COST-EFFECTIVE BATCH-BASED MIGRATION STRATEGIES FOR NEWSQL-BASED BIG DATA SYSTEMS

NAVEEN KUMAR VADLAMUDI
Bachelor of Technology, Geethanjali College of Engineering and Technology, 2020

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

COST-EFFECTIVE BATCH-BASED MIGRATION STRATEGIES FOR NEWSQL-BASED BIG DATA SYSTEMS

NAVEEN KUMAR VADLAMUDI

Date of Defence: August 12, 2024

Dr. Wendy Osborn Thesis Supervisor	Associate Professor	Ph.D.
Dr. Chali Yllias Thesis Examination Committee Member	Professor	Ph.D.
Dr. John Zhang Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Andrew Fiori Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

This is dedicated to my most beautiful mom (Mrs.Indira Vadlamudi), my father (Mr.Satyanarayana Vadlamudi), and my elder brother (Mr.Srikanth Vadlamudi) for their invaluable sacrifices. Without them, I would not have accomplished what I have today.

Abstract

Modern, high-performance applications demand scalable and efficient databases, leading to the evolution of NewSQL systems. The challenge lies in migrating data from Shardingsphere with PostgreSQL to AWS (Amazon Web Services) cloud object storage. Implementing batch migration algorithms in Apache Spark, specifically targeting Delta Lake format, introduces complexities to ensure seamless data integration and storage within AWS environments.

This thesis explores tailored batch-based migration algorithms for transferring data from Shardingsphere with PostgreSQL to AWS cloud object storage, emphasizing performance optimization by transferring the data faster. The study evaluates various batch loading techniques in Apache Spark, including sequential and concurrent strategies for shard-by-shard and aggregated-shards based algorithms. These techniques aim to maximize efficiency in storing data in Delta Lake format within AWS cloud storage, facilitating effective data management, visualization, and utilization for modern applications, business intelligence, AI and ML. Leveraging the Lakehouse architecture for integrated data processing and analytics.

Acknowledgments

I would love to sincerely thank Dr.Wendy Osborn for her continuous support and suggestions all along my master degree. She is the most supportive and friendly supervisor that one can ever have. I enjoyed my time working with Dr.Wendy in conducting my research in the area of data engineering.

Moreover, I would like to specially thank Dr.Sidney Shapiro, Dr.John Anvik and Dr.Jackie Rice and other members of JJWS research lab, for listening to all my research presentations and providing constructive feedback to me whenever needed. However, I also enjoyed being a lab coordinator organizing the research presentations, and I thank Dr.Wendy and other professors for allowing me to take the responsibility for doing that. Additionally, I would mainly like to thank my house mates, Chandra Suryadevara, Yash Dixit, Angad Pal Singh and Gagan Suryadevara for taking such a good care of me all along of my masters journey. Without you guys, it would have been very difficult for me to do whatever i have done until now. Moreover, I would like to thank Arnab Bose (PhD Math Student) and his wife Julia Ji for inviting me to dinner on multiple occasions. Also, I would like to thank my fellow graduate students Vijay Adoni, Wenzhao Zhu, Shraddha, Samin, Morteza, and Prasanta Bhattacharjee for all the movie nights and get togethers that we've had. Finally, I would also like to thank my other friends Disha Patel, Deep Patel, Sarah Alibhai, Fariha Haroon, Narasimha Reddy (PhD Physics Student), Vatsal Mandaliya for being a support system and reminding me of taking breaks, playing badminton, and hanging out on wings night.

Contents

Dedication iii

Abstract iv

Acknowledgments v

List of Tables ix

List of Figures x

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Motivation . . . 2
- 1.3 Contributions . . . 3
- 1.4 Organization . . . 3

2 Background Study and Related Work 5
- 2.1 Background Study . . . 5
  - 2.1.1 Cloud Computing . . . 5
  - 2.1.2 Prerequisite Terminology . . . 6
  - 2.1.3 NewSQL Systems . . . 7
  - 2.1.4 Data Pipelines . . . 9
  - 2.1.5 Big Data Engines . . . 11
  - 2.1.6 Open Table Formats . . . 13
  - 2.1.7 Infrastructure as Code . . . 14
- 2.2 Related Work . . . 15
  - 2.2.1 Development of NewSQL Systems . . . 15
  - 2.2.2 Development of Big Data Tools and New Storage Frameworks . . . 16
  - 2.2.3 Existing Literature on Data Migration Strategies . . . 17
- 2.3 Summary . . . 19

3 Data Preparation 20
- 3.1 Building Premigration Data Lake on S3 . . . 20
  - 3.1.1 Data . . . 20
  - 3.1.2 Pre-migration Process . . . 21
  - 3.1.3 Preparing Partitioned GZIP files from the Downloaded Data . . . 23
- 3.2 Building a Distributed Database . . . 24
  - 3.2.1 Types of shards . . . 24

3.2.2 Defining and Building a Sharding database . . . 25
3.2.3 Configuration decisions of ShardingspherеProxy . . . 27
3.2.4 Building a Horizontal Sharded Database . . . 27
3.3 Database Security . . . 29
3.4 Summary . . . 31

4 Shard-by-Shard Strategy 32
4.1 Introduction . . . 32
4.2 Shard-by-Shard Algorithm . . . 32
4.2.1 Sequential Shard-by-Shard Strategy - Balanced Shards . . . 33
4.2.2 Running Example of Sequential Shard-by-Shard Strategy . . . 35
4.2.3 Sorting Based Sequential Shard-by-Shard Strategy - Imbalanced Shards . . . 37
4.2.4 Concurrent Shard-by-Shard Strategy for Balanced Shards . . . 39
4.2.5 Sorting Based Concurrent Shard-by-Shard Strategy - Imbalanced Shards . . . 42
4.3 Experiments and Evaluations . . . 44
4.3.1 Experimental Conditions . . . 45
4.4 Experimental Results for 10 Million Dataset . . . 45
4.4.1 Sequential Shard-By-Shard Algorithm for Balanced and Imbalanced Shards . . . 46
4.4.2 Concurrent Shard-By-Shard Algorithm for Balanced and Imbalanced Shards . . . 47
4.5 Experimental Results for 200 Million Dataset . . . 50
4.5.1 Sequential Shard-By-Shard Algorithm for Balanced and Imbalanced Shards . . . 50
4.5.2 Concurrent Shard-By-Shard Algorithm for Balanced and Imbalanced Shards . . . 50
4.6 Summary . . . 52

5 Aggregated Shards Based Strategy 53
5.1 Introduction . . . 53
5.2 Aggregated Shards Based Algorithm . . . 53
5.2.1 Sequential Aggregated Shards Based Strategy - Balanced Shards . . . 54
5.2.2 Running Example of Sequential Aggregated Shards Based Strategy 55
5.2.3 Sorting Based Sequential Aggregated Shards Based Strategy - Imbalanced Shards . . . 59
5.2.4 Running Example of Sorting Based Sequential Aggregated Shards . 59
5.2.5 Concurrent Aggregated Shards Based Strategy for Balanced Shards 61
5.2.6 Sorting Based Concurrent Aggregated Shards Based Strategy - Imbalanced Shards . . . 68
5.3 Experimental Conditions and Evaluations Performed . . . 69
5.4 Experimental Results for 10 Million Dataset . . . 70
5.4.1 Sequential Aggregated Shards Based Algorithm . . . 70
5.4.2 Concurrent Aggregated Shards Based Algorithm . . . 78

5.5 Experimental Results for 200 Million Dataset 79
5.5.1 Sequential Aggregated Shards Based Algorithm 79
5.5.2 Concurrent Aggregated Shards Based Algorithm 82
5.6 Summary 83

6 Conclusions **87**
6.1 Accomplished Objectives 87
6.1.1 Evaluating the efficiency of migration strategies 87
6.1.2 Cost optimization of cloud services and Big data systems 90
6.2 Future Work 91
6.2.1 Optimizing Partition Functions 91
6.2.2 Stream Based Data Processing 91

Bibliography **92**

List of Tables

3.1 Business Domains Record Count. 21

4.1 Delta table after iteration 1. 36
4.2 Delta table after iteration 2. 36
4.3 Delta table after iteration 4. 37
4.4 Shard Size vs Threads output as Delta tables in Shard-by-Shard Strategy for Balanced Shards. 39
4.5 Threads vs Primary Memory Required In Balanced Concurrent Shard-by-Shard Strategy. 41
4.6 Threads vs Primary Memory Required for Imbalanced Concurrent Shard-by-Shard Strategy. 43
4.7 Threads vs Shards Execution Time in Balanced Shards. 48

5.1 Result of Union operation in iteration 1. 56
5.2 Result of Union operation in iteration 2. 57
5.3 Results of Union operation in iteration 1. 60
5.4 Results of Union operation in iteration 2. 62
5.5 Delta table result for $thread_1$ after iteration 1. 63
5.6 Delta table result for $thread_2$ after iteration 1. 63
5.7 Delta table result for $thread_1$ after iteration 2. 64
5.8 Delta table result for $thread_2$ after iteration 2. 64
5.9 Threads vs Primary Memory Required for Balanced Concurrent Aggregated Shards. 67
5.10 Shard Size vs Threads output in Delta tables for Concurrent Aggregated Shards Based Strategy. 67
5.11 Threads vs Primary Memory Required for Concurrent Aggregated Shards based strategy for Imbalanced Shards. 68

6.1 Sequential and Concurrent Strategy Runtimes and Performance Improvements. 88

List of Figures

2.1 NoSQL Migration Framework [17]. 18

3.1 Pre-Migration Data Lake Preparation Architecture. 22
3.2 Output of partitioned files in a S3 Directory. 24
3.3 Result from ShardingsphereProxy Executing Zcat Command. 26
3.4 Shardingsphere Deployment on AWS Cloud Instance. 28
3.5 Data Insertion flow Into Shardingsphere. 28
3.6 Image representing a Concurrent Data Pipeline 30
3.7 Image represents the tables being dropped. 30

4.1 Representation of Shard-by-Shard Techniques using Tree Diagram. 33
4.2 Illustration of Shard by Shard Strategy For Shard Size = 4. 35
4.3 Illustration of Imbalanced Shard by Shard Strategy For Shard Size = 4. . . . 38
4.4 Concurrent Shard-By-Shard Strategy of Thread Size 2. 40
4.5 Average runtime comparison between Imbalanced and Balanced shards in Sequential Shard-by-Shard Algorithm. 46
4.6 Effect of Threads on Balanced Concurrent Shard-by-Shard Algorithm. . . . 47
4.7 Effect of Threads on Imbalanced Concurrent Shard-by-Shard Algorithm. . . 49
4.8 Runtime comparison between Imbalanced and Balanced Shards in Sequential Shard-by-Shard Algorithm. 51
4.9 Effect of Threads on Imbalanced and Balanced Concurrent Shard-by-Shard Algorithm. 52

5.1 Tree Representation of Aggregated Shards Based Strategies. 54
5.2 Illustration of Aggregated Shards Based Strategy For Shard Size = 4. 55
5.3 Illustrating Imbalanced Aggregated Shards Based Strategy For Shard Size = 4. 61
5.4 Concurrent Aggregated Shards Based Algorithm for Thread Size 2 and Batch Size 2. 65
5.5 Effect of Shard Size on 32 GB and 4 GB Driver Memory in Balanced Shards. 72
5.6 Effect of Shard Size on 32 GB and 4 GB Driver Memory in Imbalanced Shards. 74
5.7 Effect of Driver Memory on various Batch Sizes for Shard Size of 4 and 10 in Balanced Shards. 76
5.8 Effect of Driver Memory on various Batch Sizes for Shard Sizes of 4 and 10 in Imbalanced Shards. 77
5.9 Effect of Threads on Batch Sizes for Balanced Concurrent Aggregated Shards Based Algorithm. 80

5.10 Effect of Threads on Batch Sizes for Imbalanced Concurrent Aggregated Shards Based Algorithm. 81
5.11 Runtime comparison between Imbalanced and Balanced Shards in Sequential Aggregated Shards Based Algorithm. 82
5.12 Effect of Threads on Balanced Concurrent Aggregated Shards Based Algorithm. 84
5.13 Effect of Threads on Imbalanced Concurrent Aggregated Shards Based Algorithm. 85

Chapter 1

Introduction

1.1 Introduction

Cloud computing systems are modern computing infrastructures that enable the development of cost-effective and scalable systems billed on a pay-as-you-go model [14]. In recent years following the pandemic, cloud adoption by various organizations has picked up significant momentum due to the limitations of on-premises systems [23]. Migration of such systems with data and applications from local systems to cloud systems [3] has become a challenging activity for many developers. Moreover, data migration from large databases is a complex activity that data engineers undertake as a daily routine. Therefore, migrating from big data-scale systems requires a meticulous selection of ETL (Extract, Transform, Load) [1, 2, 46] tools and a well-thought-out migration strategy. Without these considerations, data migration projects can become a collection of bad decisions that lead to chaotic data analytic results.

A Sharded Database system [29] is a system that partitions a huge table into multiple smaller tables. A table is partitioned by grouping records based on certain rules. Typically, the rules refer to a sharding function that groups the data based on defined sharding logic. Due to the grouping of rows, the read-write performance of a sharded database improves significantly due to reduced data volume and reduced index size [29]. Big tech organizations like Amazon, Meta and Google maintain and use heavily sharded databases [20, 42]. The data teams that work in these organizations create high-performance data pipelines

using big data technologies [46] and internal tools that transfer terabytes to petabytes of data daily. Moreover, data migration strategies as per our knowledge are usually not well documented and/or published in the literature because they evolve rapidly, and agile adaptation is important to make systems functional and deliver data to target systems. Therefore, experience through implementation and systems understanding is pivotal in understanding any migration strategy. A data pipeline involves the transportation of data from one place to another using programmatic methods. They are a key component in the data migration process and play a major role in data projects in terms of data.

In short, data migration strategies are mostly neglected in the literature due to their ever-changing behaviour. Documenting those strategies and maintaining the outcomes helps the data engineering community similar to software engineering. Over the past few years researchers have published some findings in the literature [21, 26, 44, 30], but this area still needs more exploration. Moreover, building and maintaining high-performance data pipelines is a cumbersome task that needs to be understood by every data team.

Therefore, the concept of cloud data migration in a big data setting with high-performance data pipelines needs to be studied more in order for cost-effective data migration strategies to be crafted in the area of modern cloud data engineering.

1.2 Motivation

An important task of a data engineer is to draft strategies for migrating data from one place to another using optimized data pipelines, providing cost-efficient solutions to an organization. This requirement is core to the motivation for our thesis because the strategies we propose will help the data engineering community implement faster data pipelines for organizations that use sharded databases for big data.

1.3 Contributions

This thesis primarily contributes to cloud data migration, data engineering, big data, distributed databases, and cloud computing. Overall our thesis aims at fulfilling the following objectives.

1. Drafting and implementing migration strategies, for migrating between sharded databases to cloud object stores.
2. Evaluating the performance of migration algorithms.
3. Evaluating the performance of our Data Pipeline.
4. Optimizing of the cost of cloud services and data migration.
5. Optimizing the migration of big data systems.

Therefore, from this thesis, one can choose the appropriate algorithms which are applicable to their migration task, from sharded databases to other systems.

1.4 Organization

The remaining parts of the thesis, are structured as follows.

1. In Chapter 2, we present the related work and background information that is required to understand various concepts and technologies used in the thesis.
2. In Chapter 3, we present the architecture of our proposed system and explain our data insertion process into ShardingSphere (To be defined in Chapter 2).
3. In Chapter 4, we propose multiple shard-by-shard algorithms and discuss how the algorithms perform.
4. In Chapter 5, we propose an aggregated shards-based algorithm and examine its performance using sequential and concurrent data pipelines.

5. Finally in Chapter 6, we conclude our work by summarizing and providing future research scope in the area of data engineering and data analytics.

Chapter 2

Background Study and Related Work

In this chapter, we will look at the background study and related work for this thesis. Section 2.1 summarizes the technologies and concepts that are required to comprehend the remaining chapters of the thesis. Similarly, Section 2.2 presents the related work that has been done in the existing area.

2.1 Background Study

2.1.1 Cloud Computing

Cloud computing is a paradigm for delivering computing services over the internet on a pay-per-use basis [14]. It involves the provision of various resources and services, including storage, processing power, networking, and software applications, through remote servers hosted in data centers. Users can access and utilize these resources and services on-demand, without the need for extensive local infrastructure or hardware investments. In general cloud computing platforms are classified into two categories:

1. Public Cloud: Public clouds are privately owned cloud platforms, that offer computing and storage over internet using pay-as-you go model. Examples of public cloud platforms include: Amazon Web Services, Azure Cloud Platform, Google Cloud Platform, and Digital Ocean.

2. Private Cloud: Private clouds are a dedicated cloud entity for an organization that handles and manages its own resources for its use cases. Examples of private cloud

platforms include: Digital Research Alliance Canada, and Private Data Centers maintained by Organizations.

Virtualization

Virtualization is a technology that allows the creation of virtual instances of physical hardware, such as servers, storage devices, or network resources. It enables multiple virtual machines (VMs) to run on a single physical server, with each VM operating as an independent system with its own operating system and applications. This abstraction of hardware resources allows for more efficient utilization, isolation, and management of computing resources.

For example, virtualization software is used by AWS to provision the virtual machines over the cloud platform.

Containerization

Containerization, on the other hand, involves encapsulating an application and its dependencies into a lightweight, portable container image. Containers share the same underlying operating system kernels and run as isolated processes on a host operating system. Containerization platforms such as Docker provide tools for building, managing, and deploying containerized applications. In this thesis, we are using Docker to create an Apache Spark container for performing our experiments.

2.1.2 Prerequisite Terminology

In this section, we will look into the important terms and definitions that we must understand before proceeding any further in this thesis.

- **Shards:** A shard generally means a small broken unit out of a big object. In a sharded database, a very large table data is organized into multiple tables using mathematical rules on a specific key.

Example: A table with a billion records is grouped into multiple tables of smaller size with smaller indexes.

- **Batch Size (n):** The number of shards processed in a specific iteration.
- **Thread Size (t):** The number of threads created to perform the data processing task.
- **Driver Memory (DM):** The primary memory allocated for the algorithm to process the source shards.
- **Shard Size:** The number of shards/tables present in a specific schema.
- **Delta Table:** A novel storage table format that enables ACID-compatible tables on cloud object stores. Delta tables can be created from the Delta Lake software package. In this thesis, we create delta tables as an output after processing the data.

2.1.3 NewSQL Systems

NewSQL [33] systems are a class of modern relational database management systems (RDBMS) that aim to provide the scalability of NoSQL systems while maintaining the ACID (Atomicity, Consistency, Isolation, Durability) properties of traditional SQL databases. There are three main categories of NewSQL systems:

1. New Architectures: In this type of system the DBMS is built from scratch implementing the NewSQL protocols, instead of extending the existing codebase of a DBMS server.
2. Transparent Sharding Middlewares: In this type, organizations build a middleware that shards the data on single node DBMS instance.
3. Database-as-a-service: In Database-as-a-service model the cloud provider maintains the NewSQL systems and provides all the services including automating the backup of data.

NewSQL Systems as Distributed Databases

Distributed databases [32] are a type of database system where data is stored across multiple physical locations. These locations can be on different computers or servers, often spread across various geographic areas. The primary goal of a distributed database is to provide efficient, scalable, and reliable access to data by distributing the workload and data storage, to where it is accessed the most. In this thesis, we use a distributed database to perform our experiments. However, unlike the data being distributed across geographic locations, we distribute the data across multiple tables in a single database and we query it as single table using a database proxy [29, 33]. Overall, every NewSQL system is a distributed database, whereas not every distributed database is a NewSQL system [33].

ShardingSphere

Apache ShardingSphere [29] is an open-source ecosystem of distributed database middleware solutions that facilitates the creation, management, and optimization of distributed databases. By providing a suite of tools and features, ShardingSphere aims to address the complexities associated with data sharding, replication, and distributed transactions, making it easier for organizations to build scalable and high-performance applications.

In a typical sharding database, there are two main types: Horizontal Sharding and Vertical Sharding. In this thesis, we use ShardingSphere to implement a Horizontal sharding strategy on the Postgres DBMS. Furthermore, Horizontal Sharding is employed for all our experiments.

Horizontal sharding

Horizontal sharding [29] is a technique that splits a huge existing table into multiple small tables based on defined sharding rules. It helps improve the performance of a database system by distributing the load. In practice, data personnel often get confused with Partitioning and sharding. Horizontal Partitioning usually refers to breaking a large table into multiple small tables. Similarly, sharding refers to Splitting the tables and distributing them across

multiple data sources.

Uses of horizontal sharding

1. It helps in distributing the volume by grouping the data into various tables based on Sharding rules defined.

2. Provides an efficient way to organize big data in OLTP [1] databases.

3. Helps in handling, updating and selecting large volumes of data concurrently with minimal response time.

Vertical sharding

Vertical sharding [29] involves splitting a database table vertically based on columns rather than rows. Each shard contains a subset of columns from the original table, allowing for efficient storage and retrieval of specific data attributes. This approach is useful for databases with tables containing many columns or varying access patterns, optimizing performance and resource utilization across distributed systems.

2.1.4 Data Pipelines

Data pipelines [34] are a sequence of interconnected data processing steps or stages that facilitate the automated flow of data from its source to its destination. These pipelines are designed to handle the extraction, transformation, and loading (ETL) of data, enabling organizations to efficiently move, manipulate, and analyze large volumes of data. There are multiple ETL/ELT [2] tools in the market, with most supporting batch and streaming data pipelines. Therefore, having knowledge of different types of pipelines helps in understanding the algorithms that we are going to implement. Usually there are two types of pipelines: a) batch pipelines, and b) real-time data streams.

[1] OLTP:(Online Transaction Processing)

[2] ELT: (Extraction,Load,Transformation), ETL (Extraction, Transformation, Load)

Batch pipelines

Batch pipelines [25] are a concept in data engineering and computer science where a series of data processing tasks are organized and executed sequentially in batches. These pipelines are designed to process large volumes of data efficiently, often in a scheduled or periodic manner. Batch pipelines typically consist of multiple stages, each performing a specific operation on the data before passing it to the next stage. In this thesis we propose and design batch strategies for migrating sharded databases.

Real time data streams

Real time data stream processing involves processing data from streaming sources whenever the data arrives into the system. Stream processing usually involves the sliding window technique [35] to process the data from the source system into the destination system. The sliding window protocol is a programmatic technique that is used in the implementation of continuous data streams. One common framework used in the industry is Kafka [28]. It follows the publisher/subscriber architecture and processes the data from streaming sources.

Datapipeline Engineering

Datapipeline engineering is the process of bringing in data by orchestrating and scheduling the workflows using programmatic frameworks. Data engineers frequently use these tools to move the data from one place to another for analysis. Therefore, understanding what pipeline engineering tools do helps in this thesis, as we use one for building our distributed database.

For Example, Apache Airflow[36], an open-source platform created by Airbnb and now under the Apache Software Foundation, is used for programmatically defining, scheduling, and monitoring workflows. It uses Directed Acyclic Graphs (DAGs) to define workflows in Python, allowing dynamic and organized task management. These types of tools are also called data Orchestrators.

In this thesis, we use Airflow [36] as an orchestrator to fetch the data from cloud object stores and will then insert into a Postgres Database using PSQL[3] command line tool.

2.1.5 Big Data Engines

Big data engines are software packages designed to process and analyze large volumes of data in a distributed and scalable manner. These engines are essential for deriving knowledge, patterns, and value from the huge datasets that are commonly known as big data. A few common characteristics of big data engines are:

1. Scalable Distributed Computing: Ability to scale existing system on high user demand.

2. Fault Tolerance: Ability of being resilient towards system failures.

3. Varied Data Processing Paradigms: Ability to provide batch and stream processing systems in a single framework.

4. Distributed file systems: Ability to support various file systems that enables access to files without any interruption.

5. Flexibility and Extensibility: Ability to add various libraries to the existing system and ease of integration.

HDFS

The Hadoop Distributed File System [19] is an open source distributed file system that is developed to handle the replication and storing of big data sets using low cost community hardware. It allows organizations to store the data with high reliability, and supports as storage system for processing Hadoop applications.

[3](PSQL Client) - Postgres Command line client

Apache Spark

Apache Spark is an open-source distributed computing system developed by Zaharia et al. [46], designed for big data processing and analytics. It provides a unified engine for batch processing, stream processing, interactive queries, and machine learning, making it a versatile tool for a wide range of data processing tasks. Apache Spark is known for its speed, scalability, and ease of use, making it one of the most widely adopted frameworks in the big data ecosystem. In this thesis we use Apache Spark to implement and evaluate our proposed strategies.

Apache Hadoop

Apache Hadoop [41] is an open-source framework for distributed storage and processing of large datasets across clusters of commodity hardware. It provides a scalable, fault-tolerant platform for storing and analyzing big data, making it a foundational component of the big data ecosystem. In this thesis, we use core hadoop libraries in apache spark to write the data from source systems to cloud object stores.

RDD

Resilient Distributed Datasets (RDDs) [46] are a fundamental abstraction in Apache Spark, serving as the primary data structure for distributed processing tasks. RDDs are designed to handle large-scale data processing across distributed clusters of machines efficiently. They provide fault tolerance, scalability, and in-memory computing capabilities, making them well-suited for a wide range of data processing applications.

DataFrame

A DataFrame is a data structure used for handling and manipulating data in tabular form [45]. It is a two-dimensional, labeled data structure with columns of potentially different types, similar to a table in a relational database or an Excel spreadsheet. DataFrames are a core component of data analysis and processing libraries such as Pandas in Python and

Apache Spark. In this thesis, we use the DataFrame API to process the data in Apache Spark.

Data Lake

A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale [31]. You can store data as-is, without having to first structure it, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning—to guide better decisions. In this thesis, we use S3 as data lake to store the data from the NYC taxi website [8].

2.1.6 Open Table Formats

Open table formats [9] are novel storage protocols that are developed to track the schema, data and column changes on big data systems. The initial work in this area has been implemented for incremental data over Hadoop file systems. Eventually it is evolved to support cloud object stores. Currently, there are three major table formats that are actively used in the big data community. They are: i) Apache Hudi, ii) Delta Lake, and iii) Apache Iceberg.

Apache Hudi

Apache Hudi [12], short for *Hadoop Upserts Deletes and Incrementals*, is an open-source data management framework. It is designed to simplify and accelerate big data processing and analytics workflows on Apache Hadoop and cloud storage systems like Apache Hadoop Distributed File System (HDFS) and Apache Hadoop-compatible storage layers in the cloud (e.g., Azure Blob Storage [4], Amazon S3 [24]). Apache Hudi is usually used from Apache Spark [46] to write data into cloud storage services like [24].

Delta Lake

Delta Lake [13] is an open-source storage layer that brings reliability, performance, and management capabilities to data lakes. Developed by Databricks, Delta Lake aims to address common issues associated with traditional data lakes, such as data corruption, lack of ACID (Atomicity, Consistency, Isolation, Durability) transactions, and challenges in handling both batch and streaming data in a unified manner. In this thesis, we use Delta Lake to create delta tables on the AWS cloud object stores [24].

Apache Iceberg

Apache Iceberg [11] is an open-source table format designed to improve the performance, scalability, and reliability of large-scale data lakes. Initially it was developed by Netflix and later Apple contributed to the project. Currently it is a part of the Apache Software Foundation. Apache Iceberg addresses many of the challenges associated with managing vast amounts of data in distributed environments, such as schema evolution and partition evolution.

2.1.7 Infrastructure as Code

Infrastructure as Code (IAC) [16] is the concept of provisioning computing infrastructure through machine readble definition files, instead of manually configuring harware or using interactive configuring tools. IAC tools help us in automation, and version control of the hardware configuration.

Terraform

Terraform [37] is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows users to define and provision infrastructure resources across various cloud providers, on-premises environments, and third-party services using declarative configuration files. In this thesis, we are using Terraform to provision the hardware virtual machines over AWS cloud platform.

2.2 Related Work

2.2.1 Development of NewSQL Systems

According to Pavlo et al. [33] at the beginning of the 21st century, many open source relational database system products (RDBMS) products gained widespread adoption. PostgreSQL [10] and MySQL [7] have become the de facto data management tool for many organizations. Due to the lack of scalability in single-node RDBMS, in early 2005 big tech organizations like Google and Facebook started pivoting towards NoSQL (Not Only SQL) systems that compromise on data consistency [40] but provide high availability. Initially, the organizations that implemented NoSQL systems were satisfied with the results. Later, they realized that most of a developer's time was devoted to fixing the data inconsistencies due to the lack of strong consistency.

This led to the proposal of a new type of system by Stonebraker et al. [39] in 2011 named NewSQL. NewSQL systems are a new type of data management system, that has both the scalability of a NoSQL system and also the consistency of a RDBMS system. NewSQL systems follow CAP (Consistency, Availability and Network Partition) Theorem [38] principles for providing consistency and availability (CA) over partition tolerance (P). The CAP Theorem [38] states that attaining Consistency (C), Availability (A) and Network Partition (P) cannot be achieved simultaneously in a distributed setting.

In 2016, Pavlo et al. [33] distinguished the properties of NewSQL systems with existing OLAP (Online Analytical Processing) database products, and established guidelines for what can be considered a NewSQL system. Also they categorized the systems into three domains: a) New architecture-based systems, b) Transparent sharding middleware, and c) Database-as-a-service systems. We have defined the above three categorizations in Section 2.1.3 of Chapter 2. Over time, many organizations implemented these systems internally for their OLTP (Online Transaction Processing) workloads.

In 2022, Li et al. [29] proposed an open-source system called ShardingSphere, which serves as transparent sharding middleware. This system is designed to address the challenges associated with database sharding by providing a robust middleware solution that can seamlessly distribute and manage data across multiple databases. The paper proposes the architecture of the database middleware and provides an overview of how the query engine is implemented. Additionally, the authors discuss its sharding capability and compare its performance with other existing NewSQL systems. So, refer to Section 2.1.3 for understanding more about NewSQL systems.

Later, as the throughput and availability problems of database systems were solved, the focus shifted towards analyzing these huge datasets for BI (Business Intelligence) and ML (Machine Learning) needs [27]. Therefore, the importance of cloud data migration for various systems became a necessity, as on-premises systems were unable to scale and process large datasets.

2.2.2 Development of Big Data Tools and New Storage Frameworks

To handle big data workloads, Zaharia et al. [46] built Apache Spark to extract and perform data processing in memory. To accommodate very large big datasets, the common destination for all source systems was cloud object stores such as AWS S3 (Simple Storage Service) [24]. Due to this, a new data management named Datalake [5, 31] evolved for managing data over cloud object stores. In Datalake the incremental data from tables is stored in an S3 bucket as a folder in an optimized Parquet [43] format with the date and timestamp appended at the end. Even though the data is present in an S3 folder, processing the incremental data requires all the datasets to be processed separately by big data tools before merging them into a single Parquet file. This situation has led to significant problems processing very large datasets that cannot fit in memory. Moreover, to build a data warehouse using a data lake, a data team must maintain an extra layer of ETL scripts to transform the data into dimensions and fact tables.

In 2020 Armbrust et al. [13] proposed the delta lake framework. It ensures transactional support and ACID properties over the Parquet files on cloud object stores [24] making it suitable for querying as a table, similar to RDBMS [10] systems. However in 2021 Armbrust et al. [15] proposed a new architecture called Lakehouse which brings in the best features of Data Warehouses, and Datalakes [5] by eliminating the need to have multiple ETL layers. Lakehouse [15] also supports querying data from Apache Spark [46] and can build Data Warehouses on top of cloud object stores within the same layer with indexing enabled over Parquet files.

2.2.3 Existing Literature on Data Migration Strategies

Elamparithi et al. [21] discussed various strategies, methodologies and tools involed in database migration. Primarily they summarized the key challenges faced during different types of migration (e.g schema migration, data migration). Also, they highlighted the key tools that are used in the industry, by comparing their features and provides recommendations for selecting the appropriate tools based on specific requirements.

Bansel et al. [17], proposed a framework for the cloud based migration of a NoSQL database into a graph database and Azure tables. The proposed framework transforms the existing NoSQL data format into target graph based systems. They proposed three algorithms that perform translations from Document to Graph, Document to Columnar, and Columnar to Graph. Overall, the authors evaluated the results of how the algorithms perform, by conducting experiments on migration of data from MongoDB (NoSQL) database to Neo4J (Graph) database and Azure tables. They found that migrating from NoSQL to graph databases saves 46% of space and also significantly reduces the number of nodes in the compressed graph database.

Alza et al. [30] devised a framework that migrates the data from RDBMS systems into a NoSQL database. The authors proposed an algorithm that reads the source tables by parsing

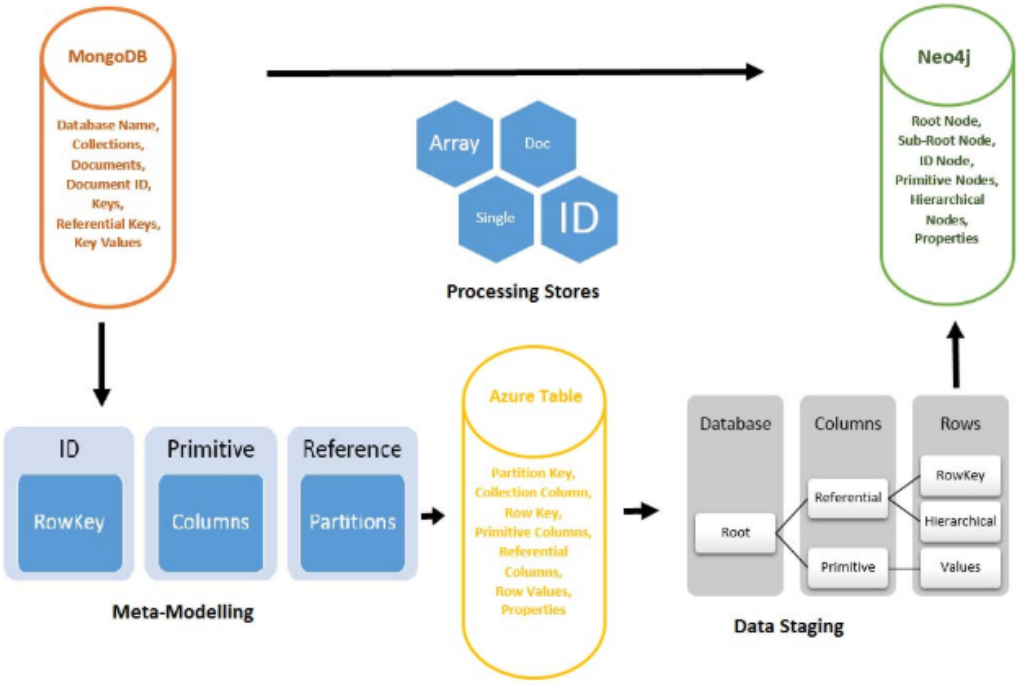

Figure 2.1: NoSQL Migration Framework [17].

the data and preparing it for inserting into a NoSQL database. For data with foreign keys, the algorithm generates the appropriate outputs. In this study, the authors used Microsoft SQL server as the source system, and MongoDB as the destination system.

Hussein et al. [26] proposes the comprehensive process of data migration to the cloud. It addresses the necessity of data migration, strategies involved, challenges faced, and methodologies used. The article categorizes different migration types, discusses associated risks, and highlights the benefits of cloud computing in this context. Additionally, it proposes a security model to enhance the efficiency and effectiveness of data migration when leveraging cloud technologies. They found that, using AES-256, SHA-512 and IDAs helps in migrating the data safely over the cloud.

Fahmi et al. [22] proposed various strategies and techniques to ensure smooth database migration while minimizing disruptions. They highlight the importance of careful plan-

ning, including thorough assessment, data profiling, and risk management. Key techniques discussed include phased migration, parallel testing, and validation processes to ensure data integrity. The article also emphasizes the need for effective communication and collaboration among stakeholders, as well as the utilization of automated tools and backup plans to address potential issues and ensure successful migration outcomes.

2.3 Summary

Overall, in this chapter, we first presented a summarized report regarding the concepts, tools, and technologies that are being used in this thesis. Secondly, we discussed the work that have been done in the area of NewSQL systems followed by innovations in big data systems. Moreover, we also discussed the existing literature proposed by several authors that introduce the concepts of data migration in cloud setting. In the next chapter, we will see how to insert the data into a distributed database using existing data engineering tools.

Chapter 3

Data Preparation

Preparing distributed sharded databases is the preliminary step before developing any migration strategies. The techniques involved in building very large sharded databases must be documented adequately in order to avoid the repetitive mistakes that are made while building distributed databases. Therefore, meticulous choices must be made from infrastructure provisioning to database development. Otherwise, database operations and maintenance will become an overburdened task for organizations instead of adding value to their technology goals.

In this chapter, we discuss various data preparation steps we use in building our distributed sharded databases. The structure of the chapter is as follows: Section 4.1 describes the source data and discusses about building *premigration datalake on S3*. Section 4.2 introduces the methodologies used to build a *distributed sharded database* [29]. This section mainly elaborates on the approach of how we build the data pipelines to insert data into shardingsphere. It also summarizes the design decisions and the concept of even distribution of data. Finally, in Section 4.3 we discuss the database security configurations that has been applied for the database in this thesis.

3.1 Building Premigration Data Lake on S3

3.1.1 Data

The first step of our data preparation is obtaining data for this work. We are using New York City (NYC) taxi data [8]. For the scope of this thesis, we limit data to the last 4.5

Business Domain	Records	Records in Million
Green taxi	10410841	10.4 Million
Yellow taxi	202208670	202.2 Million

Table 3.1: Business Domains Record Count.

years [January 2019 - July 2023] (As work on this thesis began in October 2023, data was only until July that year). The source data is hosted on Cloudfront by the NYC taxi team in Parquet [43] format.

The NYC taxi data [8] has primarily four business domains: Yellow Taxi, Green Taxi, For Hire Vehicles, and High Volume For Hire Vehicle Trip Records. The yellow taxi domain is the primary and oldest service that New York City cabs was operating. Later in 2013 and 2015 respectively the Green Taxi and For Hire Vehicle domains were added. Finally in 2019 the High Volume for-Hire Vehicles (HVFHV) domain was introduced to the New York community to provide better services to customers from a single taxi provider, carrying multiple customers from different vendors. In our work, we only use the first two categories which are yellow taxi and green taxi.

3.1.2 Pre-migration Process

To execute the pre-migration process, we use a single node Apache Spark engine running on a Docker container utilizing AWS r5.large instance capabilities. An r5.large instance is a Virtual Machine with 16 GB primary memory and 2 Virtual CPUs. We do not leverage the power of Apache Spark [46]; however, we utilized the power of instance main memory and the bandwidth associated with it.

Figure 3.1 depicts the architecture diagram of the process elaborated below. In this process, all the steps are implemented using asynchronous programming libraries in Python. We have used the **asyncio, aiohttp, and aiofiles** libraries to request, download and write data into the system concurrently. The steps are:

1. **Downloading the data** from the source system and writing it concurrently to the Docker container file system.

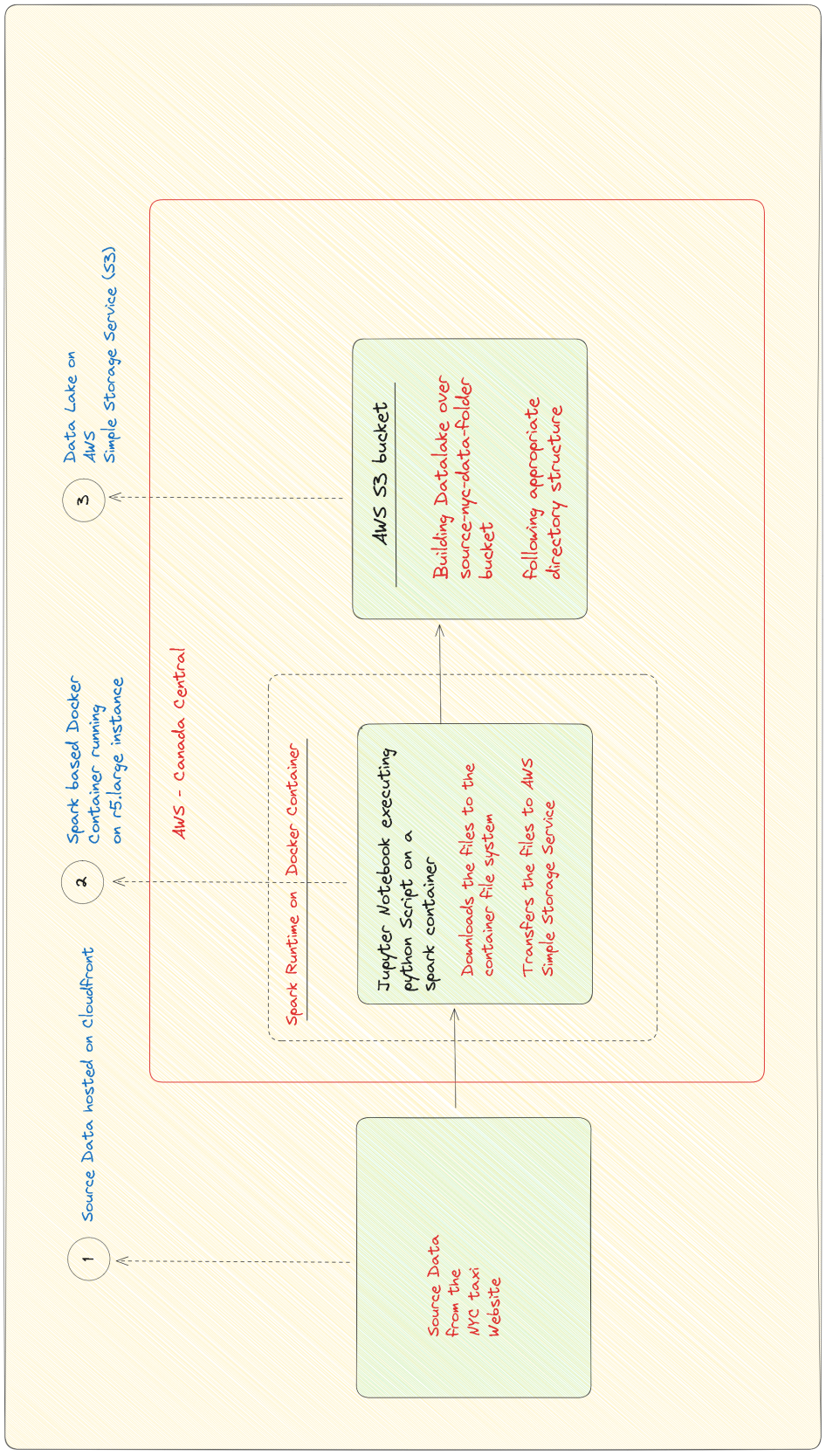

Figure 3.1: Pre-Migration Data Lake Preparation Architecture.

2. Once the data is downloaded to the container file system, another program **sends the data to the S3 bucket**.

3. After the data is transferred to S3 it is organized forming a pre-migration data lake [5] on S3.

Finally after the data transfer to the S3 bucket [24] is completed, the **source-nyc-data-folder** has approximately around 25.5 GB of Parquet [43] file data.

Obstacles Faced and Solutions:

During the local execution of the program (i.e desktop environment), Parquet files were downloaded as expected. When the same program was executed on an AWS EC2 instance over a Docker container [18], the following issues arose:

1. By default, Jupyter Notebook runs an event loop to execute programs sequentially. In our process, we create a nested event loop to download data. Therefore, the Python script running in a Jupyter Notebook encountered an event loop issue [6].

2. Due to making concurrent calls, the script could not download the files because the asynchronous calls were made continuously without waiting for previous ones to complete.

As a workaround for this problem, while running asynchronous programs inside Jupyter Notebook, we have used thread await methodology [6]. Thread await methodology helps us in avoiding the nested asynchronous loop issues that arise while running the above process.

3.1.3 Preparing Partitioned GZIP files from the Downloaded Data

The data downloaded is present on the cloud object store in a datalake. As the next step, we wrote a PySpark application to process the Parquet files into memory and partition them based on 1 million records. After partitioning, the PySpark program wrote the 1 million

records as GZIP files to the file system, and later they were moved to the object stores. Figure 3.2 refers to the GZIP files formed in the cloud object stores

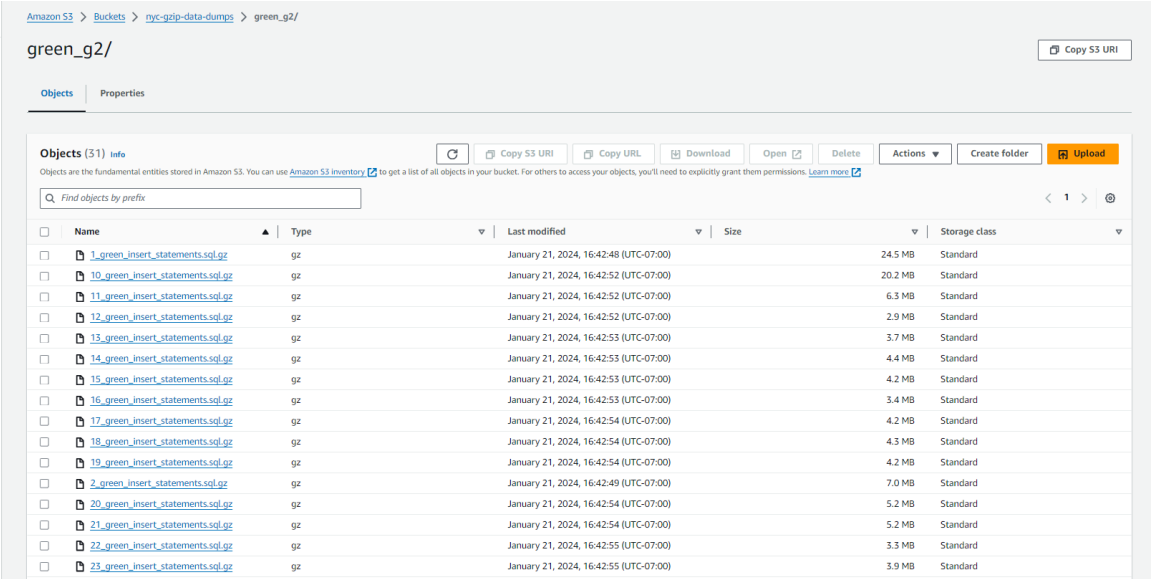

Figure 3.2: Output of partitioned files in a S3 Directory.

3.2 Building a Distributed Database

3.2.1 Types of shards

The prepared GZIP files in Subsection 3.1.3 for each business domain have two types of data. The first one is with a unique identifier, and the other one is without a unique identifier. The unique identifier datasets are used for building **balanced shards**, and the one without is used for building **imbalanced shards**.

Balanced Shards Balanced shards is a data management technique that organizes data into tables based on a unique key. It helps in effective data management by distributing data evenly across multiple shards.

Imbalanced Shards Imbalanced shards is a data management pattern that organizes data into tables using a shard key which is distributed randomly. In this technique we have

the flexibility of using any Sharding key from the given schema. Also data distribution in Shards will be imbalanced and data skewness occurs.

3.2.2 Defining and Building a Sharding database

There are mainly two ways to build a sharding database: a) Manually from the command line interface and b) Programmatically inserting from an orchestrator. Both are described below.

Building Sharding Databases from Command Line Interface (CLI)

Initially, we were building distributed databases using a bash command. The command in Listing 3.1 refers to the operation of inserting data into the database by reading records from the GZIP files. To explain more, firstly the zcat command reads the data from the compressed files and projects it onto console. Later, the pipe operator appends the statement from the zcat command and sends it to the PSQL client (Postgres Command line interface) that connects to the proxy making a network call. Once the proxy gets the statement, the parser parses it and redirects it to a specific shard based on the configured logic. In that way, until all the records projection from zcat is completed, the process repeats the above steps. Figure 3.3 represents the data insertion process using command line interface (CLI).

Listing 3.1: Script to Insert GZIP files Into Postgres via ShardingsphereProxy Using PSQL

```
zcat green_insert_statements_new.sql.gz | psql -U root -d sharding_db -p
    3308 -h 127.0.0.1
```

From the command line interface, we encountered the following issues:

1. While building distributed database, to insert 4 million records it took around 12 hours for us. So as per our estimation we anticipated much more higher time for larger datasets.

2. Due to manual observability and lack of auditing this process works only for smaller

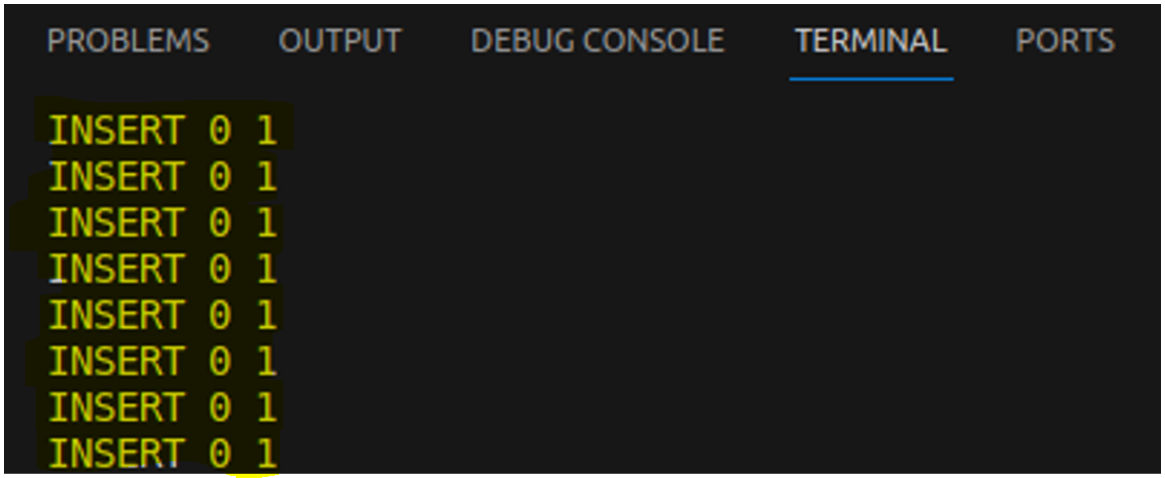

Figure 3.3: Result from ShardingsphereProxy Executing Zcat Command.

datasets.

3. Whenever there is a pipeline failure we must unzip the GZIP files and delete the records that are already inserted. This becomes cumbersome as the data increases.

Programmatically Inserting from an Orchestrator

In the above Section 3.2.2 we have presented how to insert data into Shardingsphere using the command line. In this section, we inform how to automate data ingestion using Airflow [36]. Please refer to Section 2.1.4 to learn more about Apache Airflow. When building databases manually for small datasets the CLI process is sufficient. However, for larger datasets which have 100s of millions of records, the issues mentioned in the Section 3.2.2 are encountered. Therefore we decided to create an Airflow Job [36] to automate the data insertion into the ShardingsphereProxy.

The idea of writing an Airflow job for data insertion into ShardingsphereProxy is fairly a simple process if someone already has experience in working with Airflow. However, connecting to ShardingsphereProxy from a remote instance using Airflow is an undocumented process with no available information to follow. In our implementation, we first download the data from the S3 folder shown in Figure 3.2 to the local file system. The script then

connects to ShardingsphereProxy remotely from the Airflow instance and performs network calls to insert records into the system by reading the GZIP data from the local file system

3.2.3 Configuration decisions of ShardingsphereProxy

Initially, we considered deploying ShardingsphereProxy on separate Cloud instances connected with Amazon RDS service. Amazon RDS is a highly available and reliable relational database on AWS Cloud. Due to cost constraints, we have adopted a different paradigm and have chosen to deploy the infrastructure on Docker as a container using a virtual machine with huge primary memory and extensive bandwidth.

Therefore, for the purpose of experimentation and development, we used a t2.2xlarge machine, which has 8 Virtual CPUs and 32 GB of primary memory. Later, we deployed the database and ShardingsphereProxy [29] as Docker containers, as shown in Figure 3.4. The ShardingsphereProxy and PostgreSQL database are connected via Docker network interfaces and communicate all the required information for sharding the incoming data.

3.2.4 Building a Horizontal Sharded Database

We have written a script to transfer the files that are generated from the process described in Section 3.1.3. From Figure 3.5 as a next step, we created an Airflow [36] job to pull the data into the local system. Later we unzip the files for insertion. Once the files are unzipped another Airflow [36] job reads the file paths from the file system and starts inserting data into ShardingSphere [29] using the bash operator. We have introduced this concept in the Section 3.2.2.

We validated the proof of concept of inserting data remotely from an external system in Section 3.2.2. Following this, we decided to insert data into ShardingSphere using Concurrent Data Pipelines. For this, we used the concept of a concurrent DAG (Directed Acyclic Graph) to insert the data into ShardingsphereProxy in parallel. This method is not reliable

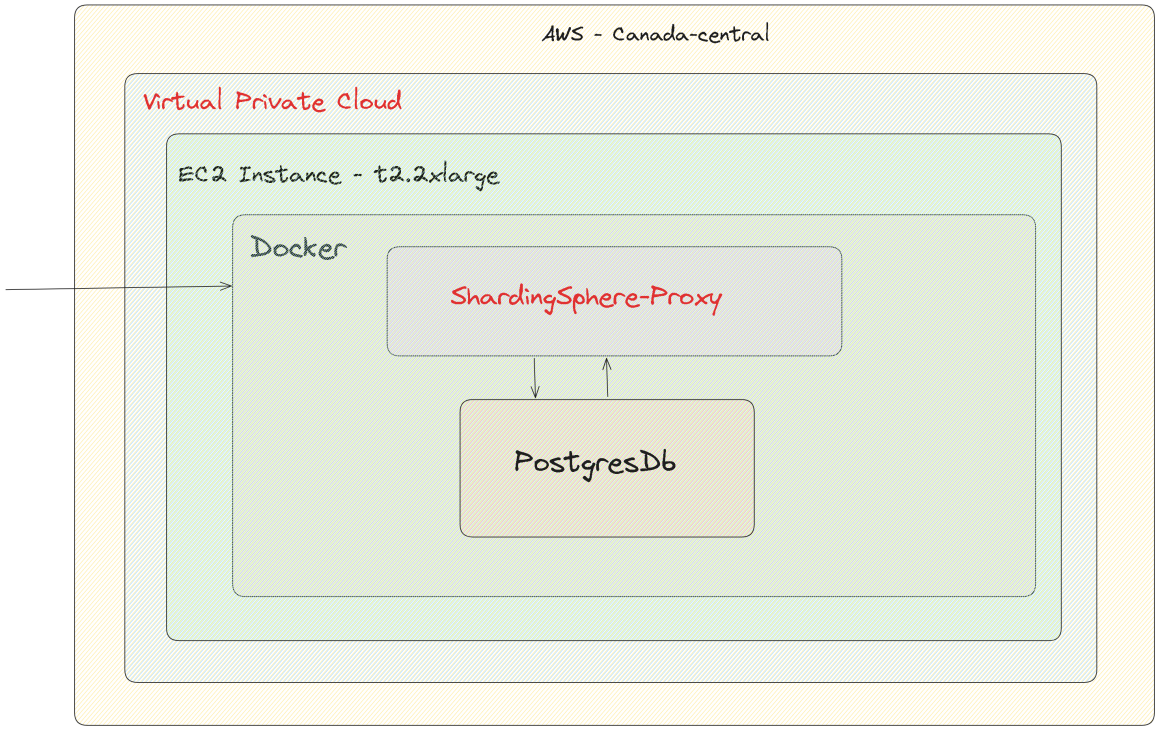

Figure 3.4: Shardingsphere Deployment on AWS Cloud Instance.

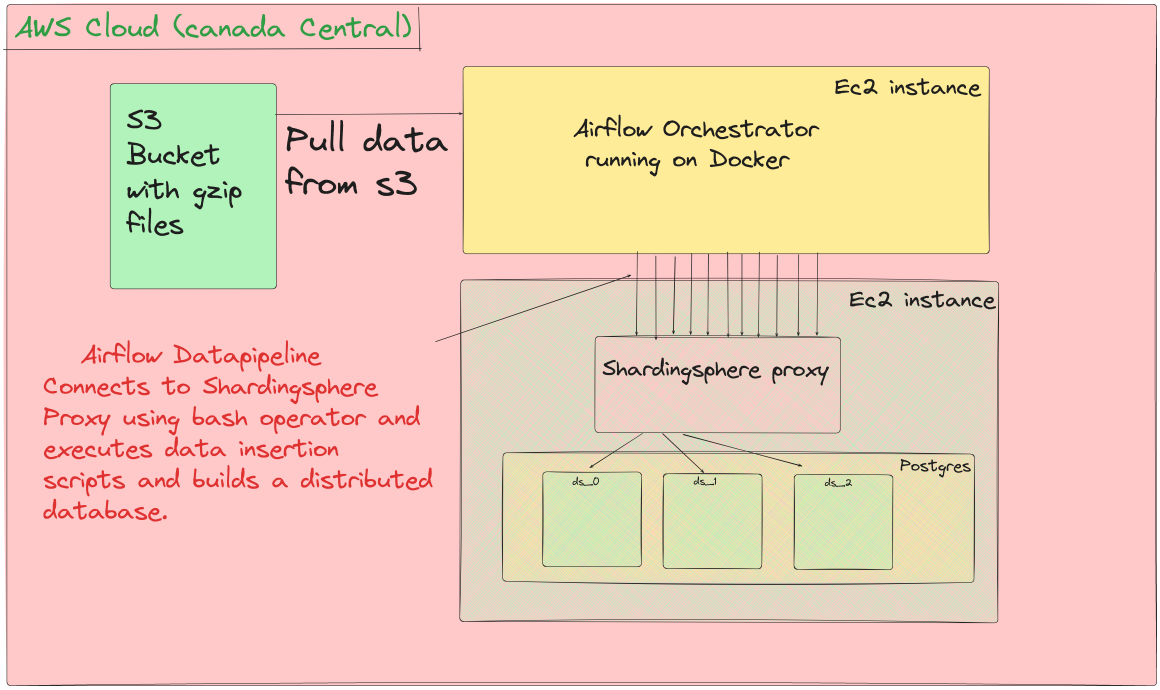

Figure 3.5: Data Insertion flow Into Shardingsphere.

but highly efficient due to its nature of design and execution. It ensures that data insertion happens in parallel utilizing the features of the underlying hardware.

During the pipeline execution, we encountered the following issues:

1. Whenever the data pipeline execution is in progress there is a chance of failure in the concurrent execution.

2. Debugging becomes difficult, as we must drill down to the root of the problem involving concurrent data pipelines.

Regardless of all the issues, we employed concurrent data pipelines strategy to build the distributed sharded databases saving time and cloud resources. Overall, concurrent data pipelines helped us insert the yellow taxi and green taxi data into our Horizontal Sharding instance. After data insertion, we have 10.4 million records for green taxi and 202.2 million records for yellow taxi. Table 3.1 provides the record count of different domains and Figure 3.6 represents the concurrent data pipeline.

3.3 Database Security

In December 2023, when we were in initial stages of experiments, we encountered a database hack on our cloud sharding database. It resulted in loss of 1 month of work. Initially, we were using a development instance with weak passwords to test out our strategies. As the password was easy to crack the attacker was able to login into the system and wiped off all our sample tables by dropping the database. Also, there was a note in a table where they were demanding bitcoin to release it. From that incident, we have learnt our lessons and have secured our databases by implementing strict security group policy. Below, Figure 3.7 represents the action log of the hacker dropping the tables from the database.

Figure 3.6: Image representing a Concurrent Data Pipeline

```
PostgreSQL Database directory appears to contain a database; Skipping initialization

2023-12-12 13:21:57.520 MST [1] LOG:  starting PostgreSQL 16.0 (Debian 16.0-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
2023-12-12 13:21:57.521 MST [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
2023-12-12 13:21:57.521 MST [1] LOG:  listening on IPv6 address "::", port 5432
2023-12-12 13:21:57.527 MST [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-12-12 13:21:57.542 MST [29] LOG:  database system was shut down at 2023-12-09 23:50:16 MST
2023-12-12 13:21:57.919 MST [1] LOG:  database system is ready to accept connections
2023-12-12 13:21:58.196 MST [33] LOG:  incomplete startup packet
2023-12-12 13:26:57.547 MST [27] LOG:  checkpoint starting: time
2023-12-12 13:26:57.564 MST [27] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.003 s, sync=0.003 s, total
=0.018 s; sync files=2, longest=0.002 s, average=0.002 s; distance=0 kB, estimate=0 kB; lsn=0/523FA320, redo lsn=0/523FA2E8
2023-12-12 15:10:48.941 MST [2018] FATAL:  database "template0" is not currently accepting connections
2023-12-12 15:10:49.796 MST [716] FATAL:  terminating connection due to administrator command
2023-12-12 15:10:49.796 MST [715] FATAL:  terminating connection due to administrator command
2023-12-12 15:10:49.796 MST [717] FATAL:  terminating connection due to administrator command
2023-12-12 15:10:49.796 MST [1714] FATAL:  terminating connection due to administrator command
2023-12-12 15:10:51.281 MST [2021] ERROR:  cannot drop the currently open database
2023-12-12 15:10:51.281 MST [2021] STATEMENT:  DROP DATABASE postgres;
2023-12-12 15:10:51.804 MST [27] LOG:  checkpoint starting: immediate force wait
2023-12-12 15:10:51.814 MST [27] LOG:  checkpoint complete: wrote 2 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.003 s, sync=0.002 s, total
=0.010 s; sync files=2, longest=0.002 s, average=0.001 s; distance=129 kB, estimate=129 kB; lsn=0/5241A7B8, redo lsn=0/5241A780
2023-12-12 15:15:51.915 MST [27] LOG:  checkpoint starting: time
2023-12-12 15:17:23.936 MST [27] LOG:  checkpoint complete: wrote 922 buffers (0.7%); 0 WAL file(s) added, 0 removed, 0 recycled; write=92.014 s, sync=0.003 s, to
tal=92.022 s; sync files=301, longest=0.002 s, average=0.001 s; distance=4262 kB, estimate=4262 kB; lsn=0/528442E0, redo lsn=0/528442A8
2023-12-12 15:23:36.073 MST [2285] FATAL:  database "ds_0" does not exist
2023-12-12 15:23:36.333 MST [2286] FATAL:  database "ds_0" does not exist
```

Figure 3.7: Image represents the tables being dropped.

3.4 Summary

Overall, in this chapter we presented how to prepare the data for building a distributed sharded database. Additionally, we highlighted important configuration decisions and described the strategies for building the database using different data pipelines. In the next chapter, we will look at the shard-by-shard strategy.

Chapter 4

Shard-by-Shard Strategy

In the previous chapter, we discussed how to create a distributed database and explored various methods to optimize it. Now, in the next chapter too, we will propose several migration strategies that quickly pull data into cloud object stores and enable us to perform analytics. The first algorithm we propose is a shard-by-shard strategy. This is the basic algorithm that will help us understand later chapters.

4.1 Introduction

In general, when we want to migrate several tables from a database to a target system, we read one table at a time and copy it to the other system. We took this common approach to propose our first strategy.

For our proposed shard-by-shard strategy, we read the tables one by one from a sharded database and write them as a single delta table to cloud object stores. We use Apache Spark [46] to implement the algorithms and to handle huge amount of data from sharded databases. Spark uses a Resilient Distributed Datasets (RDD) technique to process the data in and out of memory. Resilient Distributed Datasets are the foundation of big data processing. Please refer to Section 2.1.5 to understand more about Spark and RDDs.

4.2 Shard-by-Shard Algorithm

The term shard-by-shard originates from the idea of processing one shard at a time in memory. In a shard-by-shard algorithm, the batch size is 1 (n = 1), which means only a

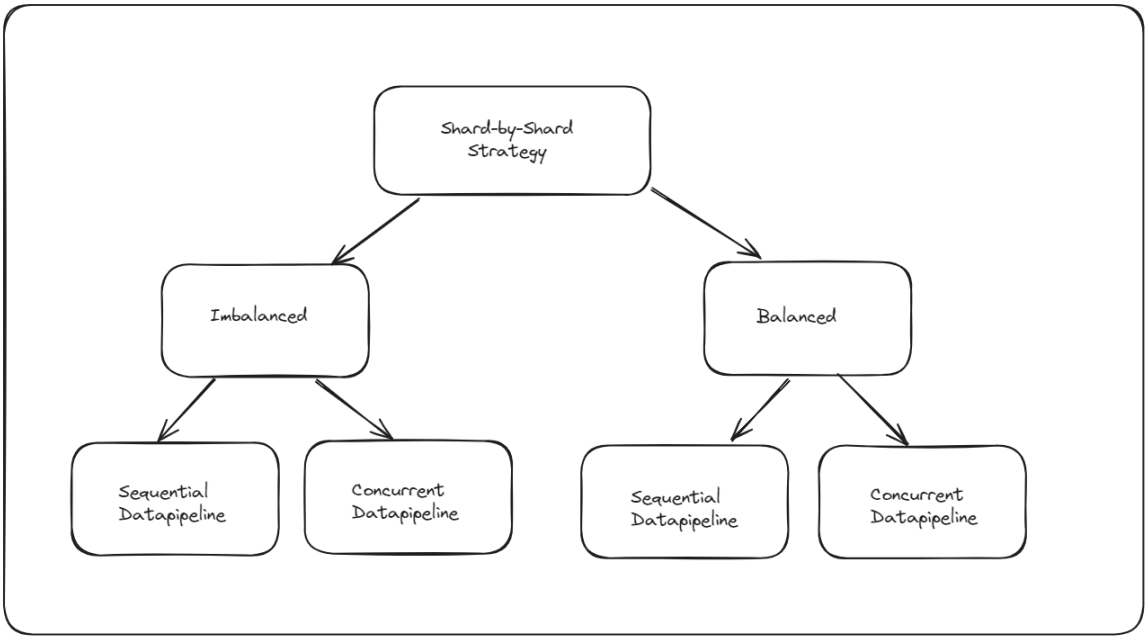

Figure 4.1: Representation of Shard-by-Shard Techniques using Tree Diagram.

single table is processed in memory by a single thread. A sharded database can consist of either imbalanced or balanced shards. Further, each type of shards can be processed either sequentially, or concurrently. Figure 4.1 depicts the multiple strategies that will be proposed in this chapter.

4.2.1 Sequential Shard-by-Shard Strategy - Balanced Shards

The first shard-by-shard algorithm is for balanced shards using sequential processing refer to Algorithm 1, 2. In this type of system, all shards consists of nearly the same number of records. So we process them sequentially, and write each as a delta table. The inputs to the algorithm are a list of shard names and a batch size of n = 1. The batch size of n = 1 states that we only read one table at a time when making a network call to the database from Spark application. Additionally, we can determine the iterations using the following formula for our estimation purposes.

$$numberOfIterations = \left\lceil \frac{shardSize}{batchSize} \right\rceil$$

Algorithm 1: sequentialShardByShard(Shards, n)

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n // Batch Size

Output: Delta table // A large single table.

```
1  i = 0;
2  while length(Shards) > 0 do
       // Getting the shard names out of the queue
3      shardName = Shards.dequeue();
       // Read shard data into Primary memory
4      dataFrame = readData(shardName);
       // Creating the Base DeltaTable
5      i ← i + 1;
6      if i = 1 then
7          DeltaTable := dataFrame;
8      end
       // Appending the new dataFrame to the existing DeltaTable
9      else
10         DeltaTable := DeltaTable.append(dataFrame);
11     end
12 end
```

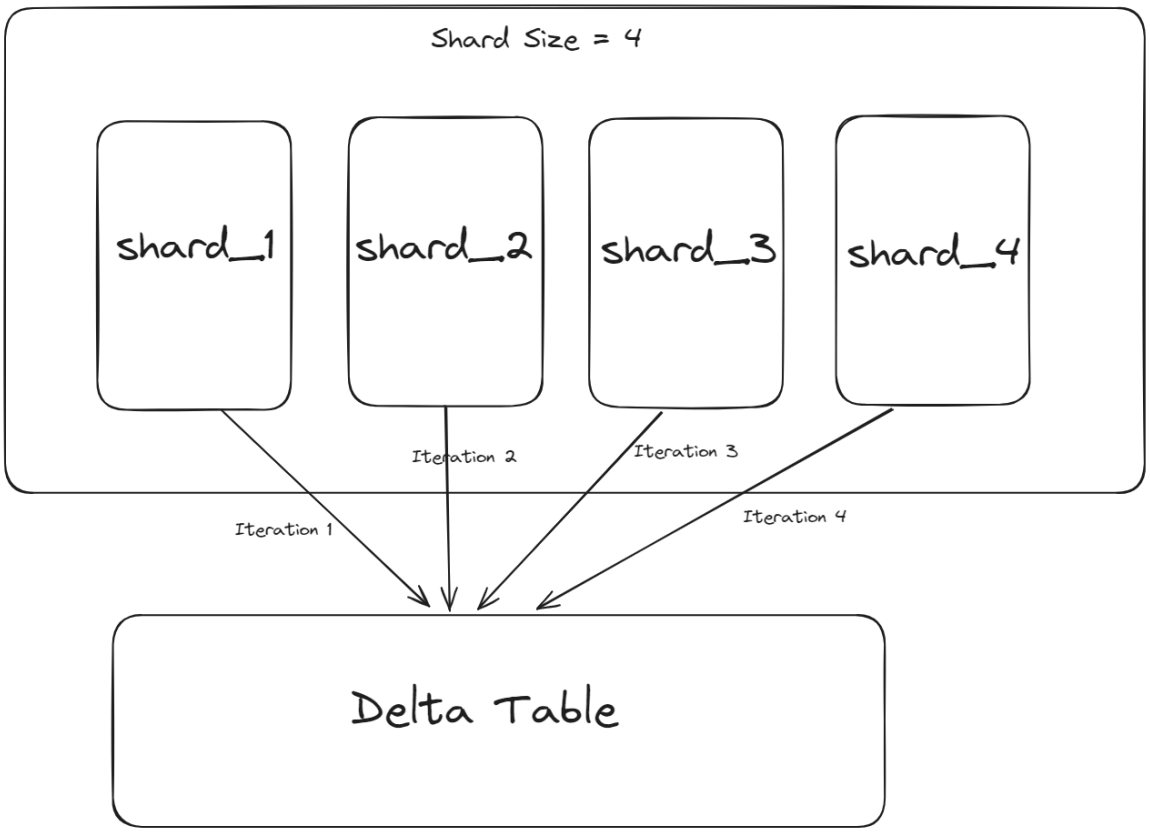

Figure 4.2: Illustration of Shard by Shard Strategy For Shard Size = 4.

Algorithm 2: Execution of Sequential Shard-By-Shard for Balanced Shards

Input : shardIdentifier = ”shardPrefix”,
n = 1

Output: A Single Delta table

```
1 shardNames = getTableNames(ShardIdentifier);
  // Invocating sequential shard-by-shard strategy
2 sequentialShardByShard(shardNames, n);
```

4.2.2 Running Example of Sequential Shard-by-Shard Strategy

We now illustrate the Algorithm 1 using an example. As per Figure 4.2, let us assume all shards contain a total of 12 records, with each shard having three records. Now, when we pass the shards list to the algorithm, as per the batch size, the number of iterations are

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
4	2024-07-01 12:10:00	2024-07-01 12:35:00	3	4.5
8	2024-07-01 18:45:00	2024-07-01 19:10:00	3	6.2
12	2024-07-02 08:30:00	2024-07-02 08:50:00	2	4.8

Table 4.1: Delta table after iteration 1.

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
4	2024-07-01 12:10:00	2024-07-01 12:35:00	3	4.5
8	2024-07-01 18:45:00	2024-07-01 19:10:00	3	6.2
12	2024-07-02 08:30:00	2024-07-02 08:50:00	2	4.8
1	2024-07-01 08:15:00	2024-07-01 08:30:00	1	3.2
5	2024-07-01 14:00:00	2024-07-01 14:20:00	4	6.0
9	2024-07-01 20:20:00	2024-07-01 20:35:00	1	2.0

Table 4.2: Delta table after iteration 2.

determined to be 4 by the formula defined in section 4.2.1. Subsequently, we dequeue the data structure and read 3 records into a data frame. Later we increment the value of i by 1.

1. After the increment operation is executed, in iteration 1 we will check, the if condition as per line 6 in Algorithm 1 to create a base delta table, with the first 3 records from $shard_1$. Table 4.1 refers to the output of the delta table after the operation.

$$DeltaTable_1 := dataFrame_1$$

2. In iteration 2, we will dequeue the second shard name from the list, read the 2nd shard data into memory, and append the data to the existing delta table. After this operation, the delta table will have 6 records in it and we denote it as ($DeltaTable_2$). Table 4.2 refers to the result after iteration 2.

$$DeltaTable_2 := DeltaTable_1 +\!\!+^{4} dataFrame_2$$

[4] $+\!\!+$ = Append Operation

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
4	2024-07-01 12:10:00	2024-07-01 12:35:00	3	4.5
8	2024-07-01 18:45:00	2024-07-01 19:10:00	3	6.2
12	2024-07-02 08:30:00	2024-07-02 08:50:00	2	4.8
1	2024-07-01 08:15:00	2024-07-01 08:30:00	1	3.2
5	2024-07-01 14:00:00	2024-07-01 14:20:00	4	6.0
9	2024-07-01 20:20:00	2024-07-01 20:35:00	1	2.0
2	2024-07-01 09:00:00	2024-07-01 09:25:00	2	5.1
6	2024-07-01 15:30:00	2024-07-01 15:50:00	1	3.8
10	2024-07-01 22:00:00	2024-07-01 22:20:00	2	4.0
3	2024-07-01 10:45:00	2024-07-01 11:00:00	1	2.7
7	2024-07-01 17:00:00	2024-07-01 17:30:00	2	7.5
11	2024-07-02 07:00:00	2024-07-02 07:20:00	1	3.5

Table 4.3: Delta table after iteration 4.

3. Similarly, in iteration 3, the algorithm makes a network call to the database to read the records from 3rd shard into a data frame and appends it to the existing delta table as ($DeltaTable_3$) with 9 records. Likewise, for iteration 4, we perform the same operation and there will be a total of 12 records. Table 4.3 depicts the result of the delta table after iteration 4.

$$DeltaTable_3 := DeltaTable_2 +\!\!+\, dataFrame_3$$

$$DeltaTable_4 := DeltaTable_3 +\!\!+\, dataFrame_4$$

Overall, after all the above operations are completed, the resulting delta table ($DeltaTable_4$) will have a total of 12 records.

4.2.3 Sorting Based Sequential Shard-by-Shard Strategy - Imbalanced Shards

The second approach is a sorting-based sequential shard-by-shard strategy tailored for imbalanced shards. The key distinction in this strategy is found in the initial steps of the

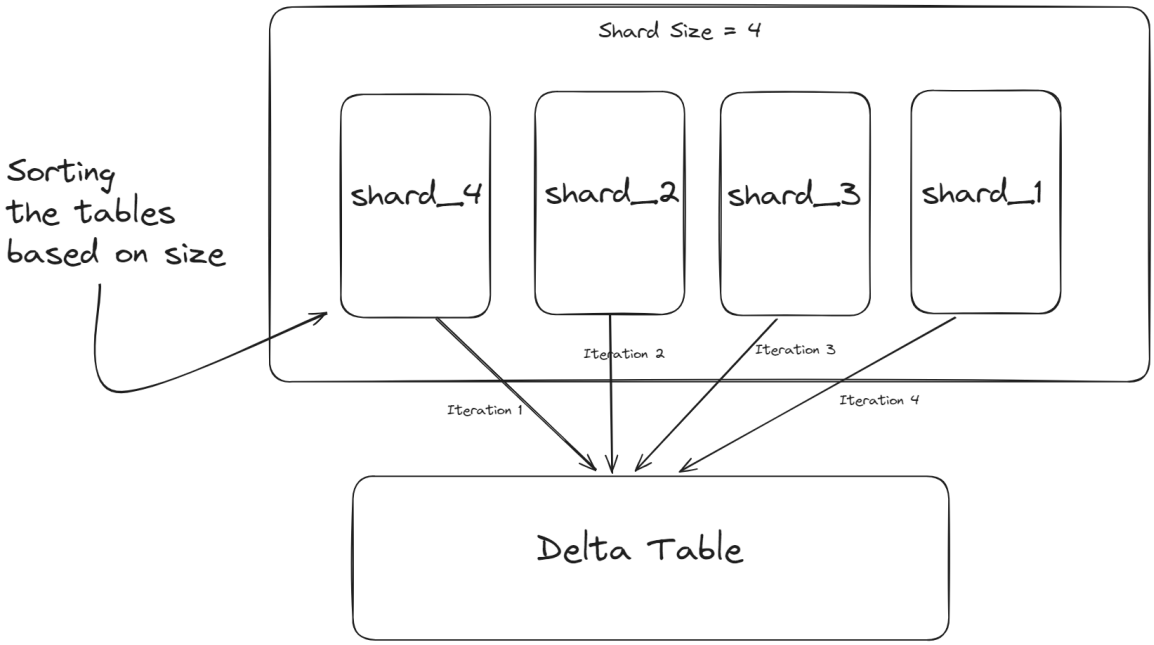

Figure 4.3: Illustration of Imbalanced Shard by Shard Strategy For Shard Size = 4.

algorithm. In this scenario, the rows in each shard are distributed according to a shard key that is not a primary key. As a result, the shards become imbalanced. To address this, we sort the shards based on their size of total rows in ascending order. Although various parameters could be chosen for sorting, we opted for size of total rows due to its simplicity. Algorithm 3 refers the steps that are performed in the strategy.

Algorithm 3: Sorting Based Sequential Shard-by-Shard Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

$n = 1$

Output: A Single Delta table

```
// Sorting the shards based on total size of records
getSortedList = sortBasedOnTableSize(Shards);
// Invocating the shard-by-shard algorithm
sequentialShardByShard(getSortedList, n);
```

Shard Size	t = 2	t = 3	t = 4
4,6,8,10	2	3	4

Table 4.4: Shard Size vs Threads output as Delta tables in Shard-by-Shard Strategy for Balanced Shards.

4.2.4 Concurrent Shard-by-Shard Strategy for Balanced Shards

The third strategy we propose is a concurrent shard-by-shard algorithm for balanced shards. In this technique, instead of using a single thread, we employ multiple threads to make concurrent network calls to the database. Before assigning tables to each thread, we partition the table list using a partition function and then assign the resulting partitions to each thread. We discuss the partition function that we use on the next page. Each thread then retrieves data into primary memory for processing. Once data fetching is completed by each thread, the data is written in parallel to the cloud object store as tables. Consequently, we will have t number of tables after migration for a given shard size. Figure 4.4 depicts the concurrent shard-by-shard strategy of thread size 2. Similarly, Algorithm 4 refers to the partition function used and Algorithm 5 refers to the execution of concurrent shard-by-shard strategy.

In this strategy, we primarily increase threads to enhance the performance of the algorithm. Table 4.4 refers to the resulting delta tables using different thread sizes for varied shard sizes.

Partition Function

The partition function plays an important role in preparing the table list of partitions, which makes the algorithm run faster. The Algorithm 4 creates partitions in a mixed order from a sorted list of table names in ascending order. However, the proposed partition function is not optimized, and there is definitely room for improvement.

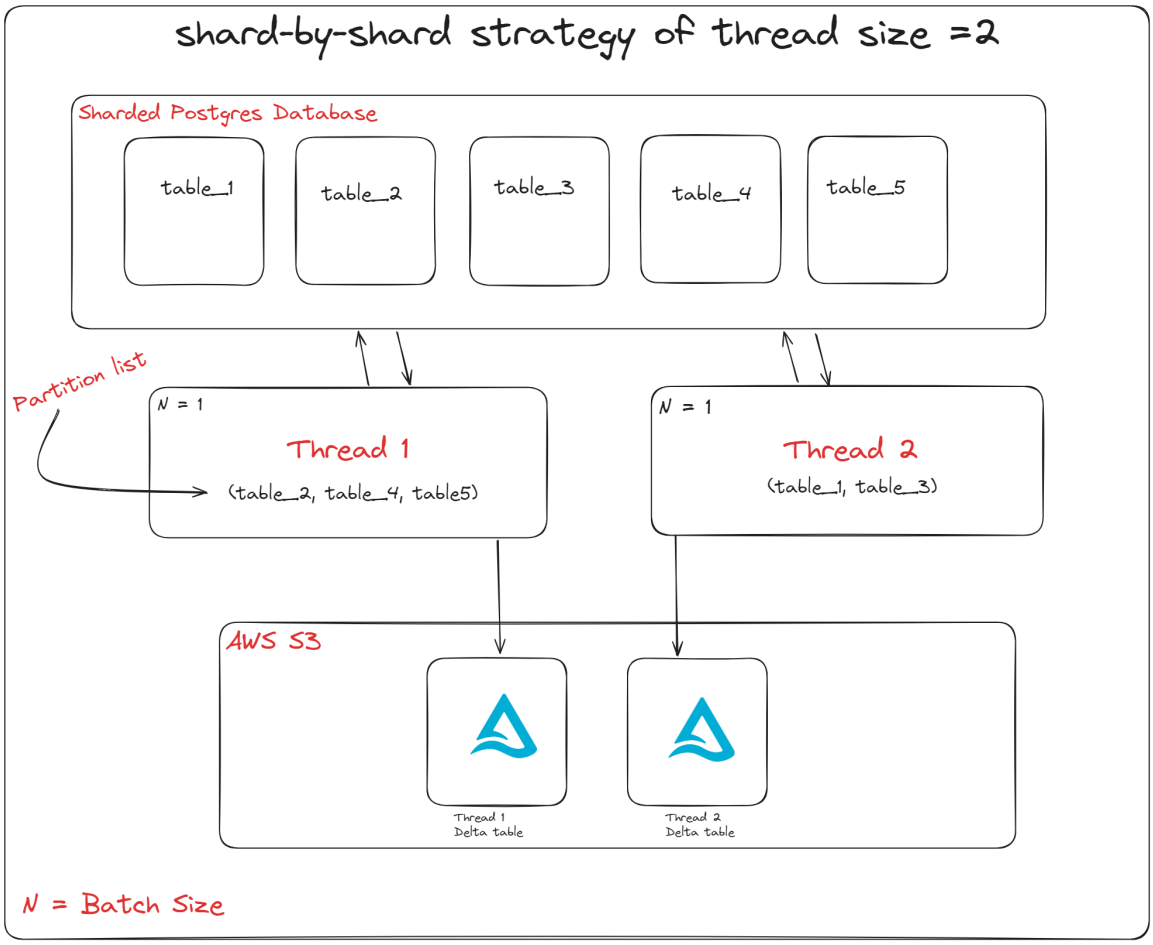

Figure 4.4: Concurrent Shard-By-Shard Strategy of Thread Size 2.

Algorithm 4: Partition Function for Concurrent Shard-By-Shard Strategy

```
1  Function getPartitionedTableNames(tableNames, numThreads):
2      k = 0;
3      partitionedTablenames = 1D array of empty lists with the size of numThreads;
       // Generating partition lists based on the thread count
4      while length(tableNames)>0 do
5          tablename = tableNames.dequeue();
6          partitionedTablenames[k].append(tablename);
7          k += 1;
8          if k = numThreads then
9              k = 0;
10     return partitionedTablenames;
```

Shard Size	Size of Each Table	t = 2	t = 3	t = 4
4	573 MB	>1146 MB	>1719 MB	>2292 MB
6	381 MB	>762 MB	>1143 MB	>1524 MB
8	285 MB	>570 MB	>855 MB	>1140 MB
10	228 MB	>456 MB	>684 MB	>912 MB

Table 4.5: Threads vs Primary Memory Required In Balanced Concurrent Shard-by-Shard Strategy.

Resource planning for balanced shards of 10 million dataset

Resource planning is a very important step before implementing the concurrent strategy. While executing various threads, we read data from multiple tables into memory. Therefore, we must allocate the required amount of primary memory to seamlessly perform the data transfer without any hindrances. Determining the primary memory for balanced shards is simpler compared to imbalanced shards because all shards in the database contain an equal number of records. Therefore for balanced shards, we use the size of any shard, which represents the minimum memory a thread must allocate. Later, in order to determine the amount of memory required we multiply the table size with the number of threads.

$$memoryForThreads = (highestTableSize * numberofThreads)$$

Table 4.5 refers to the resource allocation values for balanced shards used in our experiments for a dataset of 10 million records.

Algorithm 5: Balanced Concurrent Shard-by-Shard Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n = 1 (n = Batch Size),

numThreads = {2,3,4,5} (t = Thread Size)

// The function(s) below are called once per value of t.

Output: A Delta table for each thread

```
// Creates Partitions based on partition function in mixed order
   as list of lists
1 partitionedTables = getPartitionedTableNames(Shards, numThreads);
// Creating threads and passing the partition lists to the
   algorithm
2 for i ← 1 to numThreads do
3 |   sequentialShardByShard(partitionedTables[i], n);
4 end
```

4.2.5 Sorting Based Concurrent Shard-by-Shard Strategy - Imbalanced Shards

The fourth algorithm in this chapter is the concurrent shard-by-shard strategy for imbalanced shards. This technique adheres to the constraint of reading one table into memory, as detailed in section 4.2.1. We leverage multiple threads to process sorted, imbalanced shards concurrently, employing a partition function to distribute the workload.

Resource planning for Imbalanced shards of 10 million dataset

Unlike the resource allocation for balanced shards, in imbalanced shards we follow a different paradigm.

1. As a first step, we fetch the maximum table size using existing DBMS functions.

2. Next, we allocate driver memory based on the number of threads we are going to

Shard Size	Maximum Table Size	t = 2	t = 3	t = 4
4	556 MB	>1112 MB	>1668 MB	>2224 MB
6	356 MB	>712 MB	>1068 MB	>1424 MB
8	420 MB	>840 MB	>1260 MB	>1680 MB
10	348 MB	>696 MB	>1044 MB	>1392 MB

Table 4.6: Threads vs Primary Memory Required for Imbalanced Concurrent Shard-by-Shard Strategy.

create.

3. Later, we multiply the table size with the number of threads and will make sure that we are always allocating the memory greater than the calculated value.

Table 4.6 refers to the resource allocation values for 10 million dataset (Green Taxi). Using the provided table information, we can appropriately assign resources for various shard sizes.

Algorithm 6: Imbalanced Concurrent Shard-by-Shard Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n = 1 (n = Batch Size),

numThreads = {2,3,4,5} (t = Thread Size)

// The function(s) below are called once per value of t.

Output: A Delta table for each thread

```
1 getTableList = getTableNames(Shards);
  // Sorting the tables based on the total size of records present
2 getSortedList = sortBasedOnTableSize(getTableList);
  // Partitioning the table list based on number of threads
3 partitionedTables = getPartitionedTableNames(getSortedList, numThreads);
  // Creating threads and passing the partitioned table list and
     batch size as inputs
4 for j ← 1 to numThreads do
5 |   sequentialShardByShard(partitionedTables[j], n);
6 end
```

4.3 Experiments and Evaluations

We have created 2 datasets for our experimentation and evaluation purposes. The first one has 10 million records and the other has 200 million records. Section 3.2.3 contains more information regarding the creation of datasets.

For all experiments involving a dataset of 10 million, we executed the algorithm for 10 iterations. However, when we started testing on larger datasets, we realized that running the same algorithm 10 times consumed a lot of time, and resulted in significant cloud computing costs. Therefore, in view of cloud budget considerations, we reduced the iterations to 5 for larger datasets. Moreover, for smaller datasets that had been executed 10 times previ-

ously, we adjusted our approach so that we only used the first 5 iterations for visualization rather than all the 10.

4.3.1 Experimental Conditions

In this subsection, we discuss the experimental conditions that we followed for all the strategies in our experimental setup.

1. We restart the Spark container after each experiment to clear cached credentials of the source database.

2. Each experiment is run 5 times, with the average runtime reported.

3. We used hardware with varying memory capacities: specifically, a 32GB m5.2xlarge instance for processing 10 million records and a 64GB m5.4xlarge instance for handling 200 million records. The m5.2xlarge instance type is equipped with 8 vCPUs and is well-suited for moderate computational tasks with its balanced memory-to-CPU ratio. In contrast, the m5.4xlarge instance type offers higher memory capacity and is powered by 16 vCPUs, making it ideal for more memory-intensive computations and larger datasets.

4. The 10 million dataset used shard sizes of 4,6,8, and 10. The 200 million dataset only used a shard size of 50.

4.4 Experimental Results for 10 Million Dataset

In this section, we will look at the various experiments that were performed on the 10-million dataset. Initially, we did not implement garbage collection for evaluating the sequential shard-by-shard strategy and the concurrent shard-by-shard strategy due to the smaller data volume and faster execution. However, when we later started our experiments on the 200-million dataset, our algorithm failed multiple times due to garbage collection issues and memory out-of-bound problems. Therefore, we redid the experiments for the

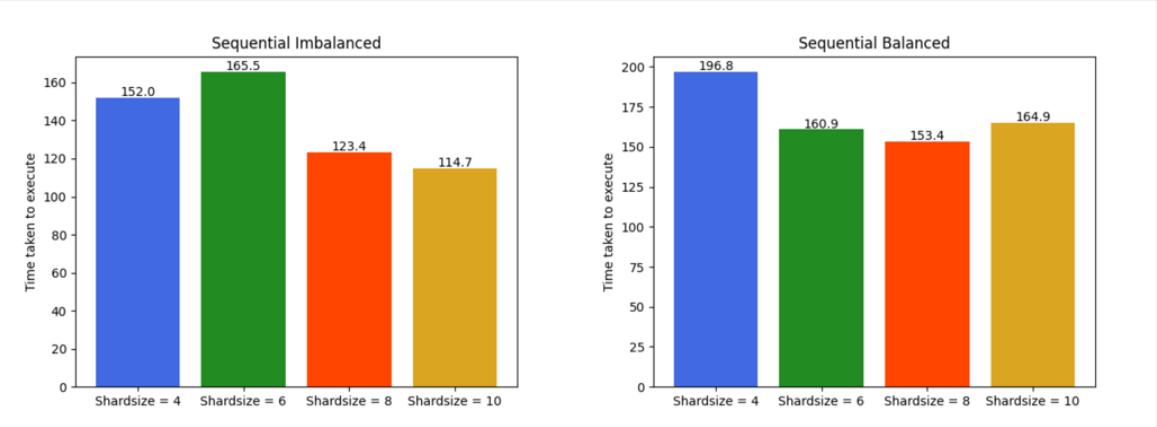

Figure 4.5: Average runtime comparison between Imbalanced and Balanced shards in Sequential Shard-by-Shard Algorithm.

concurrent strategy by implementing garbage collection to achieve predictable algorithm performance. Additionally, for the experiments of the sequential strategy, we also did not perform garbage collection for the 10-million dataset due to its sequential nature.

4.4.1 Sequential Shard-By-Shard Algorithm for Balanced and Imbalanced Shards

Figure 4.5 represents the comparison of both algorithms execution under various shard sizes.

When running the sequential shard-by-shard algorithm on a 32 GB (m5.2xlarge) instance with 32 GB of driver memory, imbalanced shard sizes of 4, 8, and 10 execute faster than balanced shards. This is because imbalanced shards require fewer network calls compared to balanced shards, where consistent network calls are made due to the equal distribution of data across multiple tables. However, the imbalanced type with a shard size of 6 experiences a slight slowdown due to Apache Spark's startup delay. This, resulted in slow execution of the algorithm for all the first 5 iterations.

Similarly, as the shard size increases for the same number of records, we observe an decrease in runtime for both balanced and imbalanced shard types, except for a slight increase in runtime with a shard size of 10. Overall, bringing in data from imbalanced shards is faster than balanced shards due to the reduced number of network calls.

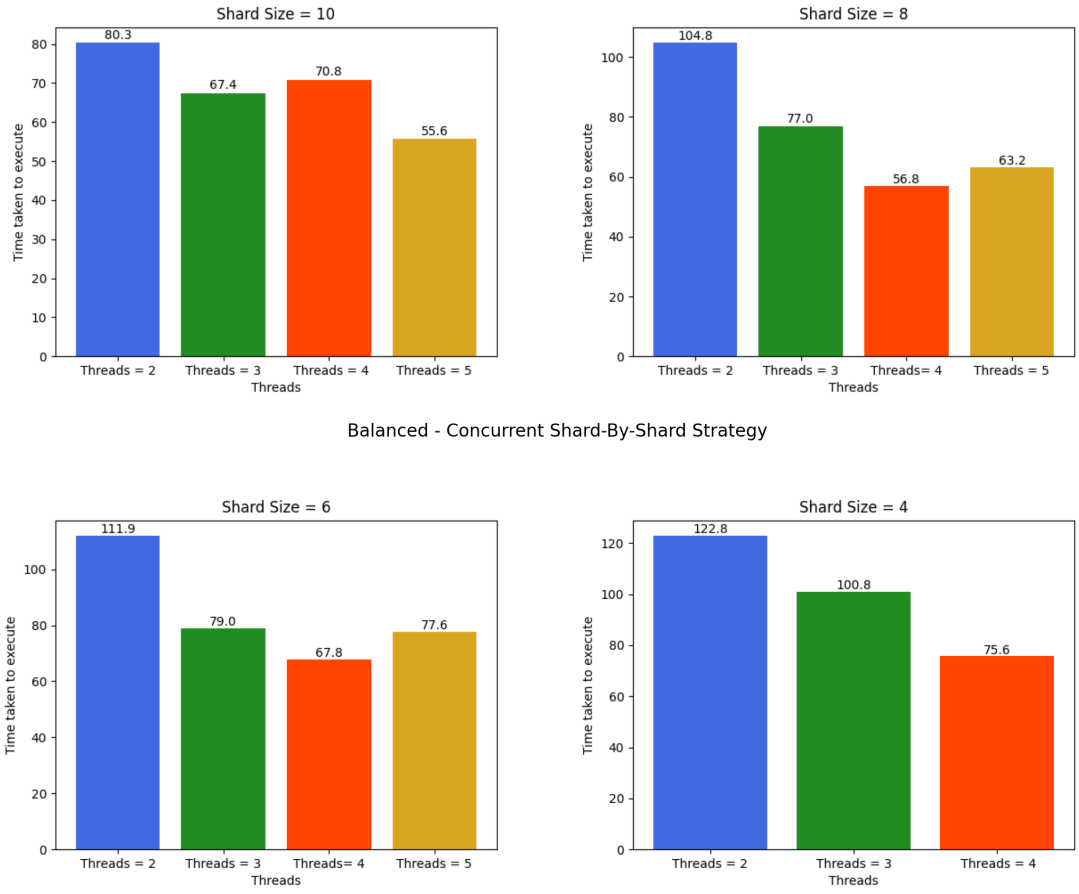

Figure 4.6: Effect of Threads on Balanced Concurrent Shard-by-Shard Algorithm.

4.4.2 Concurrent Shard-By-Shard Algorithm for Balanced and Imbalanced Shards

In concurrent shard-by-shard strategy, we evaluate the algorithm by increasing the number of threads. The parameters we consider for our results are: 1) Effect of Shard Size, 2) Effect of Threads. Figure 4.6 describes the results for concurrent shard-by-shard strategy.

Effect of Shard Size - Balanced Shards Table 4.7 clearly shows that execution time decreases as shard size increases. This is because larger shard sizes lead to lower data volume per shard, making it easier for the algorithms to process data efficiently. The relationship between shard size and data volume can be approximated as shard size being inversely proportional to volume.

Threads	shardsize = 4	shardsize = 6	shardsize = 8	shardsize = 10
2	122.8s	111.9s	104.8s	80.3s
3	100.8s	79.0s	77.0s	67.4s
4	75.6s	67.8s	56.8s	70.8s
5	N/A	77.6s	63.2s	55.6s

Table 4.7: Threads vs Shards Execution Time in Balanced Shards.

$$shardsize \propto \frac{1}{volume}$$

However, for a thread size of 4, a slight increase in execution time is observed for shard size of 10. This is likely due to the slow improvement of algorithm performance that might be caused by network congestion, as the results exhibits that characteristic. Overall, for balanced shards, shard size significantly impacts algorithm performance. This is a crucial factor to consider when triggering production pipelines with large shard sizes.

Effect of Threads - Balanced Shards As per Figure 4.6, we can say that increasing the number of threads generally leads to improved performance, assuming sufficient primary memory is available to process the data.

$$threads \propto \frac{1}{runtime}$$

This is evident in the observed decrease in execution time for a shard size of 4 as the number of threads increases. For shard sizes 8 and 6, we observe a clear decrease in runtime until 4 threads are used. However, beyond 4 threads, there is a slight increase in runtime, likely due to resource contention introduced by additional threads. At 5 threads, the runtime reduction is minimal after each iteration. Similarly, for a shard size of 10, runtime generally decreases with more threads, with the exception for thread size 4.

Effect of Shard Size and Threads - Imbalanced Shards As per Figure 4.7, we can see that shard size 4,8 and 6 has a significant decrease in runtime as the number of threads

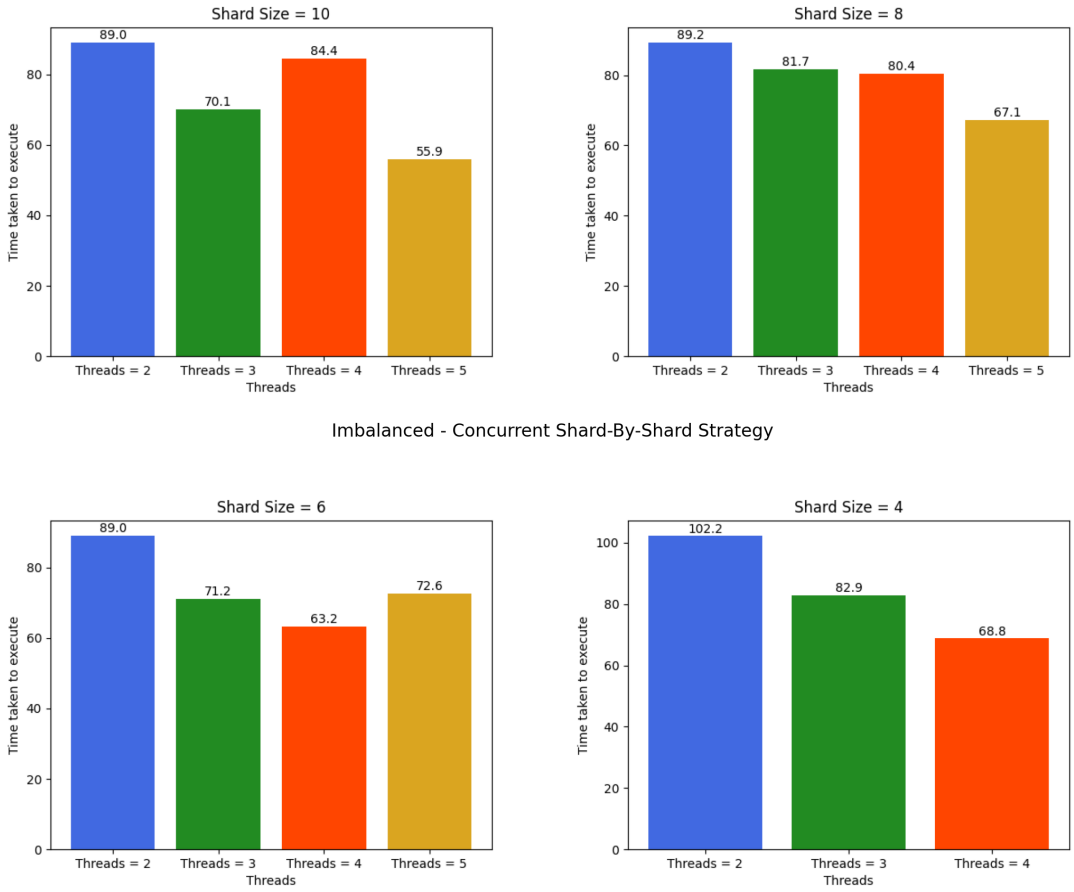

Figure 4.7: Effect of Threads on Imbalanced Concurrent Shard-by-Shard Algorithm.

increases, whereas for shard size 6 we have a slight increase in runtime for thread size 5. Similarly, for shard size 10 as we increase the number of threads linearly the runtime decreases for first 2 threads and threads 5, but for thread size 4 due to startup delay we can see a spike in runtime. Overall, we can observe that as the shard size increases the total runtime for migrating the data decreases due to the fact that volume is distributed across tables in non-uniform way performing network calls to read less volume data from source shards. Also, we can see that as the number of threads increase there is a definite performance improvement over the runtime.

4.5 Experimental Results for 200 Million Dataset

In this section, we are going to look into the experiments for 200 million dataset and we will observe how it performs.

4.5.1 Sequential Shard-By-Shard Algorithm for Balanced and Imbalanced Shards

As per Figure 4.8, when the algorithm is executed for 200 million records, we can see that for balanced shards, execution takes an average of 7370.1 seconds, which is almost 2+ hours to bring the data. Whereas, for imbalanced shards execution takes 4736s (i.e 1.31 hours). The reason for balanced runtime being significantly higher than imbalanced is due to consistent large volume of data across tables and large number of network calls that it performs on the source database. Similarly, for imbalanced shards the tables are sorted based on increasing size. Therefore, we can see that tables with the smallest number of records are migrated faster and the tables with large data takes longer to execute, which results in a lower runtime than balanced shards. Overall we can say that, imbalanced shards lead to 35.7% less runtime than balanced shards for large volumes of data.

4.5.2 Concurrent Shard-By-Shard Algorithm for Balanced and Imbalanced Shards

In this subsection, we will discuss the results for concurrent shard-by-shard technique for balanced and imbalanced shards.

Effect of threads on Imbalanced Shards As per Figure 4.9, when we vary the number of threads and execute the concurrent shard-by-shard algorithm on imbalanced shards. We can observe that the execution time steadily decreases between thread size 3 and thread size 6. Also from thread size 3 to thread size 6, we observe that there is 41.9% reduction in execution time. Which implies that there is a strong correlation in creating the threads and reduction in runtime. Moreover, we can conclude that as the threads increases the time taken to make data available becomes faster.

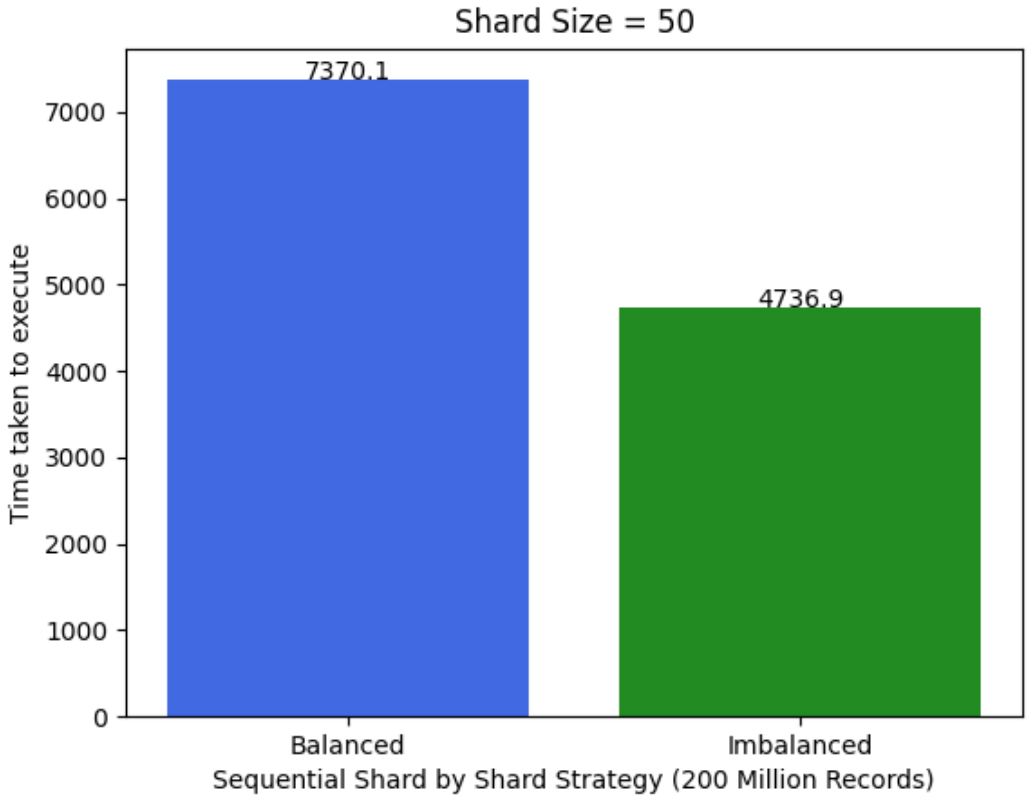

Figure 4.8: Runtime comparison between Imbalanced and Balanced Shards in Sequential Shard-by-Shard Algorithm.

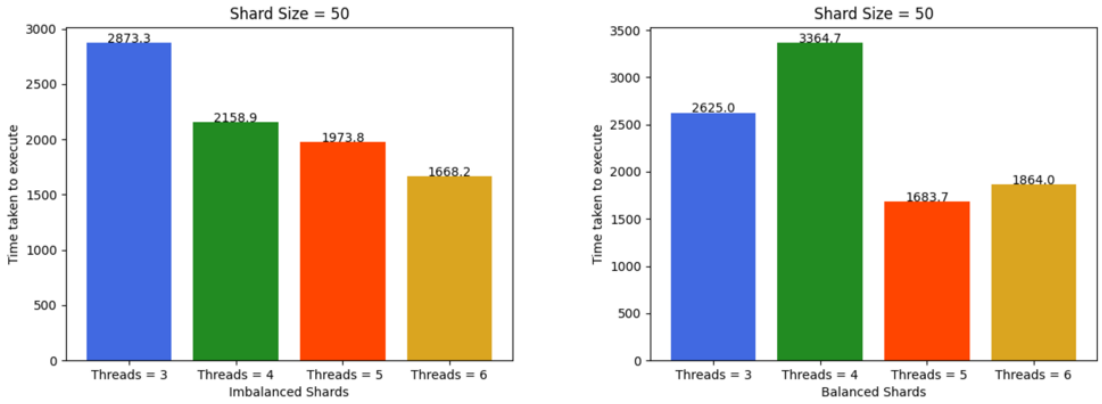

Figure 4.9: Effect of Threads on Imbalanced and Balanced Concurrent Shard-by-Shard Algorithm.

Effect of threads on Balanced Shards As per Figure 4.9, in balanced shards, as the number of threads increases, we observe the same pattern of reduction in time for thread sizes 3, 5, and 6. However, at a thread size of 4, there is a spike in runtime, and this is due to network congestion in the AWS cloud. During our experiments, we observed that the algorithm took more time than usual, and the execution time doubled after every run. However, after a couple of runs, the algorithm returned to normal execution times.

4.6 Summary

In this chapter, we have seen the experiments regarding the shard-by-shard strategy and have understood how it performs under a 10 million and 200 million dataset. In the next chapter, we will look into aggregated shards based algorithm.

Chapter 5

Aggregated-shards-based Strategy

In the previous chapter we understood the Shard-by-Shard algorithm. Now in this chapter, we will examine another strategy to check if the new algorithm enables faster extraction of data into cloud object stores. The strategy we are going to look at in this chapter is an extended version of first strategy. Additionally, the resource planning in this technique requires decent amount of understanding of first algorithm.

5.1 Introduction

In Chapter 4, for the Shard-by-Shard strategy, the base constraint is always n = 1. This means we only read one table into memory per given thread. Even when we increase the number of threads in concurrent strategy for Shard-by-Shard algorithm, each thread only processes one table. Therefore, in this chapter we will observe how aggregated shards based algorithm works. Moreover, we will evaluate its performance using different parameters by implementing algorithm using Apache Spark.

5.2 Aggregated Shards Based Algorithm

The term ”aggregated shards” comes from the operation of processing multiple shards into memory at a given iteration. In the aggregated shards based algorithm, we read multiple tables at once and will write them as a delta table. Therefore, the base constraint of the aggregated shards based algorithm is $n \geq 2$, where n is the batch size. This means at any given time, the algorithm makes network calls to the source sharded database, to read at

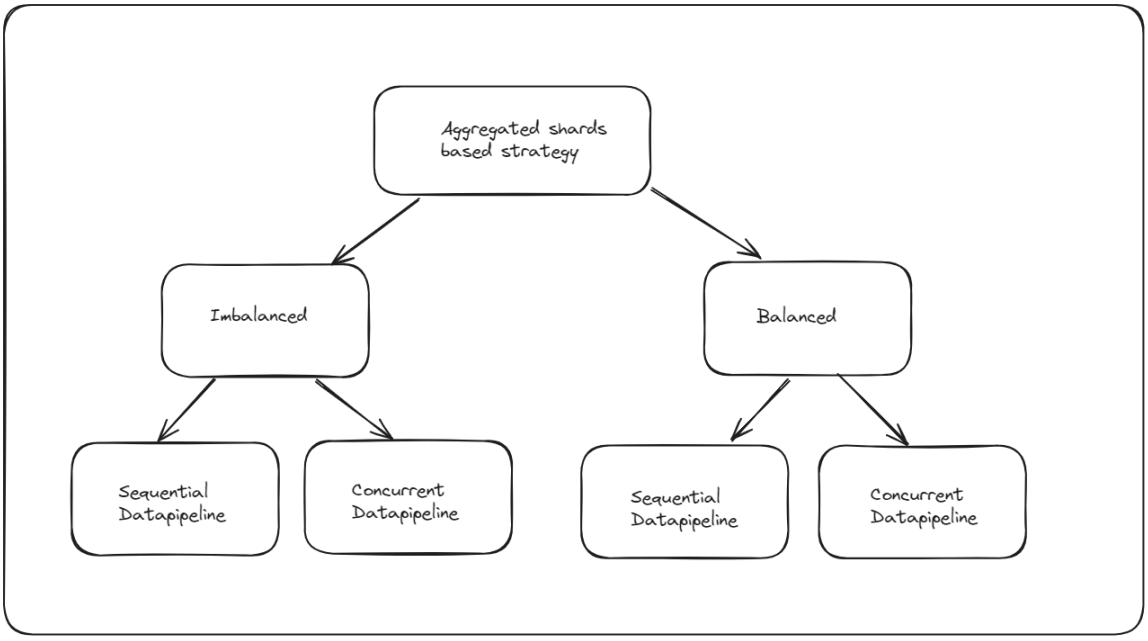

Figure 5.1: Tree Representation of Aggregated Shards Based Strategies.

least 2 tables into memory for processing data.

Just like multiple Shard-by-Shard algorithms are proposed, several aggregated shards-based algorithm are also proposed that consider sequential and concurrent execution for balanced and imbalanced shards. In this chapter, we will evaluate our algorithms considering these scenarios. Moreover, we will consider resource planning methods and their associated calculations. Figure 5.1 depicts the classification of algorithms using a tree diagram.

5.2.1 Sequential Aggregated Shards Based Strategy - Balanced Shards

The first sequential aggregated shards based algorithm is for balanced shards. In sequential processing, we migrate the tables in batch mode using a single thread. While performing the read operation, we read at least 2 tables into memory and write the data in delta table format. Later, we continue this process until all shards are migrated.

The inputs provided to Algorithm 7 are the table names of the shards and the batch size ($n \geq 2$). Additionally, we can determine the iterations using following formula for our

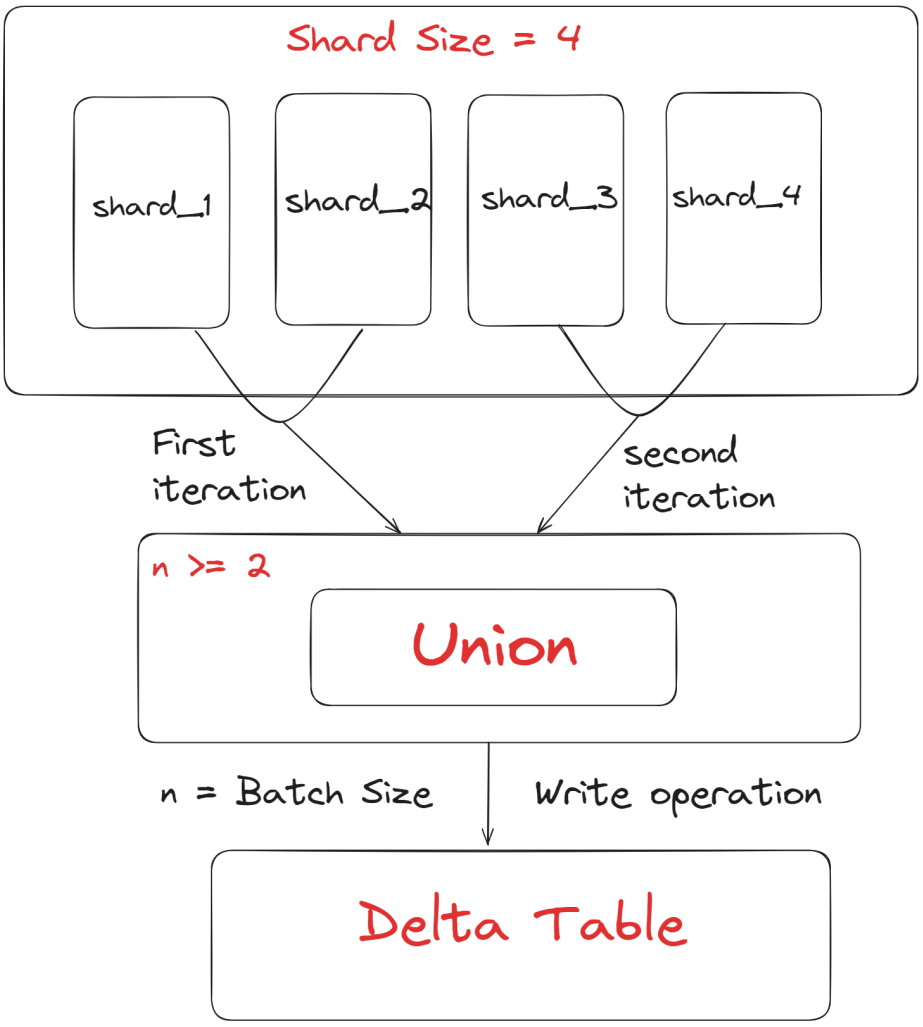

Figure 5.2: Illustration of Aggregated Shards Based Strategy For Shard Size = 4.

estimation purposes.

$$numberOfIterations = \left\lceil \frac{shardSize}{batchSize} \right\rceil$$

5.2.2 Running Example of Sequential Aggregated Shards Based Strategy

We demonstrate the aggregated shards-based strategy using an example of a shard size of 4 with a batch size of 2. Figure 5.2 visually depicts the process of migrating data from the sharded database. Let us assume the total records present in the sharded database amount to 12 records, with each shard containing 3 records.

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
4	2024-07-01 12:10:00	2024-07-01 12:35:00	3	4.5
8	2024-07-01 18:45:00	2024-07-01 19:10:00	3	6.2
12	2024-07-02 08:30:00	2024-07-02 08:50:00	2	4.8
1	2024-07-01 08:15:00	2024-07-01 08:30:00	1	3.2
5	2024-07-01 14:00:00	2024-07-01 14:20:00	4	6.0
9	2024-07-01 20:20:00	2024-07-01 20:35:00	1	2.0

Table 5.1: Result of Union operation in iteration 1.

As a first step, we allocate the required amount of driver memory to perform the read operation. After submitting the PySpark application to the engine, it will convert the Python API to Spark objects and generate a DAG (directed acyclic graph) of operations. For a shard size of 4 with a batch size of 2, we can estimate that there will be 2 iterations.

Iteration 1 In Iteration 1, as per the defined algorithmic constraint the algorithm for sequential aggregated shards generates optimized SQL queries, and requests the database to read 2 tables into memory. Then the db performs a table scan and will retrieve the data for $shard_1$ and stores the queried data of 3 records as a dataframe. Next, we append $shard_2$ result of 3 records with the existing dataframe into primary memory by performing a union operation. Table 5.1 depicts the result of the union operation after iteration 1. Once the batching process is completed, we write the 6 records in the dataframe as a delta table to the cloud object store.

$$unionResult := Shard_1 \cup^5 Shard_2$$

$$DeltaTable_1 := unionResult$$

Iteration 2 In Iteration 2, the spark application regenerates similar DAG of operations to read the remaining shards from the database. Therefore, it will read the records from

[5] $\cup = UnionOperation$

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
4	2024-07-01 12:10:00	2024-07-01 12:35:00	3	4.5
8	2024-07-01 18:45:00	2024-07-01 19:10:00	3	6.2
12	2024-07-02 08:30:00	2024-07-02 08:50:00	2	4.8
1	2024-07-01 08:15:00	2024-07-01 08:30:00	1	3.2
5	2024-07-01 14:00:00	2024-07-01 14:20:00	4	6.0
9	2024-07-01 20:20:00	2024-07-01 20:35:00	1	2.0
2	2024-07-01 09:00:00	2024-07-01 09:25:00	2	5.1
6	2024-07-01 15:30:00	2024-07-01 15:50:00	1	3.8
10	2024-07-01 22:00:00	2024-07-01 22:20:00	2	4.0
3	2024-07-01 10:45:00	2024-07-01 11:00:00	1	2.7
7	2024-07-01 17:00:00	2024-07-01 17:30:00	2	7.5
11	2024-07-02 07:00:00	2024-07-02 07:20:00	1	3.5

Table 5.2: Result of Union operation in iteration 2.

$shard_3$ and $shard_4$ into memory by using a union operation. Table 5.2 depicts the result of the union operation of iteration 2. Later, it will append the data to the previously created $DeltaTable_1$.

$$unionResult := Shard_3 \cup Shard_4$$

$$DeltaTable_2 := DeltaTable_1 +\!+ {}^{6} unionResult$$

[6] $+\!+ = AppendOperation$

Algorithm 7: sequentialAggShards(Shards, n)

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n // Batch Size

Output: A Single Delta table

```
1  i = 0;
2  masterDataFrame = createDataFrame(tableNames[0]);
3  while length(Shards)>0 do
4      i ← i + 1;
       // Clearing the masterDataFrame for subsequent iterations
5      masterDataFrame = masterDataFrame.clear();
       // Reading shards from the sharded database as dataframe and
          performing union operation based on batch size
6      for J ← 1 to n do
7          shardName = Shards.dequeue();
8          dataFrame = dataFrame.read(shardName);
9          masterDataFrame = masterDataFrame.union(dataFrame);
10     end
       // Creating Base Delta Table
11     if i = 1 then
12         DeltaTable =: masterDataFrame;
13     end
       // Appending the masterDataFrame result to the existing
          DeltaTable
14     else
15         DeltaTable =: DeltaTable.append(masterDataFrame);
16     end
17 end
```

Algorithm 8: Sequential Aggregated Shards Based Strategy

Input : $Shards = \{Shard_1, Shard_2, \ldots, Shard_k\}$,

$n = \{2,3,4,5\}$ *(n = Batch Size)*

// The function(s) below are called once per value of n.

Output: A Single Delta table

```
1 tableNames = getTableNames(Shards);
  // Invocating sequentialAggregatedShards algorithm
2 sequentialAggShards(tableNames, n);
```

5.2.3 Sorting Based Sequential Aggregated Shards Based Strategy - Imbalanced Shards

In the second approach for handling imbalanced shards in the sequential aggregated shards-based strategy, we first sort the shards based on their size and then store the table list in a data structure. Later, the table list is passed as a function call to the Algorithm 7. This is the main difference between the Algorithm 8 and the current one.

Algorithm 9: Sorting Based Sequential Aggregated Shards Based Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

$n = \{2,3,4,5\}$ *(n = Batch Size)*

// The function(s) below are called once per value of n.

Output: A Single Delta table

```
  // Sorting shards based on the total row size in ascending order
1 getSortedList = sortBasedOnTableSize(shards);
  // Invocating sequentialAggregatedShards algorithm
2 sequentialAggShards(getSortedList, n);
```

5.2.4 Running Example of Sorting Based Sequential Aggregated Shards

Let us assume we have a total of 12 records distributed across 4 shards: 5 records for $shard_1$, 2 records for $shard_2$, 4 records for $shard_3$ and 1 record for $shard_4$. As per the

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
2	2024-07-01 09:00:00	2024-07-01 09:30:00	1	2.7
3	2024-07-01 10:00:00	2024-07-01 10:45:00	3	5.3
1	2024-07-01 08:00:00	2024-07-01 08:30:00	2	4.5

Table 5.3: Results of Union operation in iteration 1.

Figure 5.3, the first step depicts sorting of tables in ascending order and storing the order in a data structure. After sorting based on row size, the table list becomes $shard_4$, $shard_2$, $shard_3$, and $shard_1$. For the illustration of this example, we use batch size of 2 with a shard size of 4. Just like the iterations estimated for balanced shards, even for imbalanced shards, we can determine the number of iterations to be 2.

Therefore, in iteration 1, we read the first 2 tables from the sorted list ($shard_4$,$shard_2$) and store them as a dataframe in memory using the union operation. After, the union operation the dataframe holds 3 records. Table 5.3 depicts the output of the above operation. Next, we write the dataframe as a delta table.

$$unionResult := Shard_4 \cup^7 Shard_2$$

$$DeltaTable_1 := unionResult$$

Following this, we process the remaining shards by reading 9 records from ($shard_3$, $shard_1$). Table 5.4 depicts the output of the union operation. Subsequently, we append this data to the existing delta table, creating a significantly large table for analysis.

$$unionResult := Shard_3 \cup Shard_1$$

[7] $\cup = UnionOperation$

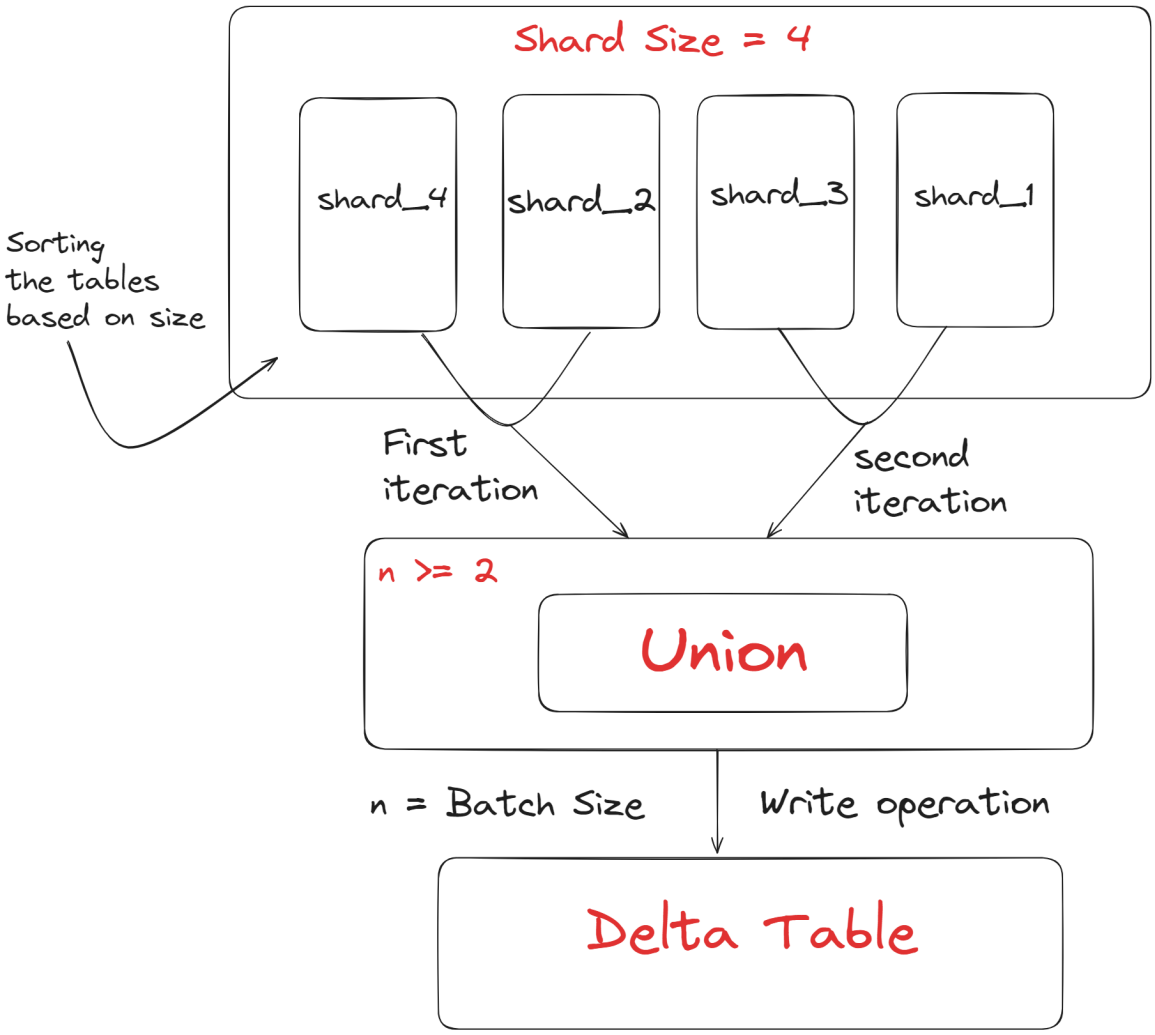

Figure 5.3: Illustrating Imbalanced Aggregated Shards Based Strategy For Shard Size = 4.

$$DeltaTable_2 := DeltaTable_1 \mathbin{+\!\!+}^{8} unionResult$$

5.2.5 Concurrent Aggregated Shards Based Strategy for Balanced Shards

Our third algorithm in this chapter is a concurrent aggregated shards-based strategy for balanced shards. In this strategy, at any given point, each thread processes at least 2 tables into memory, with multiple threads running concurrently. Let us assume that if our process has 3 threads with a batch size of 2, it means we are reading 6 tables concurrently into

[8] $+\!\!+ = AppendOperation$

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
8	2024-07-02 08:00:00	2024-07-02 08:45:00	2	4.8
9	2024-07-02 09:30:00	2024-07-02 10:15:00	1	3.1
10	2024-07-02 11:00:00	2024-07-02 11:30:00	4	7.5
11	2024-07-02 13:00:00	2024-07-02 13:45:00	3	6.4
12	2024-07-02 14:30:00	2024-07-02 15:00:00	2	4.7
4	2024-07-01 12:00:00	2024-07-01 12:45:00	3	6.2
5	2024-07-01 13:30:00	2024-07-01 14:15:00	2	5.1
6	2024-07-01 15:00:00	2024-07-01 15:30:00	1	3.8
7	2024-07-01 16:00:00	2024-07-01 16:30:00	4	7.0

Table 5.4: Results of Union operation in iteration 2.

memory. In contrast, for the same algorithm with a single thread in 5.2.1 we can only scale with batch size. Therefore, in this strategy, we examine various experiments by scaling both the thread size and batch size of the algorithm. Overall, this algorithm is an advanced version of our single-thread strategy described earlier in 5.2.1 except for the fact that our current one utilizes multiple threads.

The inputs for the algorithm are batch size n = {2,3,4,5} and numThreads = {2,3,4,5} shardSize = {4,6,8,10}

Running Example of Concurrent Aggregated Shards Based Strategy:

Figure 5.4 visually depicts the process of the concurrent aggregated shards-based strategy. In this subsection, we will explore a running example with a shard size of 6, thread size of 2, and batch size of 2.

Similar to the previous examples, let's consider a shard size of 6, with each shard containing 2 records, summing up to a total of 12 records.

As the first step, when we submit the Apache Spark application, the program partitions the table list based on the number of threads and generates two lists: ($shard_2$, $shard_4$, $shard_6$) and ($shard_1$, $shard_3$, $shard_5$).

In the next step, both threads are created, and the partitions are passed to them. Once

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
8	2024-07-02 08:00:00	2024-07-02 08:45:00	2	4.8
9	2024-07-02 09:30:00	2024-07-02 10:15:00	1	3.1
10	2024-07-02 11:00:00	2024-07-02 11:30:00	4	7.5
11	2024-07-02 13:00:00	2024-07-02 13:45:00	3	6.4

Table 5.5: Delta table result for $thread_1$ after iteration 1.

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
12	2024-07-02 14:30:00	2024-07-02 15:00:00	2	4.7
4	2024-07-01 12:00:00	2024-07-01 12:45:00	3	6.2
5	2024-07-01 13:30:00	2024-07-01 14:15:00	2	5.1
6	2024-07-01 15:00:00	2024-07-01 15:30:00	1	3.8

Table 5.6: Delta table result for $thread_2$ after iteration 1.

the threads start executing, each thread fetches data in parallel based on the batch size, loading 4 shards into memory using 2 threads. After reading these 4 shards—2 from each partition—we perform a union operation within each thread and create a delta table to write the data. Tables 5.5 and 5.6 depict the result of each thread. Overall, after iteration 1 for process $thread_1$ and $thread_2$ we have a total of 8 records with each delta table having 4 records.

Thread 1	Thread 2
$uROT_1 := Shard_2 \cup Shard_4$	$uROT_2 := Shard_3 \cup Shard_1$
$DeltaTable_1 := uROT_1$	$DeltaTable_2 := uROT_2$

Later, in iteration 2, the process reads the remaining tables from both partitions and it appends two records from $thread_1$ and two records from $thread_2$, to their respective delta tables forming a total of 12 records with each delta table having 6 records. Tables 5.7 and 5.8 depict the result of the delta tables for $thread_1$ and $thread_2$.

[7] $\text{uROT}_1 = Unionresultof\,\text{thread}_1$
[8] $\text{uROT}_2 = Unionresultof\,\text{thread}_2$

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
8	2024-07-02 08:00:00	2024-07-02 08:45:00	2	4.8
9	2024-07-02 09:30:00	2024-07-02 10:15:00	1	3.1
10	2024-07-02 11:00:00	2024-07-02 11:30:00	4	7.5
11	2024-07-02 13:00:00	2024-07-02 13:45:00	3	6.4
1	2024-07-01 08:00:00	2024-07-01 08:30:00	2	4.5
2	2024-07-01 09:00:00	2024-07-01 09:30:00	1	2.7

Table 5.7: Delta table result for $thread_1$ after iteration 2.

trip_id	pickup_datetime	dropoff_datetime	passenger_count	trip_distance
12	2024-07-02 14:30:00	2024-07-02 15:00:00	2	4.7
4	2024-07-01 12:00:00	2024-07-01 12:45:00	3	6.2
5	2024-07-01 13:30:00	2024-07-01 14:15:00	2	5.1
6	2024-07-01 15:00:00	2024-07-01 15:30:00	1	3.8
3	2024-07-01 10:00:00	2024-07-01 10:45:00	3	5.3
7	2024-07-01 16:00:00	2024-07-01 16:30:00	4	7.0

Table 5.8: Delta table result for $thread_2$ after iteration 2.

Thread 1	Thread 2
$uROT_1 := Shard_6$	$uROT_2 := Shard_5$
$DeltaTable_1 := DeltaTable_1 +\!\!+ uROT_1$	$DeltaTable_2 := DeltaTable_2 +\!\!+ uROT_2$

This process allows the algorithm to efficiently process and write data to the delta tables based on the shard size, thread size, and batch size.

Resource planning for balanced shards of 10 million dataset

To efficiently plan the resources for balanced shards, we follow the same principles that we applied in Shard-by-Shard Algorithm. In Shard-by-Shard algorithm, where the $batchsize = 1$ and $threadsize \geq 2$, to estimate resources we used to fetch the maximum table size and simply multiply it by the number of threads. We used the below formula to calculate the memory:

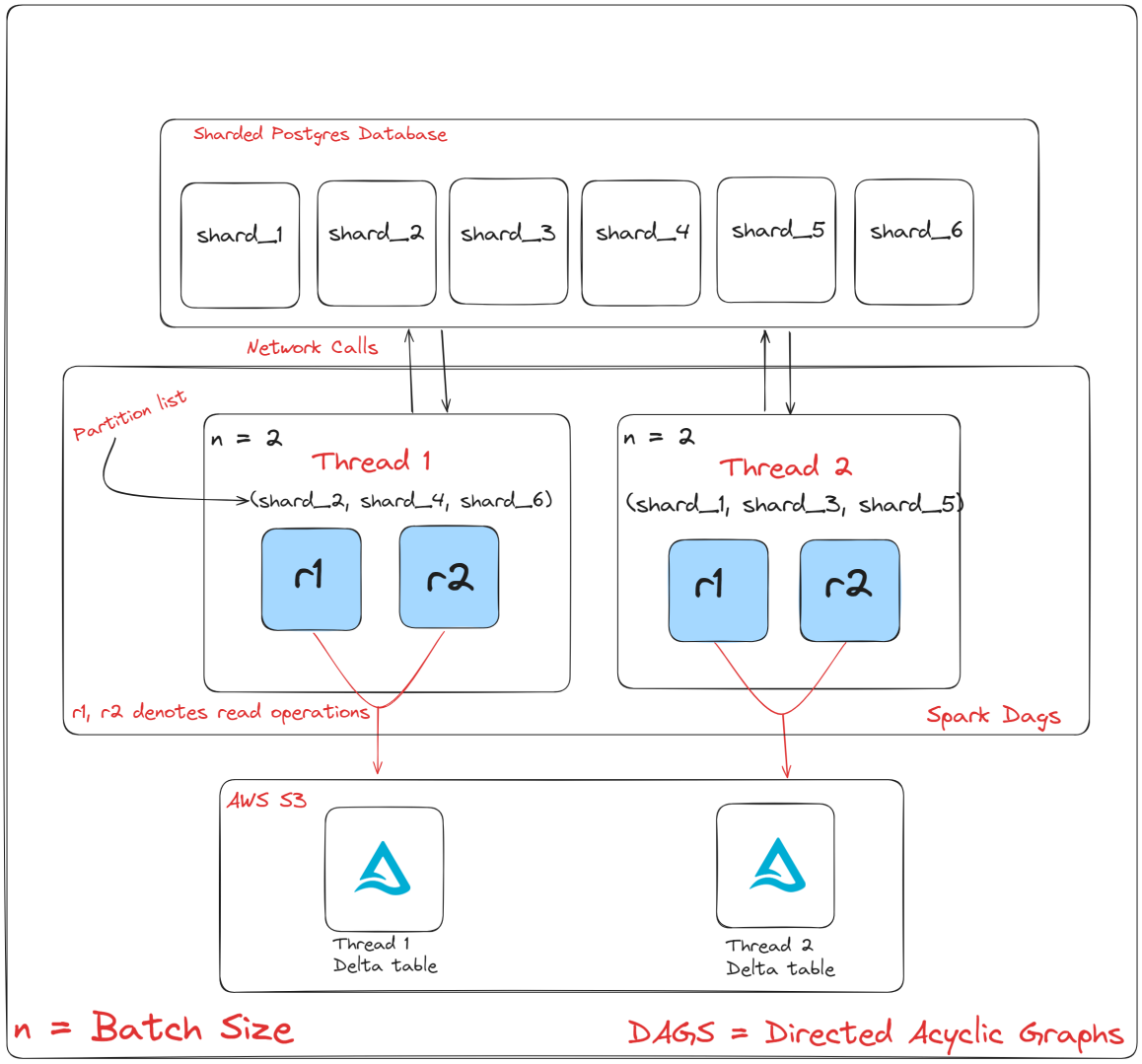

Figure 5.4: Concurrent Aggregated Shards Based Algorithm for Thread Size 2 and Batch Size 2.

$$memoryForThreads = (highestTableSize * numberofThreads)$$

Similarly, in this approach, we first fetch the table size and then calculate the memory required for a given batch. Later, we multiply it by the number of threads. The reason of multiplying by batch size in this technique is that each thread in the algorithm processes at least 2 tables into memory. Therefore, we must ensure that the primary memory is sufficient to process each table from an individual thread, and we consider this resulting value as the minimum threshold for our driver process. Additionally, the formula below is used to determine the memory required for a given shard:

$$memoryForThreads = ((highestTableSize * batchSize) * numberofThreads)$$

Specifically, for the scope of introducing the concept we considered thread size = {2,3,4} and batch size = 2. Table 5.9 presents the relationship between shard size, size of each shard, driver memory required (DM) for a batch size of 2, and the total memory required for different thread counts *(t)* for a 10 million dataset. For a shard size of 4, each shard is 573 MB, and the memory required for a batch size of 2 is a minimum of 1146 MB. When using 2 threads, the total memory required is at least 2292 MB; for 3 threads, it increases to at least 3438 MB, and for 4 threads, it rises to at least 4584 MB. Similarly, with a shard size of 6 and each shard being 381 MB, the memory required for a batch size of 2 is at least 762 MB. The memory requirements are at least 1524 MB for 2 threads, at least 2286 MB for 3 threads, and at least 3048 MB for 4 threads. For a shard size of 8, with each shard being 285 MB, the memory required for a batch size of 2 is at least 570 MB. The required memory increases to at least 1140 MB for 2 threads, at least 1710 MB for 3 threads, and at least 2280 MB for 4 threads. Lastly, with a shard size of 10 and each shard being 228 MB, the memory required for a batch size of 2 is at least 456 MB. The total memory needed is at least 912 MB for 2 threads, at least 1368 MB for 3 threads, and at least 1824 MB

Shard Size	Size of Shard	DM for batchSize =2	t = 2	t = 3	t = 4
4	573 MB	1146 MB	>2292 MB	>3438 MB	>4584 MB
6	381 MB	762 MB	>1524 MB	>2286 MB	>3048 MB
8	285 MB	570 MB	>1140 MB	>1710 MB	>2280 MB
10	228 MB	456 MB	>912 MB	>1368 MB	>1,824 MB

Table 5.9: Threads vs Primary Memory Required for Balanced Concurrent Aggregated Shards.

Shard Size	t = 2	t = 3	t = 4
4,6,8,10	2	3	4

Table 5.10: Shard Size vs Threads output in Delta tables for Concurrent Aggregated Shards Based Strategy.

for 4 threads. This table provides a clear view of how memory requirements scale with increasing shard sizes and thread counts for a 10 million dataset.

After processing the data, each thread creates one delta table on S3. Therefore, the table 5.10 gives us an idea of the resulting delta tables that are created on the cloud for their respective threads.

Algorithm 10: Balanced Concurrent Aggregated Shards Based Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n = {2,3,4,5}, (n = Batch Size)

// The function(s) below are called once per value of n.

numThreads = {2,3,4,5} (t = Thread Size)

// The function(s) below are called once per value of t.

Output: A Delta table for each thread

```
// Partitioning tables
partitionedTablesList = getPartitionedTableNames(Shards, numThreads);
// Creating threads and passing the partitioned table list
for j ← 1 to numThreads do
    sequentialAggShards(partitionedTablesList[j], n);
end
```

Shard Size	Maximum Table Size	t = 2	t = 3	t = 4
4	556 MB	>1112 MB	>1668 MB	>2224 MB
6	356 MB	>712 MB	>1068 MB	>1424 MB
8	420 MB	>840 MB	>1260 MB	>1680 MB
10	348 MB	>696 MB	>1044 MB	>1392 MB

Table 5.11: Threads vs Primary Memory Required for Concurrent Aggregated Shards based strategy for Imbalanced Shards.

5.2.6 Sorting Based Concurrent Aggregated Shards Based Strategy - Imbalanced Shards

Our fourth algorithm in this chapter refers to Algorithm 11 and it is sorting based concurrent aggregated shards based strategy. We apply this algorithm for imbalanced shards. In this technique, we first sort the tables based on their sizes and then execute the algorithm using multiple threads to evaluate performance. The difference between this strategy and the previous Algorithm 10 lies in sorting the shards based on their sizes, whereas in balanced shards all shard sizes are the same.

Resource planning for imbalanced shards of 10 million dataset

Table 5.11 illustrates the estimated driver memory consumption for various shard sizes and different numbers of threads *(t)* in our algorithm. The maximum table size represents the largest individual table size within each shard.

The figures in the table represent the minimum estimated driver memory requirements for each shard size and thread count combination. For instance, for a shard size of 4 and t = 2 threads, the driver should allocate more than 1112 MB of memory to handle the processing. Similarly, as the number of threads increases (to 3 or 4 threads), the required memory allocation also increases proportionally to accommodate concurrent data processing from multiple tables within each shard.

This estimation is crucial for resource planning to ensure optimal performance of the algorithm, taking into account the variability in table sizes and the concurrency introduced by multiple threads.

Algorithm 11: Imbalanced Concurrent Aggregated Shards Based Strategy

Input : Shards = $\{Shard_1, Shard_2, \ldots, Shard_k\}$,

n = {2,3,4,5} *(n = Batch Size),*

// The function(s) below are called once per value of n.

numThreads = {2,3,4,5} *(t = Thread Size)*

// The function(s) below are called once per value of t.

Output: A Delta table for each thread

```
// Sorting shards based on the total row size in ascending order
getSortedList = sortBasedOnTableSize(Shards);
// Partitioning the table list using partition function
partitionedTablesList = getPartitionedTableNames(getSortedList, numThreads);
// Creating threads and passing the partitioned table list
for j ← 1 to numThreads do
    sequentialAggShards(partitionedTablesList[j], n);
end
```

5.3 Experimental Conditions and Evaluations Performed

To conduct our experiments, we utilized memory-intensive instances capable of handling various types of data loads. As outlined in Chapter 4, our sharded database comprises two datasets: one with 10 million records and another with 200 million records.

For the 10 million record dataset, we employed an m5.2xlarge instance that has 32 GB of memory and 8 vCPUs, which performs moderately well for data-intensive tasks. Similarly, to accommodate the larger dataset of 200 million records, we vertically scaled our primary memory and utilized an m5.4xlarge instance with 64 GB of memory and 16 vCPUs, and an m5.8xlarge instance with 128 GB of memory and 32 vCPUs to process the large-scale data.

Moreover, we restarted the Apache Spark container after each experiment for larger

datasets, conducting each experiment five times. For the 10 million record dataset, we executed the algorithm ten times initially but used only the first five results to report the average runtime. This approach helped optimize the cloud bill while performing experiments with larger datasets.

5.4 Experimental Results for 10 Million Dataset

In this section, we present the experiments conducted on the 10 million record dataset by varying the types of shards and the amount of driver memory. Our objective is to assess the impact of shards on driver memory and how driver memory affects batch size.

5.4.1 Sequential Aggregated Shards Based Algorithm

For the sequential aggregated shards-based algorithm, we execute and evaluate the strategy on the parameters: 1) Effect of Shard Size on Driver Memory, and 2) Effect of Driver Memory on Batch Sizes.

Effect of Shard Size on Driver Memory In this category, we performed various experiments on shard sizes 4,6,8, and 10 by varying the driver memory from 32 GB to 4 GB, halving it each time. However, due to the page limitations in our thesis, we will focus on comparing the results obtained with maximum and minimum driver memory settings, to establish if there is a dependency on driver memory with shard sizes. So, we will discuss the results of experiments for 32 GB and 4 GB.

Balanced Shards Figure 5.5 depicts the execution time for a driver memory of 32 GB and 4 GB respectively. When we attempt to reserve any amount of driver memory in Spark, the engine defaults to 50% to 60% from the total requested driver memory. Therefore, for 32 GB driver memory the process allocates between 16 to 17 GB for the first experiment, while for 4 GB it allocates between 2 GB to 2.2 GB of primary memory. However, the resulting driver memory from the first experiment after allocation for this dataset is excessive, as the

entire dataset is 1.8 GB. For the second experiment we only have up to 2.2 GB of primary memory, which does effect loading of data into RDDs.

When comparing the execution outputs for a shard size of 4 using 32 GB of driver memory to those using 4 GB of driver memory, we observe that the algorithm does not execute for any batch size due to out-of-memory issues. Therefore, we do not see a plot for that experiment. However, it begins to process data with a shard size of 6 for a batch size of 2. It also fails at the remaining batch sizes due to out-of-memory issues. Similarly, shard sizes 8 and 10 execute for batch sizes 2 and 3, and 2, 3, and 4, respectively, with a linearly increasing runtime. This is because the process has limited driver memory.

In the same way, as the shard size increases, the execution time decreases in 32 GB driver memory. When the shard size increases, the volume of the shards decreases, and the driver memory required to process the data also decreases. To illustrate this point, we can observe the execution time for shard size of 8. In the 4 GB output with a shard size of 8, the time taken for batch size 3 is greater when compared to that with 32 GB. This is because the allocated driver memory is insufficient to process the source shards.

Therefore, after examining the above results we can conclude that appropriate driver memory is required to make the algorithms run faster.

Imbalanced Shards In the case of imbalanced shards, we conducted the same set of experiments on shard sizes 4, 6, 8, and 10 with the previously mentioned driver memory configurations.

For the 32 GB experiment, with a shard size of 4, we observed that all batch sizes execute without any issues. Furthermore, as the shard size increases, there was a noticeable improvement in performance.

In contrast, with 4 GB of driver memory, the same algorithm failed to execute even for a batch size of 2 due to out-of-memory errors. This is why in Figure 5.6, there is no data plotted for the first half. Consequently, in the 4 GB result set for shard sizes 6, 8, and 10, we observed that the runtime increased for each shard size, as the driver memory was limited

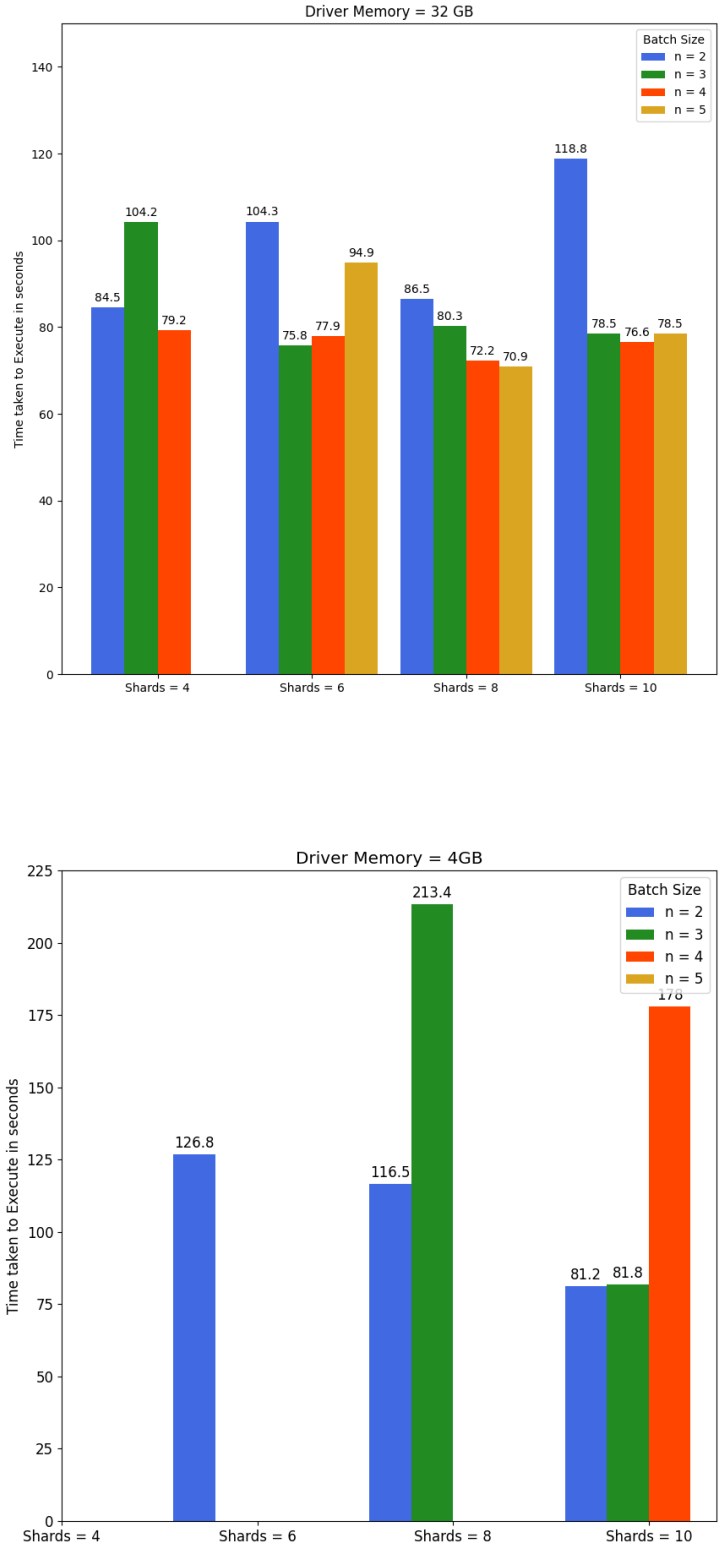

Figure 5.5: Effect of Shard Size on 32 GB and 4 GB Driver Memory in Balanced Shards.

and the shards were imbalanced.

Therefore, we can conclude that shard size is inversely dependent on driver memory to process the source data efficiently.

Effect of Driver Memory on Batch Sizes In this section, we observe and report the status of experiments regarding whether driver memory has any impact on the batch sizes of the algorithm. To test the above point, we have conducted experiments using driver memory of 4 GB, 8 GB, 16 GB, and 32 GB on shard sizes of 4 and 10.

Balanced Shards For balanced shards, we performed our tests on shard sizes of 4 and 10. Firstly, as shown in Figure 5.7, we will discuss the results for a shard size of 4 with 4 GB of driver memory. In our first set of experiments with 4 GB of driver memory on batch sizes 2, 3, and 4, the algorithm could not perform the migration of data. This is why for 4 GB, there is no bar plot associated with any batch size.

However, with 8 GB of driver memory, the algorithm did execute for batch sizes 2 and 3, with average execution times of 114.1 seconds for batch size 2 and 176.1 seconds for batch size 3. In batch size 4, it failed due to insufficient driver memory. As we increase the batch size, we process more shards into memory and require more resources to accommodate those shards. This is why 8 GB of driver memory did not suffice for batch size 4.

Similarly, with 16 GB and 32 GB of driver memory, the algorithm executed smoothly without any memory issues due to the ample amount of primary memory. As shown in Figure 5.7, with 16 GB and 32 GB, as the batch size increases, the runtime should also decrease accordingly. However, as we are performing experiments in a cloud environment, uneven startup times caused a slight increase in runtime for batch size 3.

When using a shard size of 10 in Figure 5.7 with a 10-million record dataset, the volume in each shard decreases. Therefore, for a driver memory of 4 GB, as the batch size increases, the time taken to execute also increases, and it eventually fails due to out-of-memory issues. This occurs because as the batch size increases, the number of shards queried from the

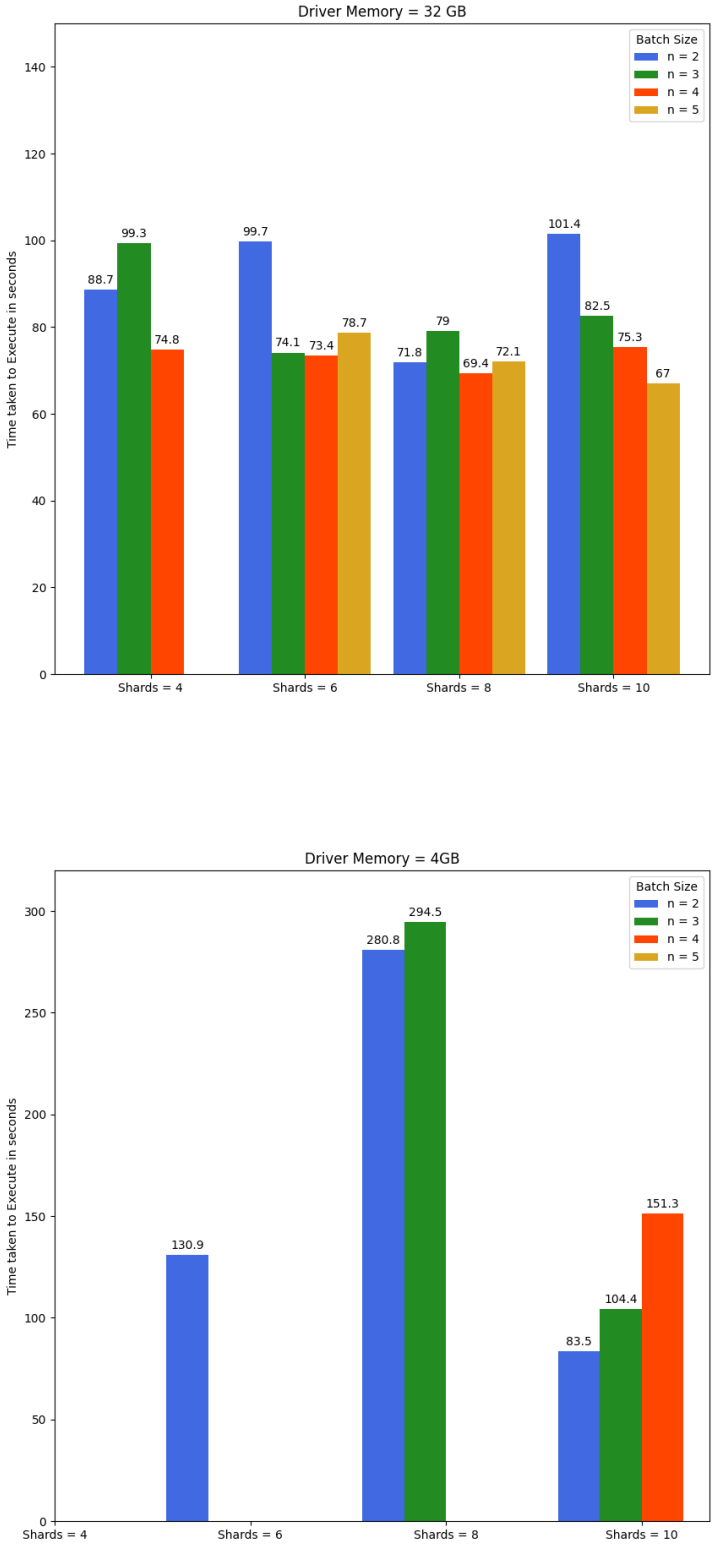

Figure 5.6: Effect of Shard Size on 32 GB and 4 GB Driver Memory in Imbalanced Shards.

database also increases, which requires more memory for the process. Similarly, with 8 GB of driver memory, the process allocates around 4 - 4.5 GB of primary memory. Since the entire dataset is around 1.8 GB and the available memory is around 4 GB, increasing the batch size linearly decreases the execution time, as the network calls made to the database for reading the tables are reduced. Likewise, for 16 GB and 32 GB of driver memory, having a decent amount of primary memory allows the process execution time to decrease as the batch size increases. However, the optimal driver memory in this case is 8 GB. This is because having a significant driver memory allocated does not necessarily make the algorithm execute faster. In summary, while higher driver memory can help manage larger batch sizes without out-of-memory issues, there is a point beyond which increasing memory further does not yield performance benefits. For this dataset, the optimal driver memory appears to be 8 GB, balancing memory usage and execution efficiency.

Imbalanced Shards In the above section we have evaluated the balanced shards performance. Now we will discuss the experiments results on the imbalanced shards of size 4 and 10 shown in Figure5.8. For the shard size of 4, and driver memory of 4 GB the migration strategy does not execute. For 8 GB driver memory, unlike balanced shards failing at batch size 4, imbalanced shards increases its execution time linearly and it takes more than 400 seconds to migrate the data. Similarly, for 16 GB and 32 GB memory, as per the theory when the batch size increases the execution time must decrease. In this case there was slight increase in run time at batch size 3 and again a sharp decrease in runtime at batch size 4 due to the unstable cloud environments or it could be delayed start up time.

Also, for the shard size of 10, and driver memory of 4 GB, as the memory is restricted and the batch size is increased, the algorithm run time also increases steadily. Whereas, when the driver memory increases from 8 GB to 16 GB, and 16 GB to 32 GB we can observe that there is an overall decrease of run time as the batch size increases with slight peaks at batch size 3.

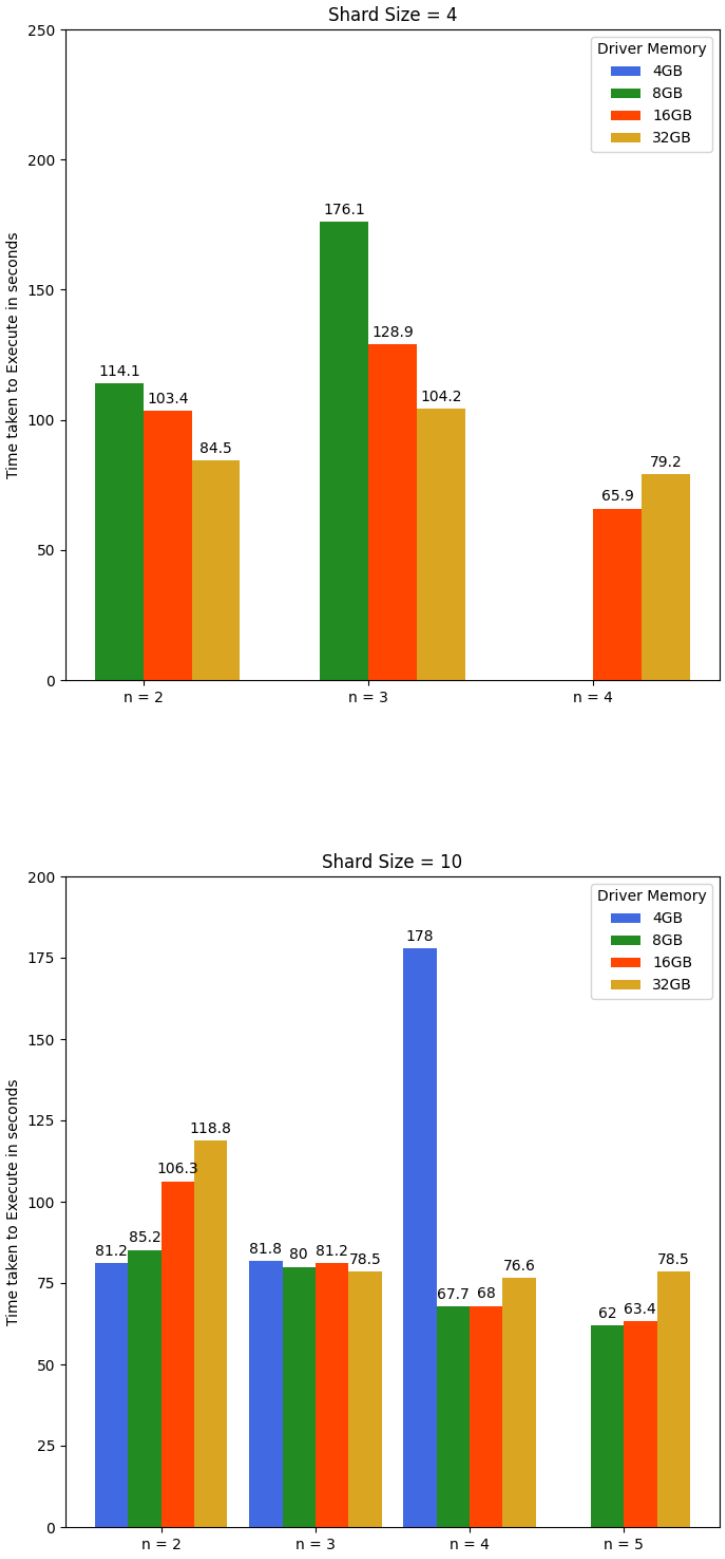

Figure 5.7: Effect of Driver Memory on various Batch Sizes for Shard Size of 4 and 10 in Balanced Shards.

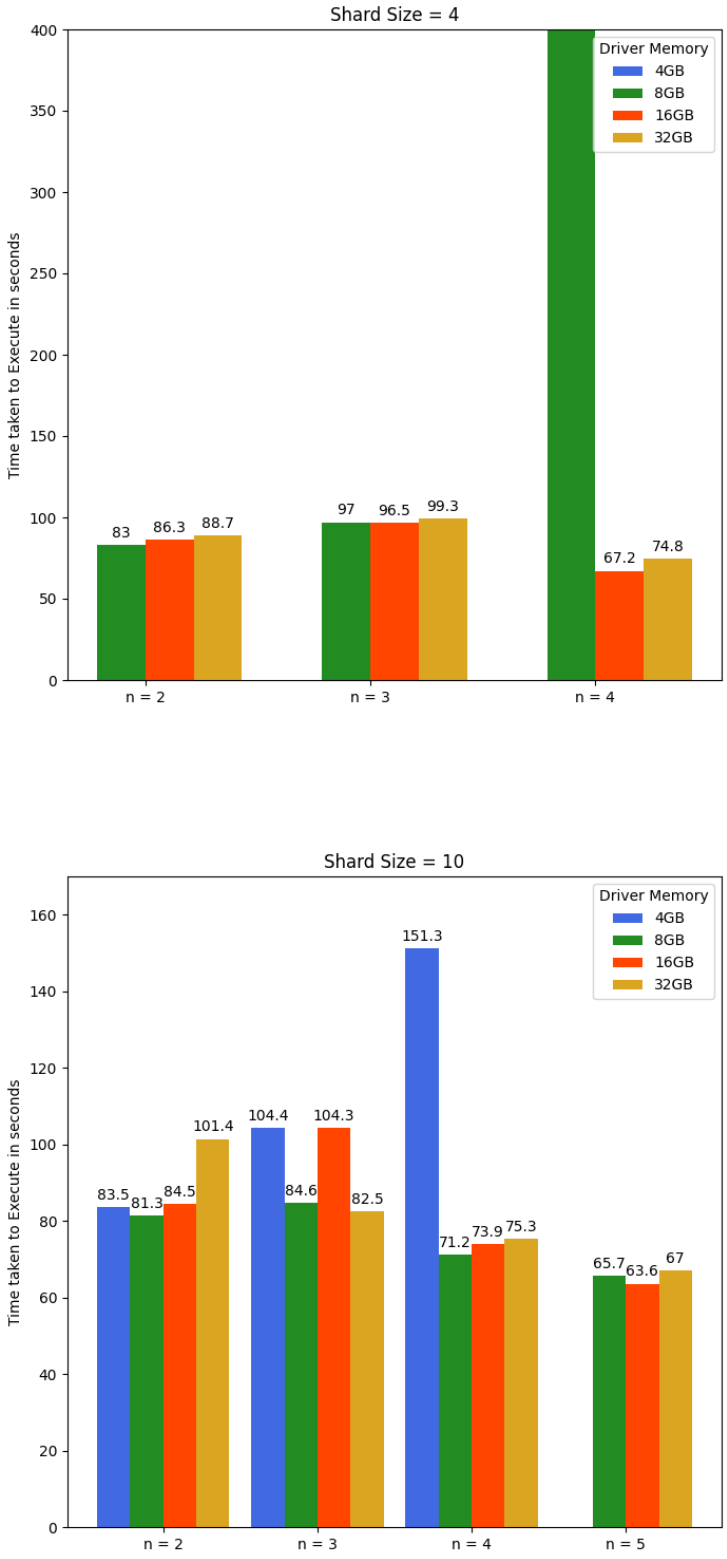

Figure 5.8: Effect of Driver Memory on various Batch Sizes for Shard Sizes of 4 and 10 in Imbalanced Shards.

5.4.2 Concurrent Aggregated Shards Based Algorithm

In this section, we will apply the concurrent aggregated shards based algorithm to various shard sizes ranging from 4 to 10.

Effect of Batch Sizes on Various Thread Sizes We will observe if increasing batch sizes has any effect on a given thread size. To do this, we will increase our batch size while ensuring it always follows the constraints of the aggregated shards-based algorithm.

Balanced Shards

For Shard Size 10 As per Figure 5.9, in balanced shards of size 10, with a thread size of 2 and a batch size of 2, the algorithm takes an average of 82.6 seconds to create a delta table. As the batch size increases for the same thread size, we observe a clear performance improvement. When we increase the thread size to 3 and execute it with a batch size of 2, the algorithm runs faster, averaging 60.7 seconds. The time then increases to 75.3 seconds for a batch size of 3, but decreases to 56.9 seconds when the batch size is increased to 4. Similarly, with a thread size of 4, as the batch size increases, we observe stable performance.

For Shard Size 8,6,4 For a shard size of 8, with thread sizes of 2 and 3, as the batch size increases, we observe a clear performance improvement. Similarly, with a thread size of 4, each thread handles around 2 shards (8 shards / 4 threads) and performs the data migration in one iteration.

For a shard size of 6, the same performance improvement pattern is observed with a thread size of 2. When the thread size is increased to 3 or 4, the algorithm completes the entire data migration in one iteration, and the time taken decreases linearly.

For a shard size of 4, to maintain the constraints of the aggregated shards-based algorithm, we spawn only 2 threads, and it takes an average of 74.7 seconds to perform the task.

Overall, by evaluating all the experiments, we observed that most of the time, as the number of threads increases, the performance of the algorithm also improves, regardless of how distributed the systems are. In the upcoming section, we will examine the performance for imbalanced shards.

Imbalanced Shards Similar to balanced shards, the performance improvements in imbalanced shards are illustrated in Figure 5.10. For a shard size of 10 and thread sizes of 2,3, and 4 we observe an overall decrease in runtime as the batch size increases, except for the batch size of 2 with a thread size of 2.

For a shard size of 8 with thread sizes of 2,3 and 4, we find the same performance improvement as the batch size increases. For a shard size of 6 and thread sizes of 2 and 3, we observe similar performance improvements. Likewise, for a shard size of 4 and batch size of 2 we can observe the algorithm took an average time of 74.7 seconds.

Overall, imbalanced shards exhibit same effects to balanced shards, with efficiency improving as the batch size increases for a given thread size.

5.5 Experimental Results for 200 Million Dataset

In this section, we will examine the results of the aggregated shards-based algorithm on a dataset containing 200 million records and discuss its performance.

5.5.1 Sequential Aggregated Shards Based Algorithm

When executing the sequential aggregated shards-based strategy on balanced shards of size 50, we started with a batch size of 3 and experimented up to a batch size of 6. As shown in Figure 5.11, the algorithm takes an average of approximately 2500 seconds for a batch size of 3, decreasing steadily to 1518 seconds for a batch size of 6.

Similarly, for imbalanced shards with sufficient driver memory allocation, the time taken for a batch size of 3 starts at an average of 2375 seconds and decreases to 1864 seconds for a batch size of 6.

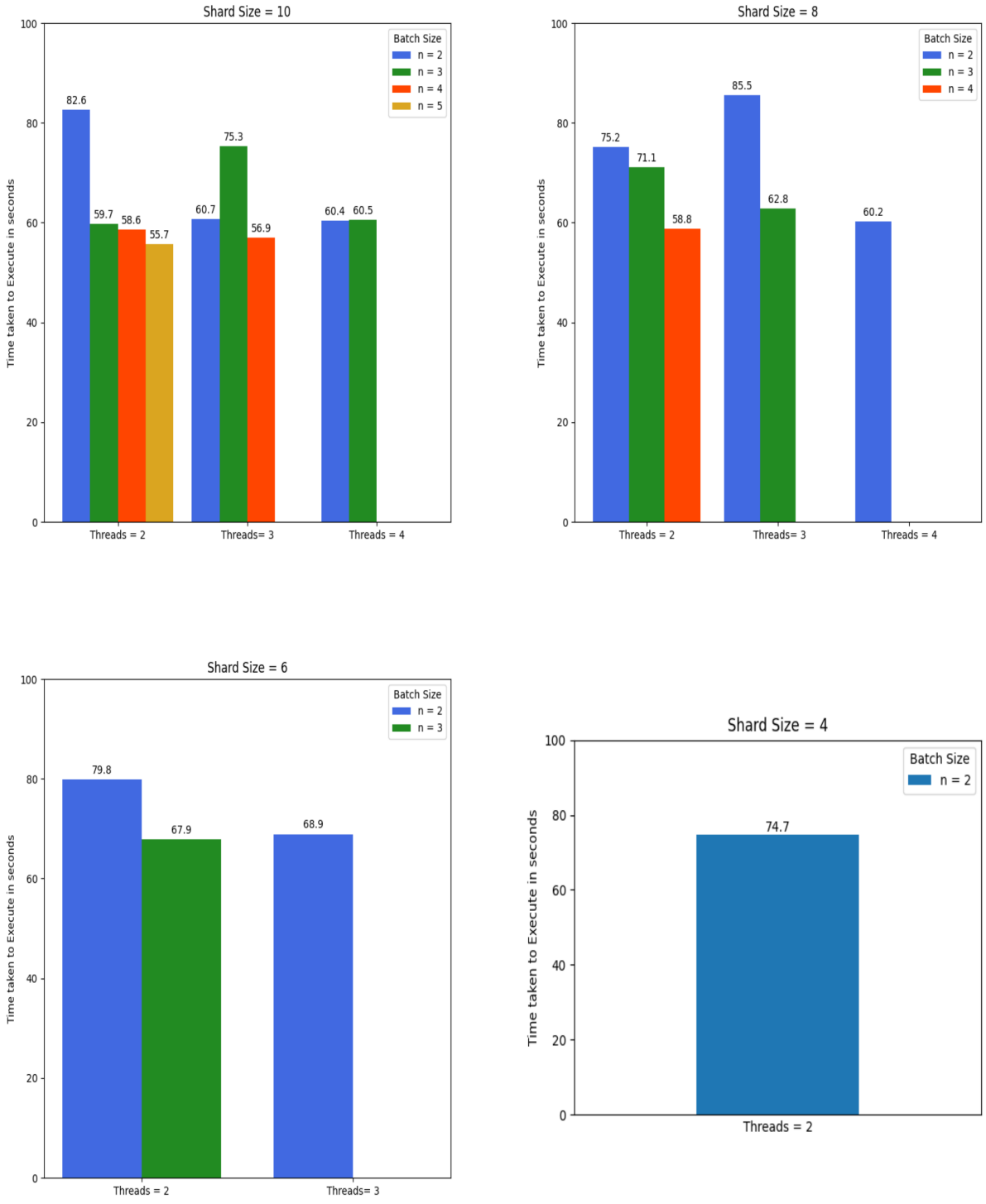

Figure 5.9: Effect of Threads on Batch Sizes for Balanced Concurrent Aggregated Shards Based Algorithm.

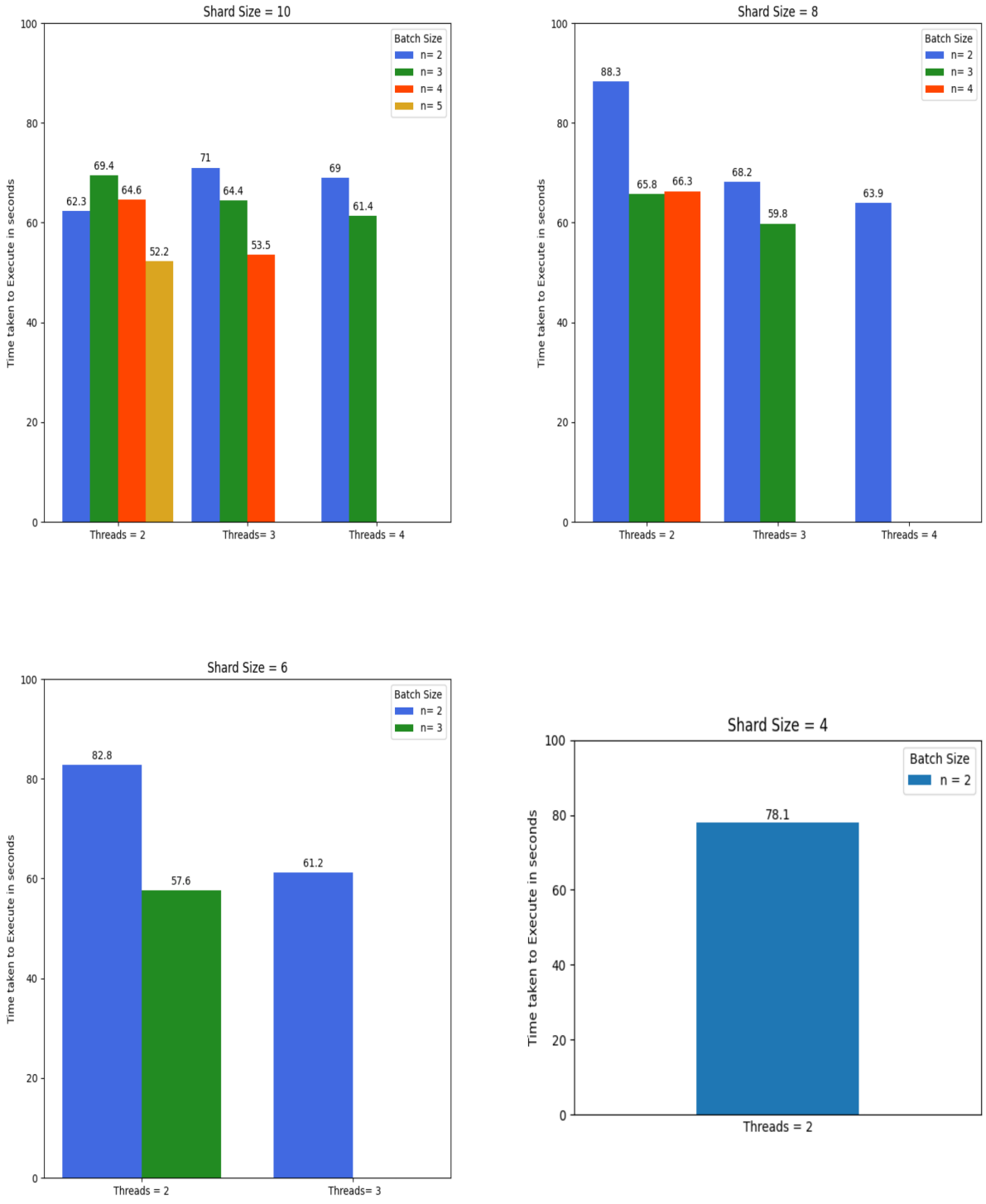

Figure 5.10: Effect of Threads on Batch Sizes for Imbalanced Concurrent Aggregated Shards Based Algorithm.

Overall, we can conclude that as the batch size increases, there is a consistent performance improvement for both balanced and imbalanced shards.

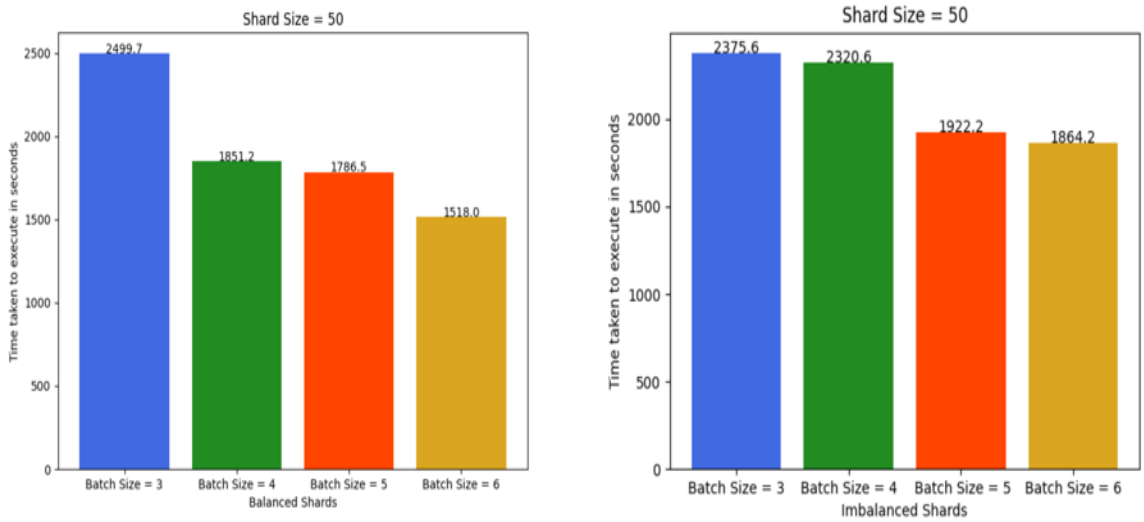

Figure 5.11: Runtime comparison between Imbalanced and Balanced Shards in Sequential Aggregated Shards Based Algorithm.

5.5.2 Concurrent Aggregated Shards Based Algorithm

In this subsection, we will look at how the algorithm performs on balanced shards and imbalanced shards and will discuss the results.

Balanced Shards

As per Figure 5.12, for a thread size of 3 and a batch size of 2 with 50 GB of driver memory, the algorithm took an average of 1721 seconds. When the batch size was increased, the average time increased to 1965 seconds. This occurred because the driver memory allocated to the process was around 25-26 GB, and as the batch size increased, the shards that needed to be accommodated in primary memory also increased. The memory required to process these shards was insufficient, which caused the algorithm to take more time to process with a thread size of 3 when the batch size was increased from 2 to 3.

For thread sizes of 4, 5, and 6, we performed experiments to check if thread size had any effect. To confirm this, we scaled the driver memory from 50 GB to 90 GB and 110 GB.

We observed that as the driver memory and thread count increased, the time taken by the algorithm drastically reduced. This confirms that scaling threads improves the performance of the migration strategy.

Overall, from the experiments with balanced shards and a thread size of 3, we learned that while scaling using batch size, the algorithm must have sufficient driver memory to execute faster. Similarly, when scaling using threads, it is crucial to reconsider and accurately perform driver memory calculations to ensure the algorithm transfers data to the target systems reliably.

Imbalanced Shards

Unlike balanced shards, in imbalanced shards, we maintain the same driver memory of 110 GB, utilizing a 128 GB (m5.8xlarge) instance. In our experiments, with a thread size of 3 and a batch size of 2, the algorithm took an average of 1679 seconds over 5 iterations. When we increased the batch size to 3, we observed a runtime improvement of approximately 22.5%, thanks to the ample driver memory available.

Similarly, for thread sizes of 4 and 5 with a batch size of 2, there was a clear improvement in performance. However, for a thread size of 6, we noticed a slight increase in time, which is attributed to the slow startup time of Apache Spark.

Overall, these experiments suggest that sufficient driver memory aids the algorithm when scaling using batch size. Additionally, scaling using thread number generally results in clear performance improvements, although there might be an increase in time if resources are insufficient or if cloud congestion occurs. Please, refer Figure 5.13 for more details regarding the above explanation.

5.6 Summary

In this chapter, we have examined the results of the aggregated shards based algorithm and has seen how it has performed for 10 million and 200 million dataset under various

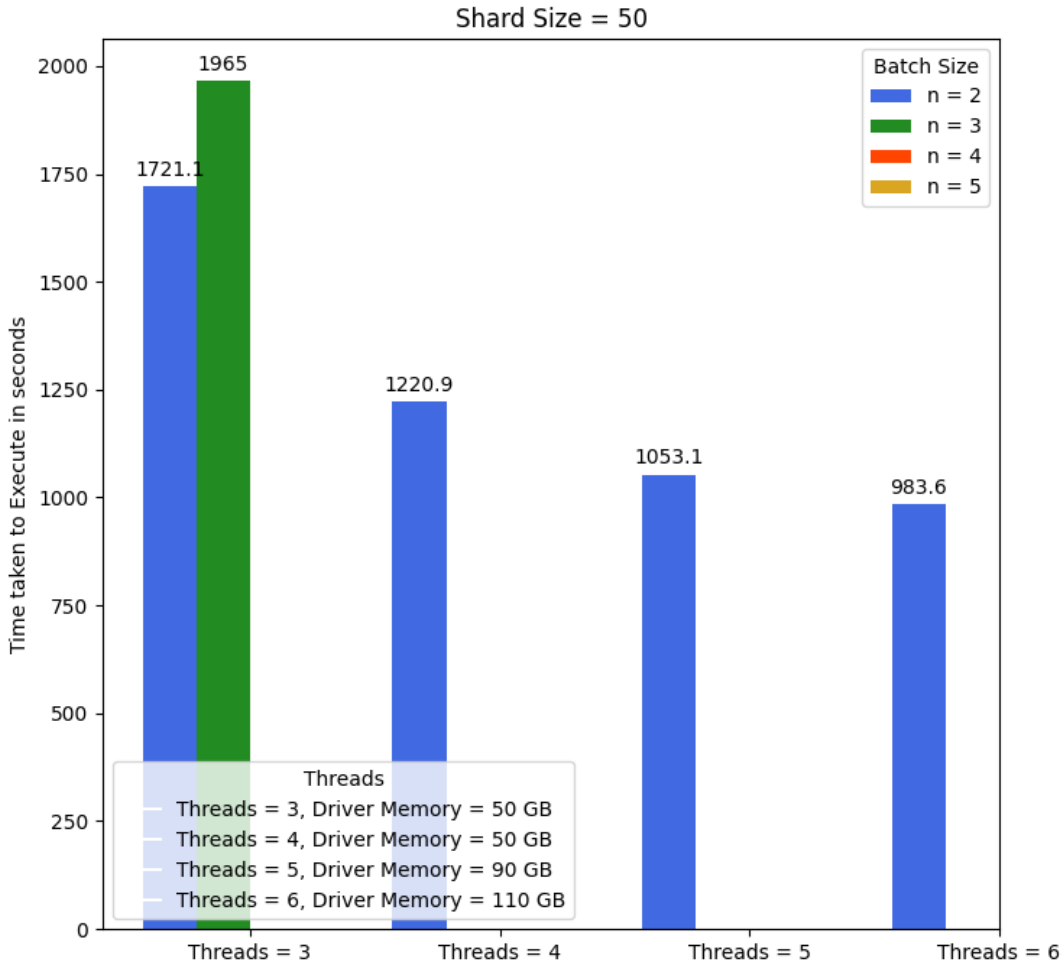

Figure 5.12: Effect of Threads on Balanced Concurrent Aggregated Shards Based Algorithm.

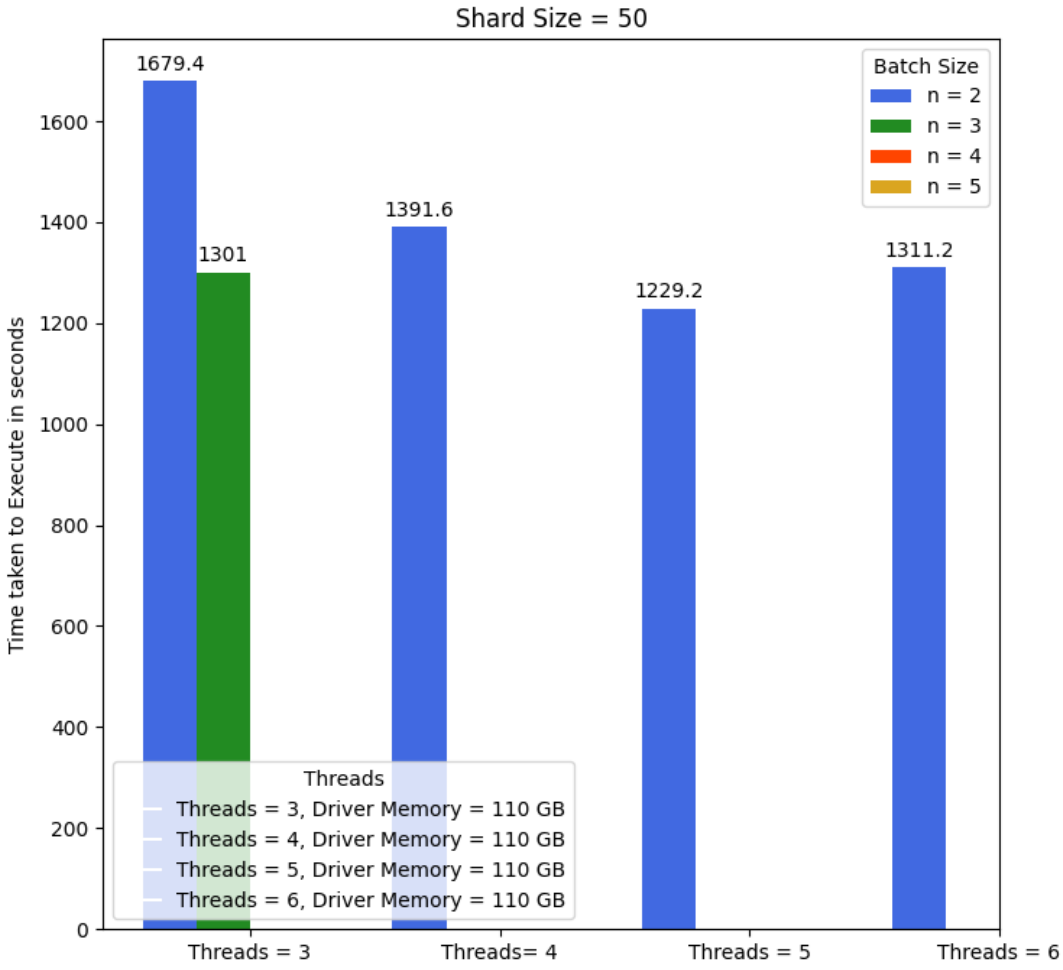

Figure 5.13: Effect of Threads on Imbalanced Concurrent Aggregated Shards Based Algorithm.

batch sizes, shard sizes and thread sizes. In the upcoming chapter, we will conclude the best performing strategies by contrasting the performance improvement on 200 million dataset.

Chapter 6

Conclusions

In this thesis, we proposed several batch algorithms to enable migration from sharded databases. These consider Sequential and Concurrent strategies for both balanced and imbalanced shards. Initially, in Chapter 1 at Section 1.3 we have proposed the objectives of our thesis. Therefore, in the conclusion we recall the objectives and will summarize them by contrasting the proposed algorithms.

6.1 Accomplished Objectives

In this section we will answer few of the main objectives defined in the Section 1.3.

1. Evaluating the efficiency of migration strategies.

2. Cost optimization of cloud services and Big data systems.

6.1.1 Evaluating the efficiency of migration strategies

From the algorithms that we have proposed. We will look at the efficiency of them by analyzing the time it took to execute on the 200 million dataset, due to its huge data proportion.

As we know, we have the following variations of the algorithms: 1) Sequential, and 2) Concurrent. Table 6.1 depicts the execution time taken by the algorithms for a shard size of 50 and the IRT in the table implies initial runtime and FRT means final runtime. The units used for calculation are in seconds.

Category	**Algorithm**	**IRT (s)**	**FRT (s)**	**Performance Improvement (%)**
Sequential Strategy				
Balanced Shards	Sequential Shard-by-Shard	7370.1	N/A	N/A
Balanced Shards	Sequential Aggregated Shards-based (Batch Size 3-6)	2499	1518	39.2%
Imbalanced Shards	Sequential Shard-by-Shard	4736.9	N/A	N/A
Imbalanced Shards	Sequential Aggregated Shards-based (Batch Size 3-6)	2375.6	1864.2	21.5%
Concurrent Strategy				
Balanced Shards	Concurrent Shard-by-Shard (Thread Size 3-6)	2625.0	1864	29%
Balanced Shards	Concurrent Aggregated Shards-based (Batch Size 2, Thread Size 3-6)	1721.1	983.6	42.8%
Imbalanced Shards	Concurrent Shard-by-Shard (Thread Size 3-6)	2873.3	1668.2	41.9%
Imbalanced Shards	Concurrent Aggregated Shards-based (Batch Size 2, Thread Size 3-6)	1679.4	1311.2	21.9%

Table 6.1: Sequential and Concurrent Strategy Runtimes and Performance Improvements.

Sequential

We can infer that in the sequential category for balanced shards, the shard-by-shard algorithm takes 2.04 hours to migrate the data. Similarly, for the sequential aggregated shards-based strategy in row number 2, we can observe that for batch sizes 3 to 6, the algorithm starts at an average of 2499 seconds (41.65 minutes) and ends at an average of 1518 seconds (25.30 minutes), with a runtime reduction of around 39.2%.

For imbalanced shards, the shard-by-shard algorithm takes an average of 4736.9 seconds (1.31 hours). In contrast, the sequential aggregated shards-based algorithm, for batch sizes 3 to 6, starts at an average of 2375.6 seconds (39.59 minutes) and ends at 1864.2 seconds (31.06 minutes), with a runtime reduction of 21.5%.

Concurrent

In the concurrent technique, for thread sizes of 3 to 6 in the balanced shard-by-shard strategy, the algorithm took an average runtime from 2625 seconds (43.75 minutes) to 1864 seconds (31.06 minutes), with a runtime reduction of 29%. Similarly, in the concurrent aggregated shards-based strategy for thread sizes of 3 to 6 and a batch size of 2, it took an average of 1721.1 seconds (28.6 minutes) for a thread size of 3 and ended at 983.6 seconds (16.39 minutes) for a thread size of 6, with a performance improvement of 42.8%.

For imbalanced shards, in the concurrent shard-by-shard strategy with a thread size of 3, the algorithm started with an average runtime of 2873.3 seconds (47.88 minutes) and, for a thread size of 6, it took around 1668.2 seconds (27.80 minutes), with an efficiency improvement of 41.9%. Similarly, in the concurrent aggregated shards-based strategy, the algorithm took an average of 1679 seconds (27.98 minutes) for a batch size of 2 and a thread size of 3, and ended at 1311 seconds (21.85 minutes) for a batch size of 2 and a thread size of 6 with a performance improvement of 21.9%.

Upon careful evaluation of the aforementioned strategies, in the sequential category, employing the aggregated shards-based algorithm facilitates faster data retrieval for ana-

lytics. This is achieved by scaling the algorithm to process multiple shards in memory. Moreover, in a distributed cloud-based environment, despite the isolation of machines, significant performance improvements of approximately 39.2% and 21.5% were observed for balanced and imbalanced shards, respectively.

In the concurrent strategy, increasing the thread size in the shard-by-shard algorithm significantly improved performance compared to the sequential strategy in both balanced and imbalanced shards. Similarly, the aggregated shards-based strategy executed faster than the shard-by-shard strategy, as it processes 2 shards unlike the shard-by-shard strategy processes only 1 shard. However, the algorithm's efficiency is limited by the thread size. As the thread size increases, the system's limited resources can lead to network contention.

6.1.2 Cost optimization of cloud services and Big data systems

Here, we will discuss the following points: 1) Optimization of Cloud Services, and 2) Optimization of Big Data Systems.

Optimization of cloud services involves improving the running time of algorithms. Typically, the service fees associated with the cloud are paid for storage, network, and compute resources of the instance. Therefore, with our proposed algorithms, organizations can transfer data from sharded middleware systems into target systems faster, reducing runtime and consequently saving thousands of dollars.

In addition to runtime optimization, it is crucial to optimize big data systems. This requires a solid understanding of big data processing frameworks such as Apache Spark. Before implementing the algorithms, understanding the internal processing concepts of these frameworks helps in building faster data pipelines. In our case, we have proposed resource allocation strategies for algorithms in previous chapters, which aided our implementation, allowing the algorithms to run more efficiently.

6.2 Future Work

While conducting our experiments and evaluating our results, we have noted a few future directions to optimize the existing work and have seen an opportunity to implement the proposed strategy in a different system.

6.2.1 Optimizing Partition Functions

In our proposed, partition strategy the current technique generates mixed partitions. In this context, the meaning of mixed partitions means having table sizes in the partitions in a totally distributed manner. We experimented with a few other strategies that creates partitions in a sorted manner. Therefore, finding a better strategy can enable the proposed strategies to execute even faster.

6.2.2 Stream Based Data Processing

In this thesis, we have proposed the batch algorithms to process the data. Therefore in future, data engineering researchers can look into the opportunity of applying the above strategies to the area of stream processing from sharded databases.

Bibliography

[1] Airbyte. https://github.com/airbytehq/airbyte. Accessed: 2024-06-30.

[2] Amazon EMR Documentation. https://docs.aws.amazon.com/emr/. Accessed: 2024-06-30.

[3] Amazon web services. https://aws.amazon.com/. Accessed: 2024-06-30.

[4] Azure blob storage. https://learn.microsoft.com/en-us/azure/storage/blobs/. Accessed: 2024-06-30.

[5] Datalakes. https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/. Accessed: 2024-06-30.

[6] Event loop issue. https://stackoverflow.com/questions/55409641/asyncio-run-cannot-be-called-from-a-running-event-loop-when-using-jupyter-no. Accessed: 2024-06-30.

[7] Mysql. https://www.mysql.com/. Accessed: 2024-08-15.

[8] Nycdatasets. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2024-06-30.

[9] Open table formats. https://www.starburst.io/data-glossary/open-table-formats/. Accessed: 2024-07-03.

[10] Postgresql database system. https://www.postgresql.org/. Accessed: 2024-06-30.

[11] Spark and Iceberg Quickstart - Apache Iceberg. Accessed: 2024-06-30.

[12] Spark Guide | Apache Hudi. Accessed: 2024-06-30.

[13] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of computing. *Communications of the ACM*, 53(4):50–58, 2010.

[15] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, page 8, 2021.

[16] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, 2017.

[17] Aryan Bansel, Horacio González-Vélez, and Adriana E. Chis. Cloud-based nosql data migration. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 224–231, 2016.

[18] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, jan 2015.

[19] Dhruba Borthakur. Hdfs architecture. *Document on Hadoop Wiki. URL http://hadoop. apache. org/common/docs/r0*, 20, 2010.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[21] M Elamparithi and V Anuratha. A review on database migration strategies, techniques and tools. *World Journal of Computer Application and Technology*, 3(3):41–48, 2015.

[22] A Fahmi and Y H Putra. Database migration strategies and techniques to minimize unexpected dysfunctionality. *IOP Conference Series: Materials Science and Engineering*, 662(6):062003, nov 2019.

[23] Mayank Gokarna. Reasons behind growing adoption of cloud after covid-19 pandemic and challenges ahead. *CoRR*, abs/2103.00176, 2021.

[24] Developer Guide. Amazon simple storage service, 2008.

[25] Scott Haines. Workflow orchestration with apache airflow. In *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*, pages 255–295. Springer, 2022.

[26] Abou_el_ela Abdou Hussein et al. Data migration need, strategy, challenges, methodology, categories, risks, uses with cloud computing, and improvements in its using with cloud using suggested proposed model (dmig 1). *Journal of Information Security*, 12(01):79, 2021.

[27] Joanna Konopko. Distributed and parallel approach for handle and perform huge datasets. In *AIP Conference Proceedings*, volume 1702. AIP Publishing, 2015.

[28] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[29] Ruiyuan Li, Liang Zhang, Juan Pan, Junwen Liu, Peng Wang, Nianjun Sun, Shanmin Wang, Chao Chen, Fuqiang Gu, and Songtao Guo. Apache shardingsphere: A holistic and pluggable platform for data sharding. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2468–2480, 2022.

[30] Alza A Mahmood. Automated algorithm for data migration from relational to nosql databases. *Al-Nahrain J. Eng. Sci*, 21(1):60, 2018.

[31] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, aug 2019.

[32] M Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, 28(1):125–128, 1996.

[33] Andrew Pavlo and Matthew Aslett. What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55, sep 2016.

[34] Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Tian J Wang. Modelling data pipelines. In *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*, pages 13–20. IEEE, 2020.

[35] Motwani Rajeev. Query processing, resource management, and approximation ina data stream management system. In *Conference on Innovative Data Systems Research, 2003*, 2003.

[36] Haines Scott. *Workflow Orchestration with Apache Airflow*, pages 255–295. Apress, Berkeley, CA, 2022.

[37] Kirill Shirinkin. *Getting Started with Terraform*. Packt Publishing Ltd, 2017.

[38] Salomé Simon. Brewer's cap theorem. *CS341 Distributed Information Systems, University of Basel (HS2012)*, 2000.

[39] Michael Stonebraker. New opportunities for new sql. *Commun. ACM*, 55(11):10–11, nov 2012.

[40] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20(24):79, 2011.

[41] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 789–796, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Deepak Vohra and Deepak Vohra. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pages 325–335, 2016.

[44] Yansyah Saputra Wijaya and Arry Akhmad Arman. A framework for data migration between different datastore of nosql database. In *2018 International Conference on ICT for Smart Society (ICISS)*, pages 1–6, 2018.

[45] Yifan Wu. Is a dataframe just a table? In *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.

[46] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.