UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

# Implementation of Join Operation over NoSQL Databases

Dimitrios Panakos ME2123

Master of Science

Information Systems and Services

Big Data And Analytics

Supervising Professor: Christos Doulkeridis

Athens January 2024

# Πίνακας περιεχομένων

## Abstract

In the large volumes of data scattered and stored in various non-relational database systems, there is a need for information retrieval. With many systems, each having a different API for retrieving information, there arises the need to find data located in more than one database and retrieve it based on common information. For this reason, this thesis extends the NODA API with the join process. NODA offers a rich API that provides connectivity with various non-relational databases without requiring the client to use the API of each non-relational database. Therefore, this work will extend NODA with the join function to enable the connection of common data in more than one different database.

Στους μεγάλους όγκους δεδομένων που υπάρχουν διάσπαρτοι αποθηκευμένοι σε διάφορα συστήματα μη σχεσιακών βάσεων δεδομένων, υπάρχει ανάγκη ανάκτησης της πληροφορίας. Τα συστήματα πολλά με διαφορετικό API το κάθε ένα για ανάκτηση πληροφορίας προκύπτει η ανάγκη εύρεσης δεδομένων που βρίσκονται σε πάνω από μία βάση να ανακτηθούν βάσει κοινής πληροφορίας. Για τον λόγο αυτό έρχεται η διπλωματική αυτή εργασία να επεκτείνει το API του NODA με την διαδικασία του join. Το NODA προσφέρει ένα πλούσιο API που δίνει την συνδεσιμότητα με διάφορες μη σχεσιακές βάσεις δεδομένων χωρίς να απαιτείται από τον πελάτη να χρησιμοποιήσει το API της κάθε μη σχεσιακής βάσης. Η εργασία αυτή λοιπόν θα επεκτείνει το NODA με την λειτουργία του join ώστε να δώσουμε την δυνατότητα να συνδέσουμε κοινά δεδομένα σε παραπάνω από μία βάση διαφορετική.

# 1.Introduction

NoSQL databases, mark a significant shift in data management by offering flexible and scalable alternatives to traditional relational databases. These databases cater for a range of unstructured data types, improving storage and retrieval processes for large volumes of data. Their prominence has risen due to the challenges posed by the scale, variety, and velocity of big data, areas where traditional relational models often fall short. NoSQL systems eschew the rigid schemas of relational databases, opting instead for dynamic, schema-less structures that are well-suited for rapidly changing data sets. They focus on horizontal scalability and are integral to distributed computing environments, making them crucial for applications that demand high performance, scalability, and data management flexibility, such as real-time analytics, content management, e-commerce platforms, and more.

This thesis delves into the implementation of the join operation in various NoSQL database systems. The integration of join operations in NoSQL databases is challenging due to their fundamental design goals of scalability, performance, and flexibility, which are prioritized over the fixed relationships found in relational databases. While NoSQL databases excel in environments that require scalability and adaptability, careful planning of data modeling and querying is essential for any specific application. Although some NoSQL databases provide methods to handle relationships, like embedding or linking documents, these approaches can differ significantly from traditional relational database JOIN operations.

So the main goal of this thesis is an implementation of join operation among different NoSQL databases. This will be handled through updating the NODA API. This API offers operations for transactions over NoSQL databases such as filtering, projections and efficient use of spatiotemporal queries. NODA uses Apache Spark domain models for databases connection and result viewing. The result returned with the use of Dataset object. Here comes this thesis to update the NODA API to handle join operations through different databases. This is done by using Apache Spark join operations among Datasets, plus by enriching the API through a custom join implementation mechanism based on hash join algorithm.

The chapter are written to give an easy way to the reader to understand the implementation of join operation inside NODA API without having to learn about in detail.

Chapter 2 first subchapter represent a category of database management systems that diverge from the traditional relational database framework by accommodating various data models, enhancing flexibility and scalability for large, dynamic datasets. Unlike relational databases like MySQL and PostgreSQL, which rely on structured schemas and SQL, NoSQL databases handle unstructured, semi-structured, and structured data, making them suitable for web, mobile, social media, and big data applications.

Types of NoSQL databases include:

- Document-oriented databases (e.g., MongoDB) that store data in JSON-like documents.
- Key-value stores (e.g., Redis, Amazon DynamoDB) for rapid processing.
- Column-family stores (e.g., Apache Cassandra) for efficiently managing large datasets.
- Graph databases (e.g., Neo4j) for handling data with complex relationships.

NoSQL databases operate without fixed schemas, allowing for rapid development and horizontal scaling by adding nodes to distributed systems. They offer optimized performance for specific query types and improve fault tolerance and data availability through distributed architectures. However, they may have limited querying capabilities, lack standardized query languages, and offer less transactional integrity compared to relational databases.

Choosing between NoSQL and relational databases depends on the application's specific requirements, such as data structure, query complexity, scalability, and development needs.

The second subchapter describes the need of big data processing. Big data encompasses extensive and complex datasets that exceed the capabilities of traditional data processing tools, typically ranging from terabytes to petabytes. These datasets originate from various sources, including social media, sensors, and transaction logs, and require prompt processing for timely insights. Characterized by its volume, variety, and velocity, big data includes structured, semi-structured, and unstructured formats. Managing and analyzing such data necessitates advanced technologies

like Hadoop and Apache Spark, which facilitate distributed and real-time processing, respectively. NoSQL databases like MongoDB and Cassandra support handling large volumes of unstructured data.

The third subchapter continues with a description of Apache Spark framework that it is used. Apache Spark is an open-source, distributed computing system designed for fast and versatile data processing. It extends the MapReduce model to support a variety of computations, such as interactive queries and stream processing, providing high-level APIs in Java, Scala, Python, and R. Spark's in-memory processing capabilities make it significantly faster than traditional disk-based engines like Hadoop. It offers advanced analytics through components like Spark SQL for SQL queries, MLlib for machine learning, and GraphX for graph processing. With its unified engine, Spark handles diverse workloads, including batch processing, real-time analytics, and machine learning, while integrating seamlessly with Hadoop and other storage systems. The active community and extensive ecosystem further enhance its capabilities, making Spark a powerful tool for big data analytics across various use cases.

At the final subchapter of this there is a description of NODA API. The NoDA paper abstracts data access across various NoSQL stores and provides a unified, SQL-like querying interface. Experimental results show significant performance improvements in spatio-temporal queries across different NoSQL systems. The NODA emphasizes the popularity and benefits of NoSQL stores, such as scalability, flexible data models, and high availability, but notes their limitations in optimized indexing and standardized query languages. These issues complicate application development and hinder seamless transitions between NoSQL stores, prompting the need for a unified solution. NODA offers a simplified interface with a programming API and SQL-like language, hiding the complexities of individual NoSQL query languages. Inspired by Apache Spark, its design separates query definition from execution, enabling efficient, lazy execution of operations.

At the third chapter we introduce the implementation of join operations inside NODA. Before from that, a detailed reference upon the join operations that exists on traditional relational database systems is

performed. After that there is a description of the code implementation of the solution.

# 2 Technical Background

This chapter provides the necessary information to understand the scope of this thesis. It begins with a description of NoSQL systems and their data storage methods. The second section presents an overview of big data processing frameworks. The third section details Apache Spark, which is used during the implementation phase. The final section offers a detailed description of the NODA API, which serves as the core functionality for operating different NoSQL stores.

## 2.1 NoSQL Database Systems

NoSQL databases represent a category of database management systems that diverge significantly from the conventional relational database framework. These systems are not confined to structured tables and predefined schemas but are engineered to accommodate a broader spectrum of data models, enhancing flexibility and scalability for managing large, dynamic datasets.

Traditional relational databases, such as MySQL and PostgreSQL, rely on a structured schema to organize data into tables. Each table consists of rows and columns, with each row representing a unique record and each column representing a specific attribute of the data. While this approach is highly effective for managing structured data and supports complex queries using SQL (Structured Query Language), it can be restrictive when dealing with unstructured or semi-structured data, which is increasingly common in modern applications.

NoSQL databases, on the other hand, excel at handling various data types — unstructured, semi-structured, and structured — making them particularly effective for modern applications encountered in web and mobile environments, social media platforms, and intensive big data projects. The flexibility to manage diverse data types is one of the key advantages of NoSQL databases, enabling developers to store and process data in a way that aligns more closely with the application's needs.

The landscape of NoSQL databases is diverse, with several distinct types designed to meet specific needs. For example, document-oriented

databases like MongoDB organize data in a document format akin to JSON (JavaScript Object Notation), which supports complex data structures and facilitates efficient storage and retrieval. Documents in MongoDB can contain nested structures and arrays, allowing for a more natural representation of hierarchical data. This flexibility makes document databases ideal for content management systems, e-commerce platforms, and other applications that require the storage of complex data.

On the other hand, key-value stores maintain data in unique key-value pairs. This structure is prized for its rapid processing and ease of use, although it can sometimes restrict the complexity of queries. Key-value stores, such as Redis and Amazon DynamoDB, are often used for caching, session management, and real-time analytics due to their ability to quickly retrieve values based on a unique key.

Column-family stores, another type of NoSQL database, organize data into columns rather than rows. This approach allows for efficient storage and retrieval of large datasets with many attributes, as only the necessary columns are accessed. Apache Cassandra is a popular example of a column-family store, frequently used in applications that require high availability and scalability, such as time-series data and IoT (Internet of Things) applications.

Graph databases, such as Neo4j, are designed to handle data with complex relationships. Instead of tables, these databases use nodes, edges, and properties to represent and store data. Graph databases excel in applications that involve social networks, recommendation systems, and fraud detection, where understanding and traversing relationships between data points is crucial.

NoSQL databases typically operate without fixed schemas, allowing for greater developmental agility as fields can be added to documents or entries without necessitating changes to the overall database schema. This feature is particularly valuable for applications that need to evolve rapidly or handle data with variable structures. The schema-less nature of NoSQL databases means that developers can iterate quickly, making

changes to the data model without the need for costly and time-consuming migrations.

Designed for horizontal scaling, NoSQL databases can efficiently manage growing workloads by integrating additional nodes into the distributed system. This scalability is crucial for applications experiencing growth in data volume and user traffic. Horizontal scaling, or scaling out, involves adding more machines to handle increased load, as opposed to vertical scaling, which involves adding more resources to a single machine.

NoSQL databases are designed to scale out easily, distributing data and queries across multiple servers to ensure performance and reliability.

Optimized for specific query types and data access patterns, many NoSQL databases deliver outstanding performance for particular operations. For instance, key-value stores are optimized for swift data retrieval tasks, making them ideal for use cases where low latency is essential. Similarly, document databases are optimized for storing and querying hierarchical data structures, while column-family stores excel at handling large-scale, distributed data.

Additionally, these databases are generally structured to perform well in distributed settings, distributing data across multiple servers or nodes to improve fault tolerance and increase data availability. This distributed architecture enhances the resilience of NoSQL databases, allowing them to continue operating even if some nodes fail. Data replication and sharding are common techniques used to achieve high availability and fault tolerance in NoSQL databases.

Despite these advantages, NoSQL databases also present certain challenges. They might offer more limited querying capabilities than traditional relational databases, potentially hindering performance for certain types of applications. The absence of a standardized query language across different NoSQL systems can also complicate development and integration processes. Developers may need to learn and use different query languages and APIs for each NoSQL database, increasing the complexity of building and maintaining applications.

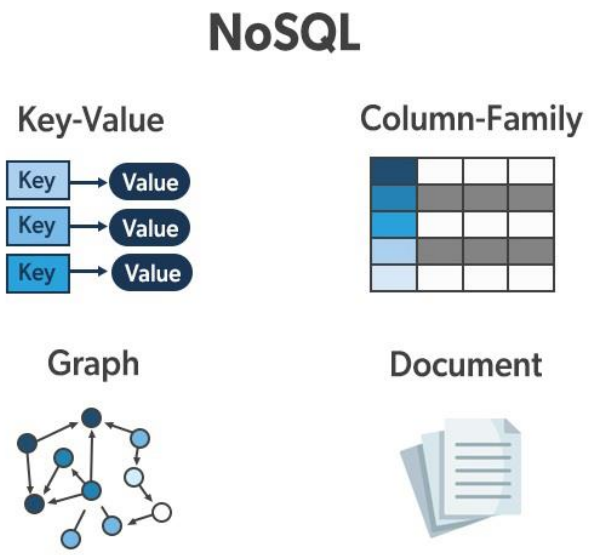
Moreover, while NoSQL databases are optimized for specific use cases, they may not provide the same level of transactional integrity as relational databases. For applications that require complex transactions, such as

financial systems or inventory management, relational databases like PostgreSQL or MySQL might be more appropriate due to their support for ACID (Atomicity, Consistency, Isolation, Durability) properties.

Although NoSQL databases are invaluable for certain applications, they are not universally applicable. For projects that require intricate transactions, closely defined schemas, and complex relational data interdependencies, traditional relational databases might be more appropriate, offering advantages in terms of transactional integrity and relational querying capabilities. The choice between NoSQL and relational databases should be guided by the specific requirements and constraints of the application, considering factors such as data structure, query complexity, scalability needs, and development agility.

In conclusion, NoSQL databases offer a versatile and powerful alternative to traditional relational databases, particularly for modern applications that demand flexibility, scalability, and the ability to handle diverse data types. By understanding the strengths and limitations of different types of NoSQL databases, developers can make informed decisions about which database technology best meets the needs of their projects.

At the image below represented the most common types of NoSQL stores.



## 2.2 Big Data Proccessing

Big data refers to the extensive and complex datasets that surpass the capacity of traditional data processing tools to manage effectively. These datasets range enormously in size, typically spanning from terabytes to petabytes and even more. They encompass a wide variety of data types, both structured and unstructured, sourced from diverse origins like social media platforms, sensor outputs, and transaction logs.

The generation of big data occurs at rapid velocity, necessitating its prompt processing to derive valuable insights. Often, real-time or near-real-time analysis is required to make timely decisions based on the continuously accumulating data. This rapid pace of data generation is driven by the increasing digitization of our world, where every action, transaction, and interaction leaves a digital footprint that contributes to the ever-growing volume of big data.

Big data is characterized by its varied formats. It includes structured data stored in traditional databases, semi-structured data such as XML files, and unstructured data, which can be anything from text files to multimedia content. Structured data is highly organized and easily searchable, while semi-structured data lacks a fixed schema but still contains tags or markers to separate elements. Unstructured data, on the other hand, is the most challenging to manage due to its lack of a predefined format, encompassing text documents, images, videos, and more.

The management and analysis of this heterogeneous data demand sophisticated approaches and technologies. Traditional data processing tools and methods fall short when dealing with the sheer volume, variety, and velocity of big data. As a result, new technologies and frameworks have emerged to address these challenges.

Key Technologies and Frameworks in Big Data Processing are presented below.

*Hadoop*

[4] Hadoop is an open-source framework that allows for the distributed processing of large datasets across clusters of computers. It uses a simple

programming model to process data in parallel, significantly improving processing speed and efficiency. Hadoop's ecosystem includes tools such as Hadoop Distributed File System (HDFS) for storage and MapReduce for processing, making it a foundational technology in the big data landscape.

*Apache Spark*

Apache Spark offers an advanced data processing engine that can handle batch and real-time data processing. Spark's in-memory computing capabilities make it much faster than traditional disk-based processing frameworks, allowing for quicker data analysis and decision-making. It supports a wide array of applications, including machine learning, graph processing, and streaming analytics.

*Apache Flink*

[5] Apache Flink is another robust framework designed for stateful computations over unbounded and bounded data streams. Flink excels in processing event-driven data and real-time analytics, providing high throughput and low latency. It supports complex event processing and machine learning, making it versatile for various big data applications.

*Apache Storm*

[7] Apache Storm is a distributed real-time computation system designed for processing streams of data. It can handle massive volumes of data and provide near-instantaneous processing and analytics. Storm is particularly well-suited for tasks like real-time analytics, online machine learning, and continuous computation.

*Apache Kafka*

[6] Apache Kafka is a distributed streaming platform that can publish, subscribe to, store, and process streams of records in real-time. Kafka is designed for high throughput and fault tolerance, making it ideal for building real-time data pipelines and streaming applications.

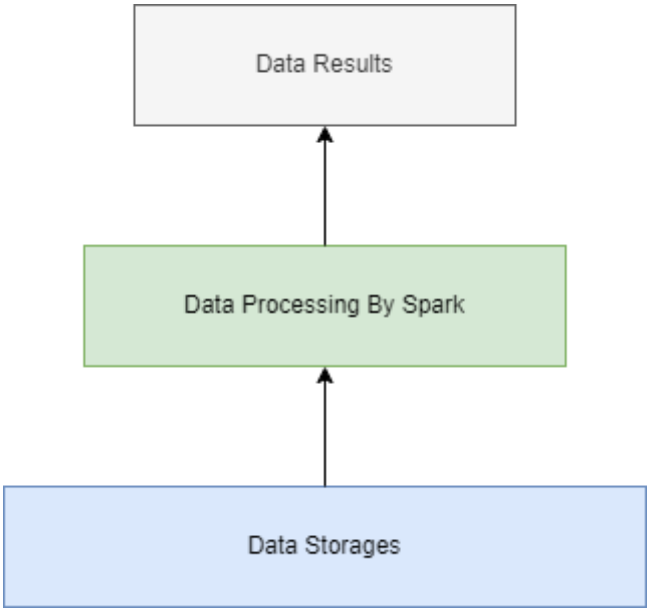
*Challenges in Big Data*

One of the significant challenges in big data is integrating and ensuring the quality of data coming from multiple sources. Data from different origins can vary greatly in terms of format, structure, and quality. Ensuring that this data is accurate, consistent, and reliable is crucial for deriving

meaningful insights. Data cleaning and preprocessing are essential steps in the big data pipeline, involving tasks such as removing duplicates, handling missing values, and converting data into a usable format.

## 2.3 Apache Spark

[3] Apache Spark is an open-source distributed computing framework maintained by the Apache Software Foundation. It is specifically crafted to handle big data processing and analytics efficiently across clustered systems. Spark is celebrated for its speed and general-purpose nature, providing scalable and robust data processing capabilities that can swiftly manage large volumes of data.




At the previous schema shows how Apache Spark works, first retrieves the data from the storages, handles any processing flows and finally return the results.

One of Spark's key strengths lies in its ability to accelerate data processing through in-memory computing and various optimization techniques. This feature makes it particularly effective for iterative algorithms and interactive data analysis, where it often outperforms traditional disk-based systems in terms of speed. In-memory computing allows data to be processed in RAM, which significantly reduces the time required for data retrieval and manipulation, as opposed to reading from and writing to disk. This approach is particularly advantageous for applications that require repeated access to the same dataset, such as machine learning algorithms and real-time analytics.

Spark offers high-level APIs for several programming languages including Java, Scala, Python, and R, broadening its accessibility to a diverse group

of developers. This multi-language support ensures that developers can utilize Spark's powerful features without having to learn a new language, thus leveraging their existing skills. For instance, Python developers can use PySpark, the Python API for Spark, to perform data processing and analytics within the familiar Python environment. Similarly, Scala and Java developers can take advantage of Spark's native APIs in their respective languages.

The framework also incorporates support for SQL queries, allowing for seamless structured data processing. Spark SQL, a component of Apache Spark, provides a programming interface for working with structured data. It allows users to run SQL queries on Spark data, seamlessly integrating SQL querying capabilities with Spark's data processing power. This integration makes it possible to perform complex queries on large datasets, enabling data analysts to use their SQL skills to analyze data stored in Spark.

The framework is versatile in supporting a wide array of data processing tasks such as batch processing, interactive queries, streaming analytics, machine learning, and graph processing. For these functions, Spark includes several specialized libraries like Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for real-time data streaming.

Spark SQL is specifically designed for working with structured and semi-structured data. It allows users to load data from various sources, such as JSON, Parquet, and Hive, and then query this data using SQL. This functionality enables data engineers and analysts to leverage their SQL knowledge while benefiting from Spark's scalability and performance.

MLlib is Spark's scalable machine learning library, offering a wide range of machine learning algorithms and utilities. It includes tools for classification, regression, clustering, collaborative filtering, and dimensionality reduction. MLlib also provides functionality for feature extraction and transformation, which are essential steps in preparing data for machine learning models. By integrating machine learning capabilities directly into the Spark ecosystem, MLlib allows data scientists to build and deploy machine learning models on large datasets without needing to transfer data between different systems.

GraphX is Spark's API for graph processing, enabling users to perform graph analytics on large datasets. GraphX provides a set of operators for manipulating graphs and a library of common graph algorithms, such as PageRank and connected components. This API allows users to build, transform, and query graphs, making it suitable for applications that involve complex relationships between data points, such as social network analysis and recommendation systems.

Spark Streaming enables real-time data processing and analytics, allowing users to process live data streams from sources like Kafka, Flume, and socket connections. This component is particularly useful for applications that require immediate insights from data as it arrives, such as monitoring systems, real-time dashboards, and fraud detection.

Due to its powerful and flexible capabilities, Apache Spark is extensively utilized across various industries for large-scale data processing and analytics. This widespread adoption underscores its utility in managing diverse big data workloads efficiently. Industries such as finance, healthcare, retail, and telecommunications leverage Spark to gain insights from their vast datasets and make data-driven decisions.

In practical applications, Spark is often employed for batch data processing, where it can manage comprehensive tasks such as ETL (Extract, Transform, Load) operations, data cleansing, and preparing data for analytics. ETL processes involve extracting data from various sources, transforming it into a suitable format, and loading it into a data warehouse or database. Spark's ability to handle large volumes of data and perform complex transformations makes it ideal for these tasks.

Its in-memory processing prowess also makes it ideal for interactive data analysis, enabling data scientists and analysts to conduct exploratory data analysis, execute SQL-like queries on large datasets, and interactively derive insights using Spark SQL. This interactive capability allows users to quickly iterate on their analyses, testing hypotheses and refining their models in real-time.

Furthermore, Spark's MLlib library offers a robust platform for developing scalable machine learning algorithms, facilitating tasks such as classification, regression, clustering, and collaborative filtering. These capabilities allow for the creation of sophisticated machine learning

models that can handle large datasets, which are particularly beneficial in building recommendation systems. Such systems are crucial in e-commerce and content streaming services, helping to personalize user experiences by suggesting products or content based on individual preferences and behaviors.

For instance, an e-commerce company might use Spark to analyze customer purchase history and browsing behavior to recommend products that are likely to interest individual users. Similarly, a streaming service might analyze viewing patterns to suggest movies or shows that align with a user's tastes. These personalized recommendations enhance user satisfaction and engagement, driving business growth.

In addition to its core components, Spark integrates well with other big data tools and frameworks. For example, it can work with Hadoop's HDFS (Hadoop Distributed File System) for storage, and it can run on Hadoop YARN (Yet Another Resource Negotiator) for resource management. This compatibility allows organizations to incorporate Spark into their existing big data ecosystems without requiring significant changes to their infrastructure.

Moreover, Spark's active community and continuous development ensure that it remains at the forefront of big data processing technology. The Apache Software Foundation and contributors from around the world regularly update Spark with new features, improvements, and bug fixes, ensuring that it continues to meet the evolving needs of data processing and analytics.

In conclusion, Apache Spark is a powerful and versatile framework for big data processing and analytics. Its speed, scalability, and support for a wide range of data processing tasks make it an essential tool for organizations dealing with large datasets. By leveraging Spark's capabilities, businesses can gain valuable insights, improve operational efficiency, and stay competitive in a data-driven world.

## 2.4 NoSQL Data Access Operators

The NoDA paper highlights the growing use of NoSQL stores for handling large spatio-temporal datasets but points out their limitations: lack of optimized indexing methods and diverse query languages. To address these challenges, the authors introduce NODA (NoSQL Data Access Operators), a system that abstracts data access across different NoSQL stores and provides a unified, SQL-like querying interface. Experimental results demonstrate significant performance improvements in spatio-temporal queries across various NoSQL systems, sets the stage by discussing the popularity and advantages of NoSQL stores in modern applications, emphasizing their scalability, flexible data models, and high availability. Despite these benefits, the authors identify two primary limitations when dealing with spatio-temporal data: inadequate support for optimized indexing methods and the lack of standardization in data access languages. These limitations complicate application development and hinder seamless transitions between different NoSQL stores, motivating the need for a unified solution.

The NODA abstraction layer is designed to provide a unified and simplified interface for accessing different NoSQL stores. It consists of a programming API with basic data access operators and a SQL interface that allows users to query NoSQL stores using a familiar SQL-like language. This approach hides the complexities of individual NoSQL store query languages from developers.

The API allows developers to chain method calls to build complex queries, separating transformations (definition phase) from actions (execution phase). This design, inspired by Apache Spark, enables lazy execution, where operations are only performed when an action is invoked, improving efficiency and usability.

Some implementation aspects are:

- Document-oriented Stores (MongoDB):

NODA utilizes MongoDB's aggregation pipeline and geospatial indexing, adapting these features to support spatio-temporal queries with custom partitioning using Hilbert space-filling curves.

- Wide-column Stores (HBase):

NODA encodes spatio-temporal data in row keys using Geohash and timestamps, leveraging HBase filters for efficient querying.

- Key-value Stores (Redis):

NODA stores spatio-temporal data as key-value pairs with additional indexing using Hilbert values, implementing efficient querying with Lua scripts for server-side operations.

NODA incorporates a SQL-like query language that maps SQL queries to sequences of NODA operations. This module, implemented with ANTLR, parses SQL queries and translates them into the respective NoSQL store's query language, leveraging existing optimization capabilities.

NODA supports geo-operators for spatial and spatio-temporal data, converting complex operations like k-nearest neighbors (k-NN) into range queries using a QuadTree for radius estimation. This approach provides efficient querying by pre-computing spatial indices.

Some implementation details are:

- MongoDB:

Spatio-temporal data is modeled as documents with GeoJSON objects and Hilbert indices. NODA improves data locality and query performance by incorporating Hilbert values into the sharding key.

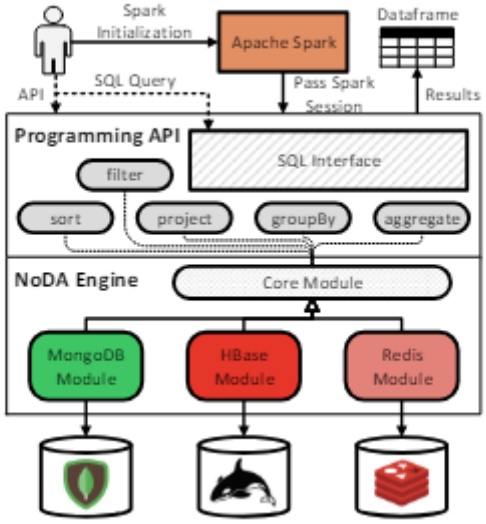
- HBase:

Spatio-temporal data is encoded in row keys using Geohash, with custom filters applied to efficiently process spatio-temporal queries using HBase's range scan capabilities.

- Redis:

Data is stored in key-value pairs with sorted sets indexed by Hilbert values, enabling efficient retrieval of spatio-temporal data through Lua scripts that handle complex query processing.

NODA maintains its performance advantages even as dataset sizes increase, demonstrating scalability. The study shows that NODA's performance gains are sustained with larger data sets, making it suitable for real-world applications with growing data volumes.




[1] The image above show us how NODA API works. The programming API offers operations like filtering, sorting, projection. The base mechanism of NODA has at top level the core module. Core module has all the defined behavior of the API. It actually implemented by Facade pattern. The Facade pattern is a design pattern that provides a simplified interface to a complex subsystem. This pattern involves creating a single class, called a facade, which hides the complexities of the underlying system and provides an easy-to-use interface for the client. The primary intent of the Facade pattern is to provide a unified and straightforward interface to a set of interfaces in a subsystem, make the subsystem easier to use by reducing the complexity exposed to the client and decouple the client from the subsystem, promoting loose coupling and easier maintenance. So the client through NODA API can easily communicate and interacting through many NoSQL databases without having to implement each system’s complexity client code. The client through a well defined programming API is interacting. After the completed interaction client can receive the results from the database through Apache Spark’s domain model Dataset.

# 3 Join Implementation Solution

The expanded NoDA API leverages the Strategy Pattern within Apache Spark's Dataset API to enhance the flexibility and scalability of join operations across various NoSQL databases. This design encapsulates each join type (e.g., inner, left, right, outer) within its own class, promoting modularity, ease of maintenance, and adherence to the single responsibility principle. The Strategy Pattern allows for dynamic selection of join strategies at runtime, optimizing performance and resource utilization in distributed computing environments. By defining a common interface and implementing specific join strategies as subclasses, the NoDA API integrates seamlessly with Spark's functional programming paradigms, enabling efficient data manipulations and transformations. The approach addresses challenges such as schema flexibility, data distribution, redundancy, and consistency by incorporating schema inference, distributed computing optimizations, and data reconciliation techniques. Performance is further enhanced through indexing, partitioning, caching, and cost-based optimization strategies. This thesis demonstrates how the Strategy Pattern improves the adaptability, maintainability, and scalability of the NoDA API, providing a robust solution for complex data integration and analytics in big data environments.

Moreover a custom join implementation is presented. The hash join algorithm provides an efficient method for performing join operations, particularly useful in scenarios where external libraries are not viable. Implemented within the NoDA API, it offers a direct and optimized approach to handling joins internally. The algorithm consists of four phases: partitioning, building, probing, and joining. Data sets are divided into smaller subsets using a hash function on the join attribute, reducing comparisons by localizing searches within buckets. Hash tables are constructed for these buckets, typically using the smaller input table to minimize memory usage, enabling rapid data access. The second input table is processed using the same hash function to efficiently search for matching tuples in the hash tables. Matching tuples are merged to form the output, ensuring accurate and relevant results. Implementing the hash join algorithm within NoDA without external libraries enhances control over performance and optimization, ensuring robustness and

stability. This self-contained approach allows for custom optimizations, delivering superior performance tailored to specific application needs.

## 3.1 NoSQL Join

NoSQL databases are tailored to manage large data volumes and are distinguished by their flexible schema arrangements. They typically emphasize horizontal scalability and performance, often at the expense of complex query capabilities such as join operations. This design choice is fundamental to the architecture of NoSQL systems and reflects their primary focus on handling large-scale, distributed data efficiently.

While different NoSQL databases exhibit unique strengths and challenges, it's notable that not all NoSQL types completely forego join operations. For instance, graph databases, a subset of NoSQL, inherently support queries that involve traversing relationships between nodes, which is similar to performing joins. This capability makes graph databases particularly suited for applications that require the exploration of complex relationships, such as social networks, recommendation engines, and fraud detection systems.

The implementation of join operations in NoSQL databases faces several challenges:

Flexible or Schema-less Design: NoSQL databases allow each record to have varying fields or attributes, which complicates the establishment of fixed relationships between entities. This contrasts with the structured schema of relational databases, where relationships are clearly defined and consistently maintained across records. The schema-less nature of NoSQL databases provides greater flexibility and adaptability, but it also introduces complexity when trying to implement join operations that rely on consistent and predictable data structures.

Horizontal Scaling and Distribution: NoSQL systems are built to scale out by distributing data across multiple nodes or partitions. Join operations, which often require data from multiple partitions, pose a challenge in such distributed environments. Efficiently executing joins without impacting performance significantly is complex and often requires

sophisticated optimization strategies. The need to retrieve and combine data from different nodes can lead to increased latency and reduced performance, undermining the scalability benefits that NoSQL databases aim to provide.

Denormalization and Data Redundancy: To optimize read performance, NoSQL databases frequently employ denormalization, where data duplication across documents or records is common. Although this can enhance read efficiency, it complicates maintaining data consistency during updates and adds complexity to join operations. Denormalization allows for faster read operations by reducing the need to perform joins, but it also means that data updates must be carefully managed to ensure consistency across duplicated records.

Optimized for Specific Use Cases: NoSQL databases are optimized for particular scenarios like high read and write throughput, prioritizing these over complex query capabilities. Introducing join operations in such environments can lead to significant performance overhead, detracting from the primary use case efficiencies. For example, a key-value store designed for rapid read and write operations may suffer performance degradation if complex join queries are introduced, as these queries are not aligned with the database's core design principles.

Diverse Data Models: The various data models used in NoSQL databases (e.g., document-oriented, key-value, column-family, and graph) each have their strengths and limitations. Some models, like graph databases, more naturally support join-like operations, whereas others may find implementing joins more cumbersome. Document-oriented databases like MongoDB, for instance, can struggle with complex joins due to their nested and hierarchical data structures, which do not lend themselves easily to relational-style joins.

Unique Query Languages and Tools: The specialized query languages and tools developed for NoSQL databases may not support joins as comprehensively as SQL does in relational systems. This limitation makes it harder to write complex queries that involve multiple collections or tables. While SQL provides a robust and standardized way to perform joins, NoSQL query languages often prioritize simplicity and performance for common use cases, resulting in less support for complex joins.

Despite these challenges, certain NoSQL databases do offer mechanisms to handle or simulate joins. For example, MongoDB provides the $lookup aggregation stage, which allows for a form of left outer join between two collections. However, such features are typically less efficient and expressive compared to traditional SQL joins. The $lookup operation in MongoDB can be useful for certain scenarios, but it often comes with performance trade-offs and limitations in terms of the complexity and depth of joins it can handle.

In conclusion, the architecture of NoSQL databases generally favors scalability, flexibility, and performance for specific use cases over the complexity of join operations. Developers using NoSQL typically design their data models and query strategies to align with the specific requirements and limitations of the database, minimizing reliance on traditional join operations to maintain performance and scalability. Instead, they often employ techniques such as denormalization, embedding, and pre-computed views to achieve similar results without the overhead of joins.

Moreover, the choice of NoSQL database and data modeling approach should be guided by the specific needs of the application. For instance, a graph database might be the best choice for an application that requires extensive relationship traversal, while a document-oriented database might be more suitable for a content management system with flexible data structures. Understanding the strengths and limitations of each NoSQL model is crucial for designing efficient and scalable applications.

As NoSQL databases continue to evolve, we may see further improvements in their ability to handle complex queries, including joins. Advances in distributed computing, query optimization, and data indexing are likely to enhance the capabilities of NoSQL databases, making them even more versatile and powerful for a wide range of applications.

In the meantime, developers must carefully consider the trade-offs involved in using NoSQL databases and design their systems to take full advantage of the performance and scalability benefits while mitigating the challenges associated with complex query operations. By doing so, they can harness the full potential of NoSQL databases to build robust, high-performance applications that meet the demands of modern data-driven environments.

## 3.2 Different Types of JOINs in SQL

[2] In relational database systems, join operations are critical for combining and manipulating data from multiple tables based on defined relationships. These operations allow for the integration of data stored in different tables, enabling comprehensive data analysis and retrieval. There are various types of join operations, each serving distinct purposes and providing different views of the data.

Inner Joins: Inner joins return only the matching tuples from both tables, ensuring an intersection of the datasets. This type of join is useful when you need to retrieve records that have corresponding values in both tables. For example, if you have a table of customers and a table of orders, an inner join can be used to find only those customers who have placed orders. This ensures that the result set contains only the related recors from both tables.

Left Outer Joins: Left outer joins preserve all tuples from the left table, complementing them with matching tuples from the right table or filling in NULLs where no match exists. This type of join is helpful when you need to retrieve all records from the left table regardless of whether there is a matching record in the right table. For instance, if you want to list all customers and their orders, including those customers who have not placed any orders, a left outer join would be appropriate.

Right Outer Joins: Right outer joins retain all tuples from the right table, similarly augmenting them with corresponding matches from the left table or NULLs for non-matching entries. This join is the mirror image of the left outer join and is useful when you need all records from the right table and the corresponding matches from the left table. For example, to list all products and their suppliers, including those products that do not have a supplier listed, a right outer join can be used.

Full Outer Joins: Full outer joins combine the effects of both left and right joins, returning all tuples from both tables with appropriate NULLs where matches are absent. This type of join is useful when you need a complete view of both tables, including all matched and unmatched records. For example, to create a comprehensive list of all employees and departments, showing all employees even if they are not assigned to a

department and all departments even if they have no employees, a full outer join would be the best choice.

Cross Joins: Cross joins generate the Cartesian product of the tables, offering all possible tuple combinations. This join is used when you need to combine each row from the first table with each row from the second table, resulting in a large number of possible combinations. Cross joins can be useful in scenarios such as generating test data or creating all possible pairs of items from two different lists.

Self Joins: Self joins enable a table to be joined with itself, useful for hierarchical or recursive data relationships. This join is particularly beneficial when dealing with data that has a hierarchical structure, such as organizational charts or category trees. For example, in an employee table, a self join can be used to find the relationship between employees and their managers by joining the table on itself based on the manager ID.

These join operations form the backbone of complex queries, facilitating comprehensive data analysis and retrieval in relational databases. By using joins, users can combine data from different tables to create more meaningful and informative results. This capability is essential for tasks such as generating reports, analyzing trends, and making data-driven decisions.

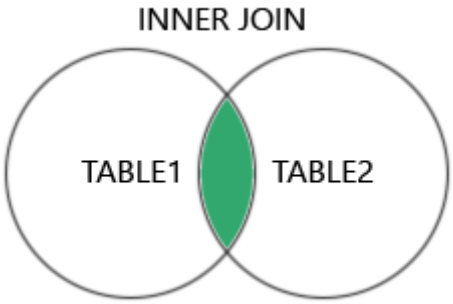
Join operations also play a crucial role in maintaining data integrity and consistency in relational databases. By defining relationships between tables and using joins to enforce these relationships, databases can ensure that related data remains accurate and consistent. For instance, foreign key constraints rely on joins to validate that every foreign key value in a child table has a corresponding primary key value in the parent table.

Additionally, understanding and effectively utilizing join operations can significantly improve query performance. Optimizing joins through indexing and query planning can lead to faster and more efficient data retrieval, which is particularly important for large datasets and complex queries.

In conclusion, join operations are fundamental to relational database systems, providing the means to combine and manipulate data across multiple tables. Whether performing simple data retrieval or complex data analysis, joins are indispensable tools for database users, enabling them to unlock the full potential of their data. By mastering the various types of joins and their appropriate use cases, users can enhance their ability to work with relational databases and achieve more robust and insightful data analysis.

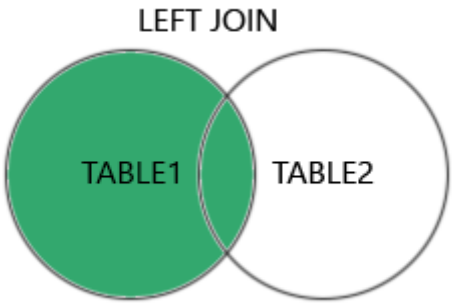
### 3.2.1 Inner Join

An inner join is a type of join operation that returns the intersection of two tables, specifically the set of tuples that have matching values in the columns specified in the join condition. This join filters out any rows that do not satisfy the join condition, ensuring that only tuples present in both relations are included in the result set. Mathematically, this can be expressed as a selection operation over the Cartesian product of the two tables, where the selection criterion is the equality of the specified columns.



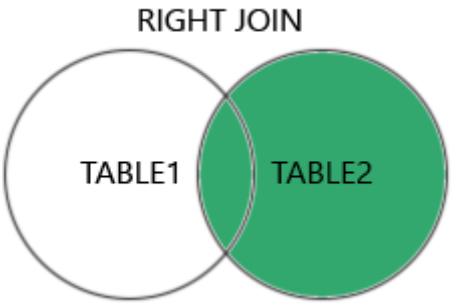

### 3.2.2 Left Outer Join

A left join or left outer join returns all tuples from the left table (the first relation), along with the matching tuples from the right table (the second relation). If no matching tuple exists in the right table, the resulting tuple in the output will contain NULL values for all attributes from the right table. This join operation preserves the non-matching tuples from the left table, making it a form of extended projection that includes tuples that meet the join condition and those that do not.



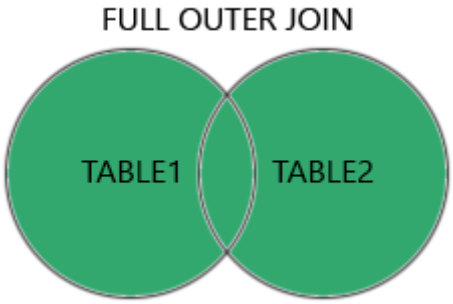
### 3.2.3 Right Outer Join

A right join or right outer join is analogous to a left join, but it returns all tuples from the right table, along with the matching tuples from the left table. In the absence of matching tuples in the left table, the resulting tuples will contain NULL values for all attributes from the left table. This operation ensures that all tuples from the right table are represented in the result set, regardless of whether they have corresponding matches in the left table.



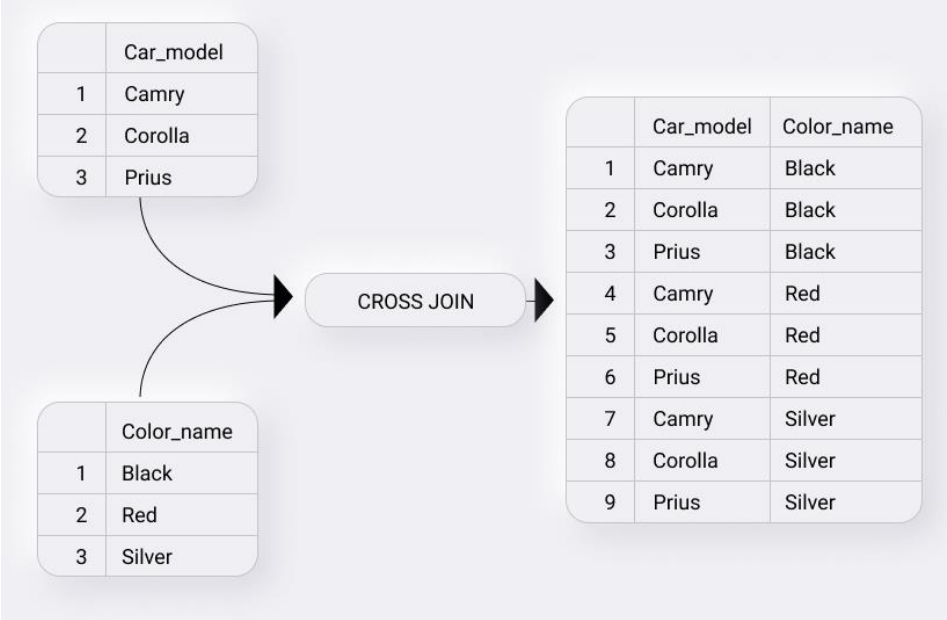

### 3.2.4 Full Outer Join

A full join or full outer join combines the properties of both left and right outer joins. It returns all tuples from both tables, along with the matching tuples where available. For tuples with no corresponding match in the opposite table, the resulting tuple will contain NULL values for the attributes of the non-matching table. This join operation generates a complete set of combined tuples, encompassing all tuples from both relations and filling in NULLs where necessary.



### 3.2.5 Cross Join

A cross join produces the Cartesian product of the two tables, generating a result set that includes all possible combinations of tuples from the two relations. The resulting set has a size equal to the product of the sizes of the two tables. Cross joins do not use any join condition and are typically used in scenarios where a combinatorial pairing of tuples is required.




The image shows us an example of a cross join result.

### 3.2.6 Self Join

A self join is a join operation where a table is joined with itself. This operation is used to relate tuples within the same table based on a specified condition, often to establish relationships among rows within the same relation. In a self join, the table is conceptually treated as two separate relations with distinct aliases to facilitate the join operation. This type of join is particularly useful in hierarchical or recursive data structures.

These join operations are fundamental in relational algebra and are essential for querying and manipulating relational data. Each type of join operation serves a specific purpose, allowing for the flexible and

comprehensive combination of data from multiple relations based on specified criteria.

**Employee Table**

| EmployeeID | LastName | Country | DepartmentID |
|---|---|---|---|
| 123 | Rafferty | Australia | 31 |
| 124 | Jones | Australia | 33 |
| 145 | Heisenberg | Australia | 33 |
| 201 | Robinson | United States | 34 |
| 305 | Smith | Germany | 34 |
| 306 | Williams | Germany | NULL |

**Employee Table after Self-join by Country**

| EmployeeID | LastName | EmployeeID | LastName | Country |
|---|---|---|---|---|
| 123 | Rafferty | 124 | Jones | Australia |
| 123 | Rafferty | 145 | Heisenberg | Australia |
| 124 | Jones | 145 | Heisenberg | Australia |
| 305 | Smith | 306 | Williams | Germany |

The image shows us the result of a self join operation.

## 3.3 Solution Implementation

In this thesis, we address the complexities and challenges associated with performing join operations across different NoSQL databases by leveraging and expanding the NoDA API. Our objective is to develop a join mechanism capable of integrating results fetched from various databases into a cohesive dataset. This effort aims to bridge the gap between the flexibility of NoSQL databases and the sophisticated query capabilities traditionally found in relational databases, enhancing the ability to perform complex data integration tasks in diverse data environments.

The NoDA API, already adept at extracting results from diverse database operations into a structured format, sets a solid foundation for our enhancements. Our primary goal with this expansion is to integrate these results using a join mechanism, allowing data from different sources to seamlessly merge based on specified criteria. This involves extending the existing capabilities of NoDA to support more complex query operations and to facilitate the combination of data across different NoSQL databases.

The traditional NoDA API supports basic CRUD (Create, Read, Update, Delete) operations across various NoSQL databases. However, the need to perform join operations across these heterogeneous data sources necessitates a significant enhancement of its functionalities. Our approach involves introducing new methods and data structures that can handle the complexities of join operations, including inner joins, outer joins, left joins, and right joins. By expanding the NoDA API, we aim to provide developers with a more robust and versatile toolset for managing and integrating data from multiple NoSQL databases.

Apache Spark's Dataset API serves as a pivotal component in our strategy. Datasets in Spark are distributed collections of data, which combine the benefits of strong typing from traditional databases with the functional programming model of Spark. This dual advantage facilitates efficient data manipulation and ensures type safety, reducing runtime errors significantly.

Spark Datasets are built on top of Spark's SQL engine, making them optimized for complex data operations through the Catalyst optimizer. This optimization can lead to substantial performance improvements over the basic RDDs (Resilient Distributed Datasets), particularly in structured and semi-structured data scenarios. The Catalyst optimizer is a powerful query optimization framework that translates high-level queries into efficient execution plans, leveraging advanced techniques such as predicate pushdown, column pruning, and query rewriting to enhance performance.

By integrating Spark's Dataset API with the expanded NoDA API, we aim to leverage these performance optimizations and the rich set of features provided by Spark. This integration enables us to perform complex join operations across different NoSQL databases while benefiting from the scalability and efficiency of Spark's distributed computing capabilities. Additionally, Spark's support for a wide range of data sources, including Hadoop, HDFS, and various NoSQL databases, makes it an ideal platform for our enhanced NoDA API.

Join operations in Spark’s Dataset API are crucial for correlating information from disparate datasets. These operations, akin to those in traditional relational databases, include:

Inner Join: Returns rows where there is a match in both datasets. This type of join is useful for finding commonalities between two datasets, ensuring that only the records that have corresponding values in both datasets are included in the result.

Outer Join: Includes full outer, left outer, and right outer joins, and can fill in null values for unmatched columns, depending on the specific type of outer join used. Outer joins are useful for preserving all records from one or both datasets, even when there is no match, providing a more comprehensive view of the data.

Left Join: Ensures all rows from the left dataset appear in the result, along with matching rows from the right dataset. This join is particularly useful when you want to retain all records from the primary dataset and only include related records from the secondary dataset.

Right Join: Ensures all rows from the right dataset are included, along with matches from the left dataset. This join is the mirror image of the left join and is useful when the right dataset is the primary focus.

Each join type is designed to cater to specific data integration needs, allowing for flexibility in how data is combined and analyzed. These join operations are fundamental for integrating data from multiple sources and enabling complex data analysis.

To implement a join, both datasets must share a common key, typically a column or a set of columns, where the values match. This key forms the basis of the join, enabling the alignment and combination of data entries from the different datasets. The common key ensures that the joined datasets are correctly aligned, preserving the relationships between the data points.

The practical implementation of this expanded API involves formulating and executing join operations within the NoDA framework, translating these operations into Spark's Dataset API. This approach not only leverages Spark’s powerful data processing capabilities but also aligns with the functional programming paradigm that Spark supports, facilitating complex data manipulations in a more intuitive manner. By utilizing Spark's Dataset API, we can take advantage of its robust support for various data formats, its efficient query optimization, and its scalability across distributed computing environments.

In addition to the basic join operations, our implementation also includes support for more advanced data integration techniques. These include handling nested and hierarchical data structures, performing multi-way joins involving more than two datasets, and implementing custom join logic to address specific application requirements. By providing a comprehensive set of join capabilities, our enhanced NoDA API offers greater flexibility and power for developers working with complex data integration tasks.

Implementing join operations across different NoSQL databases presents several challenges, including:

Schema Flexibility: NoSQL databases often use flexible or schema-less designs, where records can have varying fields or attributes. This flexibility complicates the establishment of fixed relationships between entities,

making it challenging to define and execute join operations. To address this, our implementation includes schema inference techniques that automatically determine the structure of the data and identify common keys for joining.

Data Distribution: NoSQL databases are designed for horizontal scaling, distributing data across multiple nodes or partitions. Join operations, which require data from multiple partitions, pose a challenge in such distributed environments. Efficiently executing joins without impacting performance significantly is complex and often requires sophisticated optimization strategies. We leverage Spark's Catalyst optimizer and its support for distributed computing to efficiently execute join operations across distributed datasets.

Data Redundancy and Consistency: To optimize read performance, NoSQL databases frequently employ denormalization, where data duplication across documents or records is common. This complicates maintaining data consistency during updates and adds complexity to join operations. Our approach includes data reconciliation techniques that ensure consistency across joined datasets and handle potential conflicts arising from data redundancy.

Performance Optimization: Join operations can be resource-intensive and impact performance, especially when dealing with large datasets. Our implementation includes various performance optimization techniques, such as indexing, partitioning, and caching, to enhance the efficiency of join operations. By optimizing the execution plans and minimizing data movement, we aim to achieve high performance and scalability.

By expanding the NoDA API to incorporate Spark’s Dataset join capabilities, this thesis provides a robust solution for performing complex join operations across diverse NoSQL databases. This approach significantly enhances the ability to analyze and integrate data from multiple sources, providing a powerful tool for developers working in environments where data is not only large-scale but also originates from heterogeneous NoSQL databases.

The enhanced NoDA API offers a more flexible, efficient, and effective means of data processing and analytics in distributed computing environments. It bridges the gap between the flexible data models of

NoSQL databases and the sophisticated query capabilities traditionally found in relational databases, enabling developers to perform complex data integration tasks with ease.

Furthermore, the integration with Apache Spark's Dataset API ensures that our solution leverages the latest advancements in distributed computing and query optimization. Spark's powerful data processing capabilities, combined with the flexibility and scalability of NoSQL databases, provide a comprehensive platform for modern data analytics.

In summary, the expanded NoDA API represents a significant advancement in the field of NoSQL data integration. By providing robust support for join operations and leveraging the capabilities of Apache Spark, our solution enables developers to tackle complex data integration challenges and unlock the full potential of their data. This thesis contributes to the ongoing evolution of NoSQL databases and distributed computing, offering new possibilities for data processing and analytics in the era of big data.

## 3.4 Apache Spark Solution

The implementation of the join mechanism in the expanded NoDA API, utilizing the Dataset object from the Apache Spark framework, effectively employs the Strategy Pattern—a versatile behavioral design pattern. This design approach plays a critical role in enhancing the flexibility and scalability of the application, especially in the context of handling various types of join operations across different NoSQL databases.

By incorporating the Strategy Pattern, each join operation (such as inner join, left join, right join, and outer join) is encapsulated within its own class. This encapsulation not only isolates the specific behavior of each join type but also simplifies modifications and maintenance. For instance, if the behavior of the left join needs to be adjusted, only the class representing the left join needs to be updated, without affecting other types of joins. This modularity allows for easier debugging and testing, as changes in one join strategy do not ripple across the entire codebase. Additionally, it promotes the single responsibility principle, ensuring that each class has a clear and focused purpose.

A key feature of the Strategy Pattern is the ease with which algorithms can be swapped at runtime without altering the client code. In the context of the NoDA API, this means that depending on the data requirements and the specific conditions of the query, different join strategies can be dynamically selected and executed. This is particularly beneficial in a distributed computing environment like Spark, where data characteristics and system performance can vary greatly.

The Strategy Pattern allows the NoDA API to adapt to various runtime scenarios dynamically. Since the join operations are implemented as separate strategies, the system can select the most appropriate join type based on the data distribution, size, and the specific requirements of the operation. This flexibility is crucial for optimizing performance and resource utilization in big data environments. For example, during peak data loads, the system might choose a more efficient join strategy to maintain performance levels, or it might switch to a different strategy when the data volume decreases or when specific query requirements change.

By separating the join strategies from the client code that invokes them, the NoDA API remains clean and focused on its core functionality, while the join operations can evolve independently. This separation of concerns not only enhances code readability and maintainability but also promotes a more modular architecture. Developers can work on improving or extending join strategies without impacting the main application logic, allowing for better manageability and scalability of the codebase.

In the Apache Spark's Dataset API, the application of the Strategy Pattern involves defining a common interface or a base class for the join operation, with multiple subclasses representing each specific type of join.

This design not only leverages Spark's robust data processing capabilities but also aligns with functional programming paradigms, allowing for complex data manipulations and transformations to be expressed more succinctly and clearly. Spark's Dataset API, which builds on the DataFrame API, provides a powerful abstraction for working with structured and semi-structured data. By using Datasets, developers can benefit from both the expressiveness of functional programming and the performance optimizations of Spark's Catalyst optimizer.

To implement a join, both datasets must share a common key, typically a column or a set of columns, where the values match. This key forms the basis of the join, enabling the alignment and combination of data entries from the different datasets. The common key ensures that the joined datasets are correctly aligned, preserving the relationships between the data points.

The practical implementation of this expanded API involves formulating and executing join operations within the NoDA framework, translating these operations into Spark's Dataset API. This approach not only leverages Spark’s powerful data processing capabilities but also aligns with the functional programming paradigm that Spark supports, facilitating complex data manipulations in a more intuitive manner.

In addition to the basic join operations, our implementation also includes support for more advanced data integration techniques. These include handling nested and hierarchical data structures, performing multi-way joins involving more than two datasets, and implementing custom join

logic to address specific application requirements. By providing a comprehensive set of join capabilities, our enhanced NoDA API offers greater flexibility and power for developers working with complex data integration tasks.

Join operations can be resource-intensive and impact performance, especially when dealing with large datasets. Our implementation includes various performance optimization techniques, such as indexing, partitioning, and caching, to enhance the efficiency of join operations. By optimizing the execution plans and minimizing data movement, we aim to achieve high performance and scalability. Additionally, we employ cost-based optimization strategies to select the most efficient join plan based on runtime statistics and historical performance data.

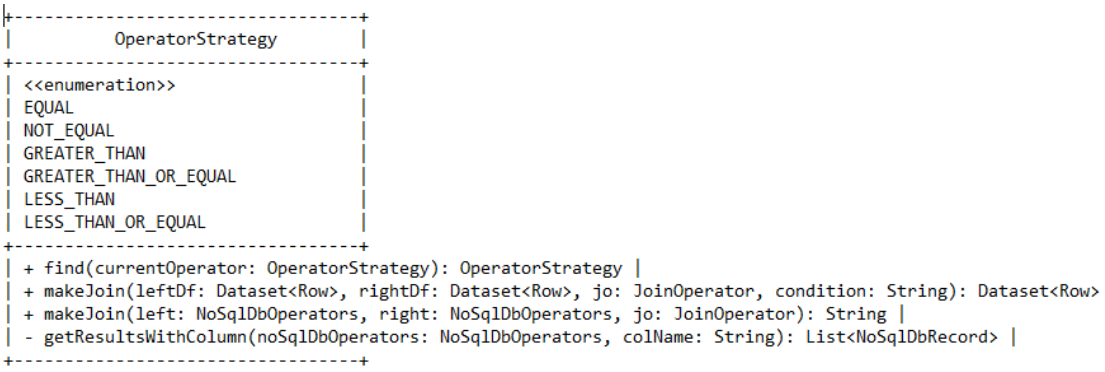
Employing the Strategy Pattern in the development of the NoDA API's join mechanism significantly enhances the system's adaptability and maintainability. It provides a clear structure for handling various join operations, offers the flexibility to adapt to different data and runtime conditions, and maintains a clean separation between the data processing logic and the underlying database interactions. This approach not only improves the efficiency of data integration tasks but also facilitates the scaling of data processing strategies in diverse and dynamic big data environments.

By expanding the NoDA API to incorporate Spark’s Dataset join capabilities, this thesis provides a robust solution for performing complex join operations across diverse NoSQL databases. This approach significantly enhances the ability to analyze and integrate data from multiple sources, providing a powerful tool for developers working in environments where data is not only large-scale but also originates from heterogeneous NoSQL databases. This enhancement to NoDA will offer a more flexible, efficient, and effective means of data processing and analytics in distributed computing environments.

At our implementation of the Strategy Pattern, we use Java enumeration. The enumeration is used to describe all the joins that the API from Dataset is given. There are operators for join cases of EQUAL, NOT_EQUAL, GREATER_THAN, GREATER_THAN_OR_EQUAL, LESS_THAN, and LESS_THAN_OR_EQUAL. All these members implement an abstract method to perform the operator needed above the two Datasets.

This strategic approach provides a robust framework for managing the complexities of join operations in NoSQL databases, ensuring that developers can build scalable, efficient, and maintainable data integration solutions. By leveraging the strengths of Apache Spark and the flexibility of the Strategy Pattern, the enhanced NoDA API represents a significant advancement in the field of distributed data processing and analytics.

The operators to be implemented are equal, not equal, greater, greater than, less, less than. For that reason the OperationStrategy enumeration was created. It contains as members all these join operators in order to be implemented by each member. Below there is a class diagram for this enumeration.




Each member overrides the makeJoin method to provide specific join implementation.

It has three public method and one private. The find method is for finding the correct enumeration member on the fly. The makeJoin method one is for join the Datasets by Spark and it returns Dataset with join result. The last one is overloaded for join with hash join algorithm and it returns String JSON with results.

The abstract method needs to be implemented is the above.

```
Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
JoinOperator jo, String condition)
```

The signature made for this method is the two Datasets that is going to be joined. The JoinOperator is the Object holds the information at its state for the columns of our Datasets that is common and finally the condition given as a String. Condition must be inner, right, left or outer.

### 3.4.1 The Equal Operator

This implementation of the abstract method is for return the common elements from two datasets according to common column they have.

```
EQUAL {
  @Override
  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
      JoinOperator jo, String condition) {
    return leftDf.join(rightDf,
        JavaConverters
            .asScalaIteratorConverter(
                Arrays.asList(jo.getColumnAName(), jo.getColumnBName())
                    .iterator())
            .asScala()
            .toSeq(),
        condition);
  }
}
```

### 3.4.2 The Negation of equality

This join is implemented with use of the operator of negation of equality. This means that will return the elements witch the common column given is not the same.

```
NOT_EQUAL {
  @Override
  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
      JoinOperator jo, String condition) {
    return leftDf.join(rightDf,
        leftDf.col(jo.getColumnAName())
            .$eq$bang$eq(rightDf.col(jo.getColumnBName())),
        condition);
  }
}
```

### 3.4.3 The Greater than operator

The implementation to achive the the handle of join two dataset to retrieve the elements that the common column from first dataset values are greater than the second's is the use of greater than scala operator *$greater* for join dataset objects.

```
GREATER_THAN {
  @Override
  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
      JoinOperator jo, String condition) {
    return leftDf.join(rightDf,
        leftDf.col(jo.getColumnAName())
            .$greater(rightDf.col(jo.getColumnBName())),
        condition);
  }
}
```

### 3.4.4 The Greater than or Equals

This join implementation is performed as the previously method implementation just with a little different operator to perform the equality also. The operator used is the scala operator for greater than or equals that gives the API of Dataset join *$greater$eq.*

```
GREATER_THAN {
  @Override
  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
      JoinOperator jo, String condition) {
    return leftDf.join(rightDf,
        leftDf.col(jo.getColumnAName())
            .$greater(rightDf.col(jo.getColumnBName())),
        condition);
  }
}
```

### 3.4.5 The Less operator

The less join handling is performed among two Dataset objects with the operator *$less.* The result of a join with this operator is returning a new Dataset object with all all elements that the common column values from first dataset is less than the second's.

```
GREATER_THAN {
  @Override
  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,
      JoinOperator jo, String condition) {
    return leftDf.join(rightDf,
        leftDf.col(jo.getColumnAName())
            .$greater(rightDf.col(jo.getColumnBName())),
        condition);
  }
}
```

### 3.4.6 The Less Than Or Equals

This join implementation is performed as the previously method implementation just with a little different operator to perform the equality also. The operator used is the scala operator for greater than or equals that gives the API of Dataset join *$less$eq.*

```
LESS_THAN_OR_EQUAL {

  @Override

  public Dataset<Row> makeJoin(Dataset<Row> leftDf, Dataset<Row> rightDf,

      JoinOperator jo, String condition) {

    return leftDf.join(rightDf,

        leftDf.col(jo.getColumnAName())

            .$less$eq(rightDf.col(jo.getColumnBName())),

        condition);

  }

}
```

### 3.4.7 The Join Operators

In order to handle the appropriate operator for a join between the two Datasets, have to be implemented the JoinOperator class for each of the join cases. The classes named with EqualJoinOperator,NotEqualJoinOperator,GreaterThanJoinOperator,GreaterThanOrEqualsJoinOperator,LowerThanJoinOperator,LowerThanOrEqualsJoinOperator.

All these implementations implementation are used as parameters to the public method that NoSqlDbOperators abstract class is enriched.

### 3.3.8 The client view

The application of the builder pattern to the NoDA API, particularly in the context of the NoSqlDbOperators class, is a strategic enhancement that greatly facilitates the composition and execution of operations on NoSQL databases. By integrating a join method that supports chaining with other builder methods, this approach effectively streamlines complex query constructions and execution.

*Builder Pattern in NoDA API*

1. Extending Functionality with Join Method:

The NoDA API's builder pattern allows for the fluent chaining of methods, enhancing readability and ease of use. The join method is a significant addition to this pattern. It accepts a reference to another instance of NoSqlDbOperators, indicating a join operation should be performed between the two. This method sets up the necessary parameters for the join but defers execution until the toDataset method is called.

### 2. State Management:

Upon invoking the join method, the state of the NoSqlDbOperators class is updated to store the reference of the second NoSqlDbOperators instance. Additionally, a flag is set to indicate that a join operation is pending. This state management ensures that all the configured operations, including any joins, are only executed when toDataset is finally called.

### 3. Flexible API Usage:

Clients can continue to use all builder methods as before, such as setting filters, projections, and sorting. The API maintains its versatility by

allowing these operations to be defined or modified even after setting up a join. This flexibility ensures that the API can handle a variety of use cases without restricting the order in which operations are defined.

### 4. Execution with toDataset:

The actual data processing and joining of datasets occur when the toDataset method is invoked. This method executes all the configured operations, including any joins, and returns the results in a Spark Dataset object. This delayed execution helps optimize resource use and performance, as operations are batched and run efficiently at once.

The integration of the builder and strategy patterns in the NoDA API provides a powerful, flexible, and intuitive means for developers to work with complex data operations across NoSQL databases. By allowing for dynamic, runtime configuration of operations and efficient execution, the NoDA API helps streamline the development process and enhances the capabilities of applications dealing with large and diverse datasets. This methodological approach ensures that developers can maintain clean and manageable code while leveraging advanced data processing techniques.

| countryName | countryName | shipLength | sourcemmsi | shiptype |
|---|---|---|---|---|
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Military Ops |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Sailing Vessel |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Dredger |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Tanker - Hazard A... |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo - Hazard A ... |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Dredger |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Undefined Type |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Tanker |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Tanker |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Tanker |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Cargo |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Sailing Vessel |
| Belgium | Belgium | 0.47237197895707284 | 205067000 | Sailing Vessel |
| Belgium | Belgium | 0.47237197895707284 | 205204000 | Military Ops |
| Belgium | Belgium | 0.47237197895707284 | 205204000 | Sailing Vessel |

We see the common columns from two datasets, the shipLength is from second dataset, the sourcemmsi and shiptype from the first.

## 3.4 Hash Join Implementation

The hash join algorithm represents an efficient approach to performing joins, particularly useful in scenarios where using external libraries is not viable. By leveraging its own mechanism for inner joins, the implementation of a hash join algorithm within the NoDA API exemplifies a direct and optimized method to handle join operations internally. This method allows for significant control and optimization, making it a powerful tool for database management.

1. Partitioning Phase

The initial step in the hash join algorithm involves partitioning the data sets. Both input tables are divided into smaller subsets or buckets using a hash function that targets the join attribute—the column or field on which the join will be based. This step is crucial as it reduces the number of comparisons needed by localizing the search for matches to within these buckets rather than across the entire data set. Partitioning helps to manage the data more efficiently, especially when dealing with large datasets that might not fit into memory all at once. By breaking down the data into manageable chunks, the algorithm ensures that the subsequent phases can be executed swiftly and effectively.

2. Building Phase

Once the data is partitioned, the next step is to construct hash tables for these buckets. Typically, this involves selecting one of the input tables (often the smaller one, to minimize memory usage) to create a hash table for each of its buckets. The hash table stores all tuples from this table, indexed by the hash values of the join attribute. This structure allows for rapid access to the data, which is essential for the next phase. The building phase focuses on creating a compact and efficient representation of the data, ensuring that it can be accessed quickly during the probing phase. The choice of the smaller table is strategic, as it reduces the memory footprint and enhances performance.

3. Probing Phase

In the probing phase, the algorithm processes the second input table. It uses the same hash function to determine the corresponding bucket for each tuple based on its join attribute. By directly accessing the hash table

of the relevant bucket, the algorithm efficiently searches for matching tuples. This targeted searching significantly speeds up the join process. The probing phase is where the actual comparison between the two tables happens. By leveraging the hash tables created in the building phase, the algorithm can quickly locate potential matches without having to scan through the entire dataset. This phase is designed to be as efficient as possible, minimizing the computational overhead and ensuring rapid execution.

4. Joining Phase

Whenever matching tuples are found in the hash tables during the probing phase, the hash join algorithm merges these tuples to form the output of the join. This result set includes only those tuples that have matching join attributes in both input tables, consistent with the behavior of an inner join. The joining phase is the culmination of the previous steps, where the actual output is generated. By focusing only on the matching tuples, the algorithm ensures that the final result is both accurate and relevant. This phase is critical for producing the desired outcome, ensuring that only the necessary data is included in the result set.

The hash join algorithm is highly efficient, especially when dealing with large data sets. By minimizing the number of comparisons required and localizing the search process through partitioning, the algorithm enhances performance. This efficiency is particularly noticeable when handling substantial amounts of data, where traditional join methods might struggle. The hash join algorithm’s ability to break down the data into smaller, manageable chunks ensures that it can handle large datasets with ease, making it a preferred choice for many database operations.

Even though the hash join algorithm is memory intensive, it is quite scalable as it handles data in partitions. This characteristic makes it suitable for distributed environments where memory resources can be managed across multiple nodes. Scalability is a crucial factor in modern database management, where data volumes are continually increasing. The hash join algorithm’s partitioning approach allows it to scale effectively, ensuring that it can handle growing datasets without compromising performance. This scalability is particularly beneficial in cloud-based environments, where resources can be allocated dynamically based on the workload.

Implementing the hash join algorithm within NoDA without relying on external libraries allows for greater control over the performance and optimization of the join operation. This self-contained approach ensures that the system remains robust and less susceptible to external dependencies. By avoiding reliance on third-party libraries, the NoDA API can maintain a high level of stability and performance. This independence also allows for custom optimizations and improvements, tailored specifically to the needs of the application. The ability to fine-tune the algorithm ensures that it can deliver the best possible performance for a given workload.

The implementation here is to make a method to join the two NoSqlDbResults of the two operations we want to perform the join. This specific application of the hash join algorithm highlights its versatility and effectiveness in handling join operations within NoDA. By focusing on the unique requirements of NoSQL databases, the algorithm can be optimized to deliver exceptional performance.

The hash join algorithm stands out as a highly efficient and scalable method for performing join operations, particularly in environments where external libraries are not an option. Its implementation within the NoDA API showcases its ability to handle large datasets effectively, ensuring fast and accurate join operations. By partitioning the data, building efficient hash tables, and probing these tables for matches, the algorithm delivers superior performance compared to traditional join methods. Moreover, its independence from external libraries provides a robust and self-contained solution, making it an invaluable tool for database management.

In summary, the hash join algorithm's design and implementation within NoDA offer a powerful and optimized approach to handling join operations. Its efficiency, scalability, and independence from external dependencies make it a preferred choice for many database applications. As data volumes continue to grow, the importance of efficient and scalable join methods like the hash join algorithm will only increase, ensuring that it remains a critical component of modern database systems.

The method signature is

```
Dataset<Row> makeJoin (Dataset<Row> leftDf, Dataset<Row> rightDf, JoinOperator jo, String condition);
```

It takes the two NoSqlDbOperators from their results we want to join and the JoinOperator that holds information from their columns. Finally returns the results as JSON format String.

The code below tranforms the NoSqlDbResults to a List of NoSqlDbRecord objects.

```
List<NoSqlDbRecord> getResultsWithColumn (NoSqlDbOperators noSqlDbOperators, String colName) {

	List<NoSqlDbRecord> list = new ArrayList<>();

	NoSqlDbResults results = noSqlDbOperators.getResults();

	while (results.hasNextRecord()) {

		NoSqlDbRecord record = results.getRecord();

		if (record.containsField(colName)) {

		list.add(record);}

	}

return list;}
```

This method will be used to make a List of the results records of two NoSqlDbOperators. After that we need to create a Map for hold the records by the common column as a key. The actual method that handles the join implementation is presenting below.

```
String makeJoin(NoSqlDbOperators left, NoSqlDbOperators right, JoinOperator jo) {

    // Fetch records containing the necessary columns

    List < NoSqlDbRecord > firstRecords = getResultsWithColumn(left, jo.getColumnAName());

    List < NoSqlDbRecord > secondRecords = getResultsWithColumn(right, jo.getColumnBName());

    Map < String, List < NoSqlDbRecord >> hashTable = new HashMap < > ();

    for (NoSqlDbRecord record: firstRecords) {

        String key = record.getString(jo.getColumnAName());

        hashTable.computeIfAbsent(key, k - > new ArrayList < > ()).add(record);

    }

    List < NoSqlDbRecord > results = new ArrayList < > ();
```

```
for (NoSqlDbRecord record: secondRecords) {
    String key = record.getString(jo.getColumnBName());
    if (hashTable.containsKey(key)) {
        for (NoSqlDbRecord matchingRecord: hashTable.get(key)) {
            results.add(matchingRecord);
            results.add(record);
        }
    }
}
return new Gson().toJson(results);}
```

The integration of a hash join approach within the NoDA API, utilizing Java Maps to store results from NoSqlDbResults and perform comparisons, showcases a practical method for combining data sets. Additionally, the use of Gson for JSON serialization adds a layer of functionality, allowing the transformed data to be easily used in various applications or returned to clients in a universally compatible format. Let's break down the process:

## 3.5 Use Case

The *NoSqlDbOperators* class acts like a builder as showed before this is maintained for letting the user interact with all the given methods of the API. Lets say a client's code looks like this below.

```
noSqlDbOperators.filter(newOperatorEq("countryName", "Denmark"))
```

At this line the client filters the data at the collection with the equality operator at the field country name with the value of Denmark.

At the other instance of the class lets say that a projection occurs as below.

```
noSqlDbOperatorSecond.project("countryName", "sourcemmsi", "shipLength")
```

At this point we must say that any other method from the API can be called even if the join method be used.

The join method remain the same aggregation stages of our interaction in order to take place the join to final results from the two instances.

Here is a code look as an example of the join.

```
noSqlDbOperators.join(noSqlDbOperatorsSecond, newOperatorEq("A","B"))
```

A and B are the String names of columns with the join will use to fetch the common data.

The join operation actual takes place when a call toDataframe().

This will return the joined two dataframes from two instances after their aggregations stages run.

## 4 Experiments

The dataset contains information from ships tracked using the Automatic Identification System, supplemented with spatial and temporal data. It includes navigation, vessel-specific, geographic, and environmental data collected over six months from October 1, 2015, to March 31, 2016. The areas covered are the Celtic Sea, the English Channel, and the Bay of Biscay. Designed for use with relational databases, it supports PostgreSQL and the PostGIS extension, facilitating the handling of its spatial aspects.

The database used for hosting the data is MongoDB.

The experiments for join made at a standalone environment, not a cluster.

The nari static collection contains data for ship characteristics. At the other collection the nari dynamic contains information of ship position tracking, so it contains except from information like ship identification number (sourcemmsi) or name, it has longitude and latitude.

The two collection have common columns so they can easily joined under these. First collection has a big size of documents, about nineteen million. The second one about a million.

The nari static structure of documents  is showed above at the picture

{

"sourcemmsi":"304091000",

"imonumber":"9509255",

"shipname":"HC JETTE-MARIT ",

"shiptype":"70",

"tobow":"130",

"tostern":"30",

"tostarboard":"18",

"toport":"6",

"destination":"BREST ",

"t":"1443650423" }

The nari dynamic document is like at the figure above.


```
{
  "_id":{
    "$oid":"61fc15d01eaf8634e34c9bdb"
  },
  "sourcemmsi":"245257000",
  "navigationalstatus":"0",
  "rateofturn":"0",
  "speedoverground":"0.1",
  "courseoverground":"13.1",
  "trueheading":"36",
  "lon":"-4.4657183",
  "lat":"48.38249",
  "t":"1443650402"
}
```


As there are two different types of join developed at this thesis one composed with apache spark and one concrete without any public framework composition the measurement is the time and as secondly the capabilities of each method to join.

Apache spark's implementation has multiple ways of join between data sets implemented. The hash join implementation joins the results on common field column.

| Method | Time(Sec) | Documents Size A Collection | Documents Size B Collection |
|---|---|---|---|
| Spark | 4 | 16 | 18 |
| HashJoin | 2 | 16 | 18 |
| | | | |
| Spark | 5 | 65 | 92 |
| HashJoin | 2 | 65 | 92 |
| | | | |
| Spark | 4 | 127 | 116 |
| HashJoin | 7 | 127 | 116 |
| | | | |
| Spark | 4 | 291 | 303 |
| HashJoin | 17 | 291 | 303 |

The selection of that two collections is made for large join case in order to penetrate the join code at both cases of Apache Spark and custom hash join implementation.

An outcome of these two addons of the NODA Api is that if we want to join across large data sets these methods can handle it. The Apache Spark approach of join two datasets is more efficient among a cluster environment. Also sparks solution provides the other operators for joins. At the other side the hash join implementation algorithm for join is a very common use. The cons here is that the join implementation is only for the common columns and not for operations like left or right join.

# 5 Conclusion

In this thesis, we have addressed the complexities of implementing join operations across various NoSQL databases by enhancing the NoDA API and integrating it with Apache Spark's Dataset API. Our approach leverages the powerful data processing capabilities of Spark, enabling efficient and flexible join operations, including inner, outer, left, and right joins. These types of joins are fundamental for combining datasets based on key attributes, facilitating the extraction of meaningful insights from large, distributed data stores. By utilizing Spark, we capitalize on its in-memory processing capabilities, which significantly reduce the latency typically associated with disk-based operations, thus enhancing overall performance.

The implementation of a hash join algorithm within the NoDA framework provides an alternative solution for scenarios where external libraries are not viable, ensuring robust performance and scalability. Hash join is particularly effective in distributed computing environments as it efficiently handles large datasets by partitioning them into smaller, manageable chunks. This partitioning allows for parallel processing, which is a cornerstone of scalable big data solutions. The choice of a hash join algorithm also circumvents the limitations posed by traditional join methods in NoSQL databases, which often struggle with complex join operations due to their schema-less nature and distributed architecture.

Through comprehensive experimentation, we demonstrated the efficacy of our solution in handling large datasets, highlighting the advantages and potential limitations of both Spark-based and custom join implementations. We measured performance metrics such as execution time, resource utilization, and scalability across different join types and dataset sizes. The results showcased that our enhanced NoDA API not only improves join operation performance but also offers flexibility in handling diverse data structures inherent to NoSQL systems.

Moving forward, further enhancements could focus on decomposing the reliance on Apache Spark to streamline join operations and extending the hash join algorithm to support more complex join types, such as semi-joins and anti-joins.

Additionally, integrating machine learning algorithms to optimize join operations dynamically based on dataset characteristics and query patterns could further enhance performance.

This work significantly contributes to the field of big data analytics by offering a versatile and efficient tool for integrating data from diverse NoSQL systems, thereby facilitating more comprehensive data analysis and decision-making in distributed computing environments. By bridging the gap between NoSQL databases and traditional relational operations, we pave the way for more seamless and powerful data integration solutions. Our contributions not only advance the technical capabilities of the NoDA framework but also provide a valuable resource for practitioners and researchers aiming to harness the full potential of big data analytics in real-world applications.

# 6 Future Upgrades

Future upgrades to the NoDA API could significantly enhance its performance and versatility, positioning it as a more robust solution for big data processing in NoSQL environments. One major upgrade could involve developing a lightweight join framework within the NoDA API that bypasses the need for Apache Spark. By eliminating this dependency on external libraries, the NoDA API could achieve improved performance, especially in environments with limited resources. This would be particularly beneficial in scenarios where the overhead of managing and operating Spark is not justified, allowing the NoDA API to operate more efficiently and with lower resource consumption.

Additionally, expanding the existing hash join algorithm to support a wider range of join types, such as left, right, and full outer joins, would further increase the API's versatility. Currently, many join operations are limited in scope, but by enabling more comprehensive join capabilities, the NoDA API could handle a broader array of data processing tasks. This would be especially useful for complex data integration scenarios where different types of joins are required to merge datasets effectively.

Another promising direction for enhancing the NoDA API is the incorporation of machine learning techniques to optimize join strategies based on dataset characteristics and query patterns. Machine learning could analyze historical data and query performance to identify the most efficient join strategies, adapting dynamically to changing data conditions. This approach could lead to significant performance improvements by reducing the time and computational resources required for join operations. For example, by learning from past queries, the system could predict the best join method for a given set of data, optimizing execution plans and improving overall efficiency.

Integrating real-time data processing capabilities is another critical upgrade that could greatly broaden the applicability of the NoDA API. By enabling the API to handle streaming data, it could support real-time analytics and decision-making scenarios. This capability is increasingly important in today's fast-paced data environments, where timely insights can provide a competitive edge. Real-time processing would allow users to analyze data as it arrives, making it possible to detect trends,

anomalies, and other important metrics instantly. This would be invaluable in applications such as financial services, healthcare, and IoT, where real-time data analysis is crucial for operational effectiveness.

Furthermore, enhancing the NoDA API to support real-time data processing would involve developing efficient mechanisms for ingesting and processing streaming data. This could include integrating with popular streaming platforms such as Apache Kafka or Apache Flink, which are designed to handle high-throughput, low-latency data streams. By combining these technologies with the NoDA API, users could build robust data pipelines that support both batch and real-time processing, providing a unified platform for comprehensive data analysis.

These upgrades would not only enhance the functionality and efficiency of the NoDA API but also solidify its position as a comprehensive solution for big data processing in NoSQL environments. By reducing dependency on external libraries, supporting a wider range of join types, leveraging machine learning for optimization, and enabling real-time data processing, the NoDA API could become an indispensable tool for data engineers and analysts. These improvements would address current limitations and open up new possibilities for innovative data processing solutions, ensuring that the NoDA API remains at the forefront of big data technology.

Ultimately, these enhancements would make the NoDA API a more powerful and flexible tool for managing and analyzing large datasets. They would provide users with the ability to perform complex data operations more efficiently, adapt to changing data environments, and derive valuable insights in real-time. As data continues to grow in volume and complexity, these upgrades will be essential for maintaining the relevance and effectiveness of the NoDA API in meeting the evolving needs of the data processing landscape.

# 7 References


- [1] Nikolaos Koutroumanis, Christos Doulkeridis, Akrivi Vlachou:
  Tearing Down the Tower of Babel: Unified and Efficient Spatio-temporal Queries for NoSQL Stores.
- [2] Abraham Silberschatz, Henry Korth, S. Sudarshan
  Database System Concepts
- [3] Jean-Georges Perrin
  Spark in Action
- [4] Chuck Lam
  Hadoop in Action
- [5] Fabian Hueske, Vasiliki Kalavri
  Stream Processing with Apache Flink
- [6] Neha Narkhede, Gwen Shapira, Todd Palino
  O'Reilly | Kafka: The Definitive Guide
- [7] Jonathan Leibiusky, Gabriel Eisbruch, Dario Simonassi
  Getting Started with Storm