_INDEX_COLUMN)
    .partitionBy(partitionCols.map(col): _*)
    .rowsBetween(Window.unboundedPreceding,
       Window.currentRow)

  asOfDF = rightPrefixed.columns
      .foldLeft(combinedTS)((df, col) =>
        df.withColumn(col, last(df(col), ignoreNulls =
           true).over(windowSpec))
      // Invalid candidates are those where the left
         values does not existing
      ).filter(col(leftTS).isNotNull)
      .drop(DF_INDEX_COLUMN).drop(COMBINED_TS_COLUMN)
```

Appendix B

Execution Plans

B.1 Exploding Point-in-Time Join Execution Plan

Since the Point-in-Time execution can be done using different joining techniques, the execution plan may differ. Specifically, execution may or may not include a shuffle operation, which greatly affects execution complexity. In the best case, where one dataset is small enough to be broadcasted, the execution plan takes the form seen in Figure B.1. However, if both datasets are very large, they need to be shuffled, resulting in the execution plan, which can be observed in Figure B.2.

B.1.1 Hive Exploding Point-in-Time Execution plan

The execution plan for the HiveQL version of the exploding algorithm can be observed in Listing B.1. As with the Spark implementation, the specific join operator for the initial inner join can vary and affect performance.

B.2 Union Point-in-Time Join Execution Plan

When executed, the union PIT operation makes use of repeated windowing operations to populate the columns from the right table in the resulting table. If the unionized dataset is large, this results in a shuffle operation performed for each window operation. The whole execution plan in this case can be observed in Figure B.3

B.3 Early Stop Sort-Merge Execution Plan

The early stop sort-merge PIT join execution plan is very similar to the sort-merge join execution plan. A sorting operation is performed on each of the datasets and then is merged using the implemented algorithm. If the datasets are large, a shuffle operation will also be executed before sorting them. The entire execution plan can be observed in Figure B.4.

Figure B.1: Best possible execution plan for exploding Point-in-Time join

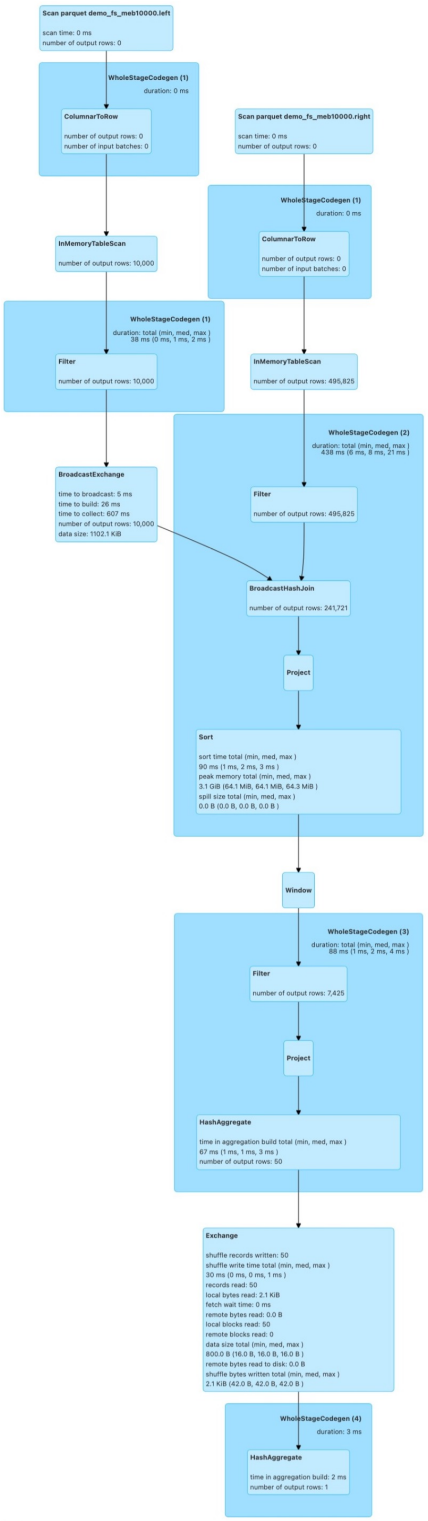


Figure B.2: Worst possible execution plan for exploding Point-in-Time join

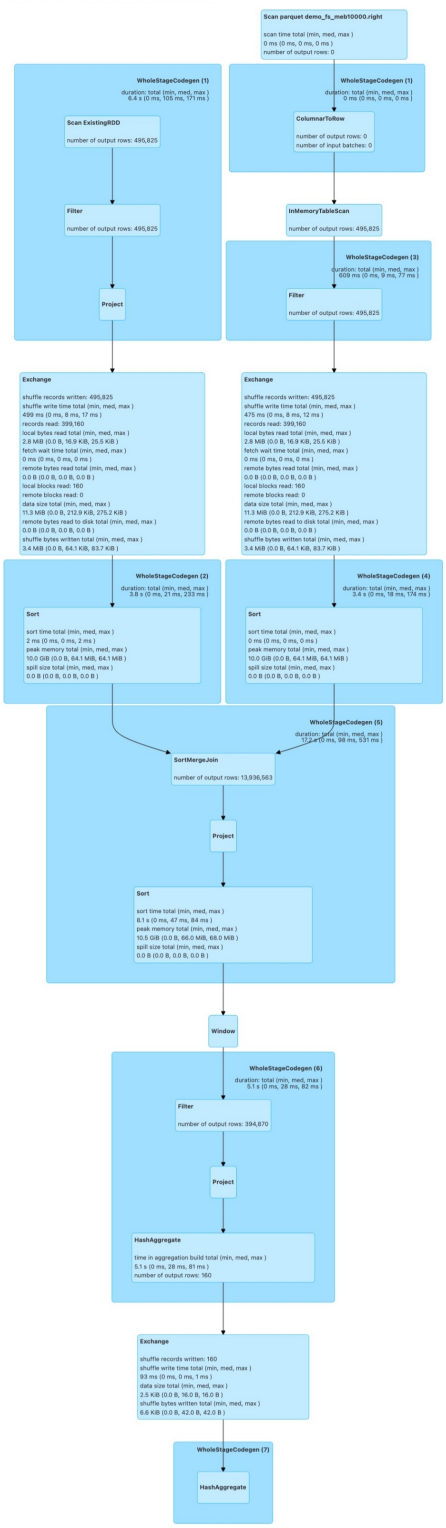


Figure B.3: Execution plan for union Point-in-Time join

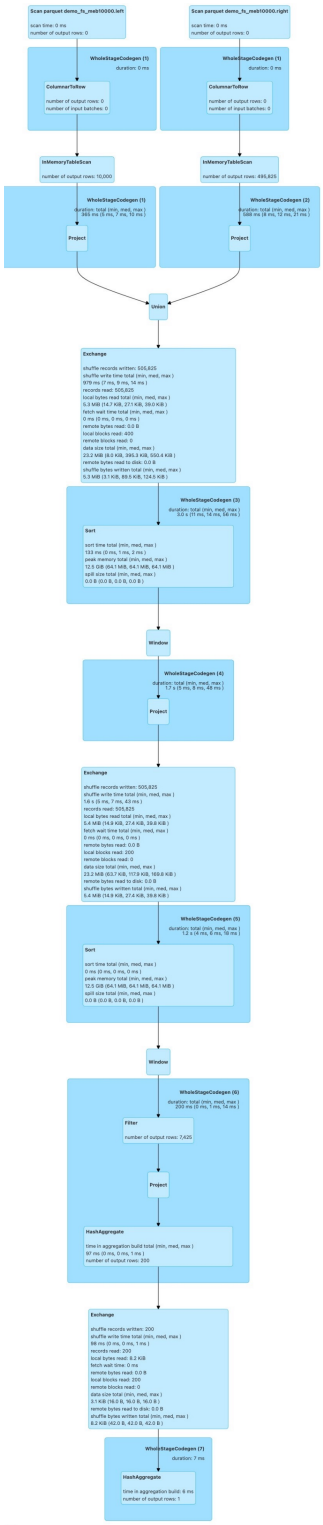
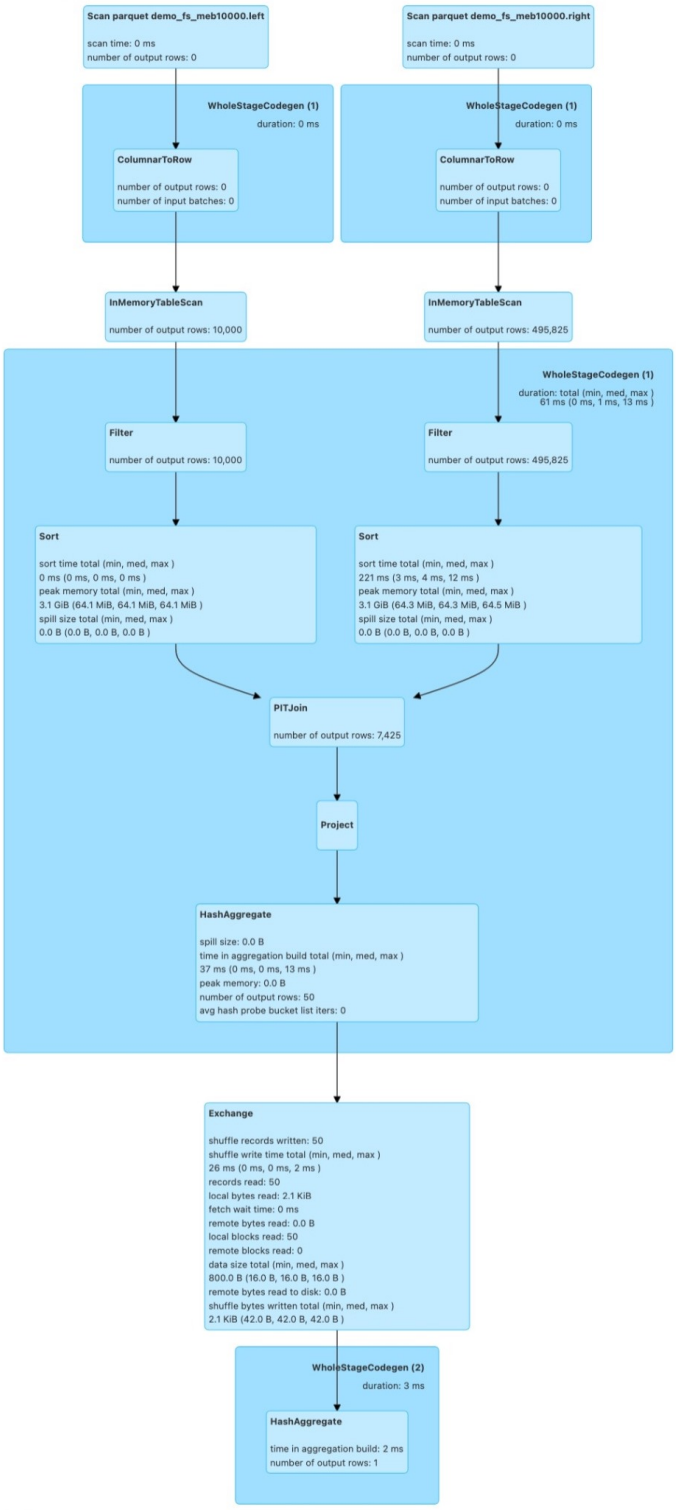


Figure B.4: Execution plan for Early Stop Sort Merge Point-in-Time join



Listing B.1: HiveQL execution plan for a Point-in-Time join

```
Plan optimized by CBO.

Vertex dependency in root stage
Reducer 2 <- Map 1 (SIMPLE_EDGE), Map 4 (SIMPLE_EDGE)
Reducer 3 <- Reducer 2 (SIMPLE_EDGE)

Stage-0
    Fetch Operator
    limit: -1
    Stage-1
      Reducer 3
      File Output Operator [FS_17]
        Select Operator [SEL_13] (rows=1481587 width=292)
          Output: ["_col0", "_col1", "_col2", "_col3", "_col4"]
          Filter Operator [FIL_20] (rows=1481587 width=292)
            predicate: (ROW_NUMBER_window_0 = 1)
            PTF Operator [PTF_12] (rows=2963174 width=292)
              Function definitions: [{}, {"name:":
                  "windowingtablefunction", "order by:": "_col4 ASC
                  NULLS FIRST", "partition by:": "_col0, _col1"}]
              Select Operator [SEL_11] (rows=2963174 width=292)
                Output: ["_col0", "_col1", "_col2", "_col4", "_col5"]
              <-Reducer 2 [SIMPLE_EDGE]
                SHUFFLE [RS_10]
                  PartitionCols: _col0, _col1
                  Filter Operator [FIL_9] (rows=2963174 width=292)
                    predicate: (_col1 >= _col4)
                    Merge Join Operator [MERGEJOIN_33] (rows=8889524
                        width=292)
                      Conds: RS_6._col0 = RS_7._col0(Inner), Output:
                          ["_col0", "_col1", "_col2", "_col4", " _col5"]
                    <-Map 1 [SIMPLE_EDGE]
                      SHUFFLE [RS_6]
                        PartitionCols: _col0
                        Select Operator [SEL_2] (rows=425950 width=292)
                          Output: ["_col0", "_col1", "_col2"]
                          Filter Operator [FIL_21] (rows=425950
                              width=292)
                            predicate: id is not null
                            TableScan [TS_0] (rows=425950 width=292)
                              axel_experiments@sorted_asc_10000000_1_year_left,
                                  sorted_asc_10000000_1_year_left, Tbl:
                                  COMPLETE, Col: NONE, Output: ["id",
                                  "ts", "label"]
                    <-Map 4 [SIMPLE_EDGE]
                      SHUFFLE [RS_7]
                        PartitionCols: _col0
                        Select Operator [SEL_5] (rows=8081386 width=292)
                          Output: ["_col0", "_col1", "_col2"]
                          Filter Operator [FIL_22] (rows=8081386
                              width=292)
                            predicate: id is not null
                            TableScan [TS_3] (rows=8081386 width=292)
                              axel_experiments@sorted_asc_10000000_1_year_right,
                                  sorted_asc_10000000_1_year_right, Tbl:
                                  COMPLETE, Col: NONE, Output: ["id",
                                  "ts", "value"]
```

TRITA-EECS-EX- 2022:00

For DIVA

{
"Author1": { "Last name": "Pettersson",
"First name": "Karl Axel",
"E-mail": "axp@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA240X",
"Credits": "30.0",
"Degree1": {"Educational program": "Master's Programme, Software Engineering of Distributed Systems, 120 credits"
,"programcode": "TSEDM"
,"Degree": "Masters degree"
,"subjectArea": "Software Engineering for Distributed Systems"
},
"Title": {
"Main title": "Resource-efficient and fast Point-in-Time joins for Apache Spark",
"Subtitle": "Optimization of time travel operations for the creation of machine learning training datasets",
"Language": "eng" },
"Alternative title": {
"Main title": "Resurseffektiva och snabba Point-in-Time joins i Apache Spark",
"Subtitle": "Optimering av tidsresningsoperationer för skapande av träningsdata för maskininlärningsmodeller",
"Language": "swe"
},
"Supervisor1": { "Last name": "Sheikholeslami",
"First name": "Sina",
"Local User Id": "0000000172364637",
"E-mail": "sinash@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Division of Software and Computer Systems" }
},
"Supervisor2": { "Last name": "Meister",
"First name": "Moritz",
"E-mail": "moritz@hopsworks.ai",
"Other organisation": "Hopsworks AB"
},
"Examiner1": { "Last name": "Payberah",
"First name": "Amir H.",
"Local User Id": "0000000227488929",
"E-mail": "payberah@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Division of Software and Computer Systems" }
},
"Cooperation": { "Partner_name": "Hopsworks AB"},
"National Subject Categories": "10201, 10206",
"Other information": {"Year": "2022", "Number of pages": "xix,76"},
"Series": { "Title of series": "TRITA-EECS-EX" , "No. in series": "2022:00" },
"Opponents": { "Name": "Ralfs Zangis"},
"Presentation": { "Date": "2022-05-25 11:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/2884945301"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€

A scenario in which modern machine learning models are trained is to make use of past data to be able to make predictions about the future. When working with multiple structured and time-labeled datasets, it has become a more common practice to make use of a join operator called the Point-in-Time join, or PIT join, to construct these datasets. The PIT join matches entries from the left dataset with entries of the right dataset where the matched entry is the row whose recorded event time is the closest to the left row's timestamp, out of all the right entries whose event time occurred before or at the same time of the left event time. This feature has long only been a part of time series data processing tools but has recently received a new wave of attention due to the rise of the popularity of feature stores. To be able to perform such an operation when dealing with a large amount of data, data engineers commonly turn to large-scale data processing tools, such as Apache Spark. However, Spark does not have a native implementation when performing these joins and there has not been a clear consensus by the community on how this should be achieved. This, along with previous implementations of the PIT join, raises the question: "How to perform fast and resource efficient Point-in-Time joins in Apache Spark?". To answer this question, three different algorithms have been developed and compared for performing a PIT join in Spark in terms of resource consumption and execution time. These algorithms were benchmarked using generated datasets using varying physical partitions and sorting structures. Furthermore, the scalability of the algorithms was tested by running the algorithms on Apache Spark clusters of varying sizes. The results received from the benchmarks showed that the best measurements were achieved by performing the join using \textit{Early Stop Sort-Merge Join}, a modified version of the regular Sort-Merge Join native to Spark. The best performing datasets were the datasets that were sorted by timestamp and primary key, ascending or descending, using a

suitable number of physical partitions. Using this new information gathered by this project, data engineers have been provided with general guidelines to optimize their data processing pipelines to be able to perform more resource-efficient and faster PIT joins.

€€€€,
"Keywords[eng]": €€€€
Apache Spark, Point-in-Time, ASOF, Join, Optimizations, Time travel €€€€,
"Abstract[swe]": €€€€

Ett vanligt scenario för maskininlärning är att träna modeller på tidigare observerad data för att för att ge förutsägelser om framtiden. När man jobbar med ett flertal strukturerade och tidsmärkta dataset har det blivit vanligare att använda sig av en join-operator som kallas Point-in-Time join, eller PIT join, för att konstruera dessa datauppsättningar. En PIT join matchar rader från det vänstra datasetet med rader i det högra datasetet där den matchade raden är den raden vars registrerade händelsetid är närmaste den vänstra raden händelsetid, av alla rader i det högra datasetet vars händelsetid inträffade före eller samtidigt som den vänstra händelsetiden. Denna funktionalitet har länge bara varit en del av datahanteringsverktyg för tidsbaserad data, men har nyligen fått en ökat popularitet på grund av det ökande intresset för feature stores. För att kunna utföra en sådan operation vid hantering av stora mängder data vänder sig data engineers vanligvis till storskaliga databehandlingsverktyg, såsom Apache Spark. Spark har dock ingen inbyggd implementation för denna join-operation, och det finns inte ett tydligt konsensus från Spark-rörelsen om hur det ska uppnås. Detta, tillsammans med de tidigare implementationerna av PIT joins, väcker frågan: ”Vad är det mest effektiva sättet att utföra en PIT join i Apache Spark?”. För att svara på denna fråga har tre olika algoritmer utvecklats och jämförts med hänsyn till resursförbrukning och exekveringstid. För att jämföra algoritmerna, exekverades de på genererade datauppsättningar med olika fysiska partitioner och sorteringstrukturer. Dessutom testades skalbarheten av algoritmerna genom att köra de på Spark-kluster av varierande storlek. Resultaten visade att de bästa mätvärdena uppnåddes genom att utföra operationen med algoritmen \textit{early stop sort-merge join}, en modifierad version av den vanliga sort-merge join som är inbyggd i Spark, med en datauppsättning som är sorterad på tidsstämpel och primärnyckel, antingen stigande eller fallande. Fysisk partitionering av data kunde även ge bättre resultat, men det optimala antal fysiska partitioner kan variera beroende på datan i sig. Med hjälp av denna nya information som samlats in av detta projekt har data engineers försetts med allmänna riktlinjer för att optimera sina databehandlings-pipelines för att kunna utföra mer resurseffektiva och snabbare PIT joins.

€€€€,
"Keywords[swe]": €€€€
Apache Spark, Point-in-Time, ASOF, Join, Optimeringar, Tidsresning €€€€,
}