University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Greece

A Big Data Analytics System Based on a High Level Query Language Using Apache Spark

Vassilis Glampedakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

Thesis Advisor: Prof. *Dimitris Plexousakis*

September 2017

University of Crete
Computer Science Department

A Big Data Analytics System based on a High Level Query Language using Apache Spark

Thesis submitted by
Vassilis Glampedakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Vassilis Glampedakis

Committee approvals: _____
Dimitris Plexousakis
Professor, Thesis Supervisor

_____
Nicolas Spyratos
Professor Emeritus, Thesis Co-Supervisor

_____
Yannis Tzitzikas
Associate Professor, Committee Member

Departmental approval: _____
Antonis A. Argyros
Professor, Director of Graduate Studies

Heraklion, September 2017

Abstract

Big data analytics is one of the most active research areas today with a lot of challenges, both theoretical and practical. This thesis makes a contribution to the area of big data analytics by implementing the HIFUN language using the Apache Spark framework. HIFUN is a high level query language, proposed for expressing analytic queries over big data sets. This language makes a clear separation between the conceptual and the physical level. An analytic query and its answer are defined at the conceptual level independently of the nature and location of data. The abstract definitions are then mapped to lower level evaluation mechanisms, taking into account the nature and location of data, as well as other related aspects. In this thesis, we leverage this language to design and implement a system which allows a user to analyse, visualize and discover information useful for decision making, which is "hidden" in large-scale data sets. In the physical level, HIFUN queries are mapped to lower level evaluation mechanisms of the Apache Spark framework following the conceptual evaluation scheme proposed by HIFUN and supporting a big range of data set formats. In the conceptual level, we apply the query rewriting rules and create query execution plans, proposed by HIFUN. Our work shows that the HIFUN formal model is useful in practice and the experimental evaluation of the system proves that the model's approach to query rewriting and to the generation of query execution plans succeeds in reducing the computational costs *regardless* of the nature of the data.

Περίληψη

Η ανάλυση μεγάλων δεδομένων είναι μία από τις πιο ενεργές ερευνητικές περιοχές σήμερα και συνοδεύεται από πολλές θεωρητικές και πρακτικές προκλήσεις. Αυτή η εργασία συμβάλλει σε αυτήν την περιοχή υλοποιώντας την γλώσσα HIFUN χρησιμοποιώντας το framework Apache Spark. Η HIFUN είναι μία γλώσσα επερωτήσεων υψηλού επιπέδου η οποία έχει προταθεί για την έκφραση επερωτήσεων ανάλυσης πάνω σε δεδομένα μεγάλου όγκου. Αυτή η γλώσσα κάνει έναν σαφή διαχωρισμό μεταξύ του εννοιολογικού και του φυσικού επιπέδου. Μία επερώτηση ανάλυσης και η απάντησή της ορίζονται στο εννοιολογικό επίπεδο ανεξάρτητα από την φύση και την τοποθεσία των δεδομένων. Κατόπιν, αυτοί οι αφηρημένοι ορισμοί αντιστοιχίζονται σε μηχανισμούς αποτίμησης χαμηλότερου επιπέδου, λαμβάνοντας υπόψιν την φύση και την τοποθεσία των δεδομένων, όπως επίσης και άλλους σχετιζόμενους παράγοντες. Σε αυτήν την εργασία, αξιοποιούμε αυτήν την γλώσσα για την σχεδίαση και την υλοποίηση ενός συστήματος το οποίο επιτρέπει σε έναν χρήστη να αναλύσει, να οπτικοποιήσει και να ανακαλύψει πληροφορία η οποία μπορεί να είναι χρήσιμη στη λήψη αποφάσεων και η οποία είναι κρυμμένη σε δεδομένα μεγάλου όγκου. Στο φυσικό επίπεδο, οι επερωτήσεις της HIFUN αντιστοιχίζονται σε μηχανισμούς αποτίμησης χαμηλότερου επιπέδου του Apache Spark, ακολουθώντας το προτεινόμενο από την HIFUN εννοιολογικό πλάνο αποτίμησης, υποστηρίζοντας μία μεγάλη γκάμα μορφών δεδομένων. Στο εννοιολογικό επίπεδο, εφαρμόζουμε τους κανόνες επανεγγραφής επερωτήσεων και δημιουργούμε πλάνα εκτέλεσης επερωτήσεων, προτεινόμενα από την HIFUN. Η εργασία αυτή δείχνει ότι το τυπικό μοντέλο της HIFUN είναι χρήσιμο στην πράξη και η πειραματική αξιολόγησή του συστήματος αποδεικνύει ότι η προσέγγιση του μοντέλου στην επανεγγραφή την επερωτήσεων και στην παραγωγή πλάνων εκτέλεσης επερωτήσεων επιτυγχάνει τη μείωση του υπολογιστικού κόστους *ανεξάρτητα* από την φύση των δεδομένων.

Acknowledgements

I would like to thank all the people that influenced me during this wonderful journey.

Firstly, my supervisors Prof. Dimitris Plexousakis and Prof. Nicolas Spyratos, for their continuous support and guidance in my work and studies.

Secondly, the member of my committee, Yannis Tzitzikas for his valuable comments and questions during my defense.

Thirdly, I would like to thank the Institute of Computer Science (ICS), Foundation of Research and Technology – Hellas (FORTH) and more specifically the Information Systems Laboratory (ISL) for the support during both my undergraduate and graduate studies.

My best thanks to my friends and colleagues Giorgos Nikitakis, Dimitris Arampatzis, Michalis Loukakis, Nikos Liapakis, David Parastatidis, Prodromos Lilitsis, Nikoleta Tsabanaki, Nina Saveta, Georgia Troullinou, Roula Avgoustaki, Katerina Papantoniou and Dimitra Zografistou for making FORTH a fun and interesting place, and supporting me during my studies.

Moreover, I would like to thank Panagiotis Papadakos and Vangelis Kritsotakis for their tutorials and advices and for providing me access to the hardware needed for the experiments of this thesis.

Last but not least I am more than grateful to my parents Chrysi and George and my brother Nikos for their endless love, support and encouragement through my studies and my life in general. I wouldn't have made it this far without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Big data analytics is one of the most active research areas with a lot of challenges and needs for innovations which affect a wide range of industries. In response to this trend, the research community and the IT industry have developed formal models along with a number of frameworks to facilitate large-scale data analytics. This thesis makes a contribution to this area by implementing HIFUN, a high level functional query language for big data analytics [1][2]. Our implementation is based on Apache Spark, a widely used framework framework for big data analytics. In this section, we describe the context, the motivation and the contributions of our work.

1.1 Big Data

In the information era, enormous amounts of data have become available on hand to decision makers [3]. Big data are data characterized by the 6 Vs [4][5][6][7]:

1. **Volume**: there is no specific size at which a data set is considered big and this definition is subjective. In any case , the term big data refers to data sets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze.

2. **Velocity**: it is the measure of how fast the data are generated and processed and this dimension refers to the capability of understanding and responding to events as they occur. A number of applications today rely on real-time data generation and processing to meet the demands of everyday life e.g. Google Maps, Facebook etc.

3. **Variety**: it deals with the complexity of big data along with information and semantic models behind these data. Big data can be collected as structured, unstructured, semi-structured or mixed and this dimension imposes new requirements to data storage and database design.

4. **Veracity**: this dimension refers to the quality, the uncertainty and the imprecision which this kind of data may have. They may have missing values, misstatements or untruths which create a number of challenges during their processing.

5. **Voracity**: this dimension refers to the strong appetite of the consumers for data. There is an increase in the information requirements of users as a result of the new capabilities coming from these large-scale datasets.

6. **Value**: this is the most important dimension of big data. Nowadays, businesses invest a lot of time and money collecting and saving data and their motive is to turn these data into potential value.

It is undeniable that the collection and the processing of big data can enable insights that unlock new sources of value and can support an array of human activities [8]. This will be supported by the use of new technical architectures, analytics, and frameworks [3].

1.2 Big Data Analytics

Big data analytics is the process of collecting, transforming and analyzing big data sets with the goal of discovering "hidden" patterns and other useful information to suggest conclusions and support decision making. Analysts working with big data sets basically want the knowledge that comes from analyzing the data [1]. This process can be useful in multiple domains including businesses, health, security, research, advertising etc. For example, in the business field, it can help organizations to better understand the information contained within the data and identify the data that is most important to the business and future business decisions [9]. Both can result in revenue increase, cost reduction, increased productivity and better performance. In the security field, it can help people prevent crime and predict possible future crime locations by analyzing criminal records and revealing crime patterns.

This process offers a lot of opportunities but also comes with a number of challenges. For example, as we mentioned in the previous subsection, big data are characterized by variety. As a result, one of the challenges which appear is how to integrate data of different structure level from different sources. Moreover, this process demands specific skills and for this reason, people trained in big data analytics tools are needed. Furthermore, there is a need to reduce the computational cost of this procedure to the minimum.

In any case, the analysis of big data requires new specialized frameworks and programming models that process data sets in a parallel manner in a distributive computing environment consisting of a considerable number of machines. This is because traditional database software tools do not have a good performance in large-scale datasets processing [1]. Some of those basic frameworks and models are described in the next section.

1.3 Motivation

As stated in [1], there is currently a number of systems developed for large-scale data sets processing. While these systems co-exist, each carefully optimized in accordance with the final application goals and constraints, their evolution has resulted in an array of solutions catering to a wide range of diverse application environments. Unfortunately, this has also fragmented the big data solutions that are now adapted to particular types of applications. At the same time, applications have moved towards leveraging multiple paradigms in conjunction, for instance combining real time data and historical data. This has led to a pressing need for solutions that seamlessly and transparently allow practitioners to mix different approaches that can function and provide answers as an all-in-one solution. Based on this observation, a high level query language, called HIFUN, was proposed in [1][2]. HIFUN allows an analyst to formulate queries and study their properties at the conceptual level along with mappings to existing evaluation mechanisms which perform the actual evaluation of queries. The objective of HIFUN was to clearly separate the conceptual and the physical level so that one can express analysis tasks as queries at the conceptual level independently of how their evaluation is done at the physical level. This solves the problem described above, as HIFUN is agnostic of the application environment as well as of the nature and location of data.

However, HIFUN is a formal language that was not implemented in practice. Moreover, although some theory around *Query Rewriting* and *Query Execution Plans* was proposed for minimizing the evaluation cost of queries, this work lacks an algorithm which generates query execution plans and an experimental proof that the rewritings proposed are efficient.

1.4 Thesis Contribution

In this thesis, we show that the HIFUN language is indeed of great benefit in practice. We do this by implementing this language using a widely used framework, Apache Spark. More precisely:

1. We map the conceptually defined queries of this language to physical level mechanisms of the *Apache Spark* framework according to a proposed conceptual query evaluation scheme. The mapping is done using two APIs of the framework which can support a big variety of data set formats.

2. Based on the HIFUN definitions of *Query Rewriting* rules and *Query Execution Plans*, we design and implement an algorithm which rewrites queries and generates query execution plans to achieve the minimization of the cost of the evaluation of a set of queries.

3. We map the rewritten queries in the Apache Spark framework and prove experimentally that the HIFUN approach provides noticeable benefits related

to query evaluation cost reduction.

4. We leverage the simple format of HIFUN queries to create a user-friendly interface which allows any user to express analytic queries and discover useful information in a data set, while hiding the evaluation mechanisms from him.

1.5 Thesis Organization

The remaining of this report is organized as follows: In section 2, we provide the background of this thesis. In subsection 2.1 we present the Google's MapReduce programming model and in the next 2 subsections, 2.2 and 2.3, we briefly describe and compare two of the currently most popular big data processing frameworks which use this model to perform computations. In section 3, we present the HIFUN language [1][2]. More precisely, in subsection 3.1, the definitions of this language are reviewed; in 3.1.1, the definition of an analytic query and its answer is given; and in subsection 3.1.2, the definition of the *Analysis Context* is given and a query language over it is defined. Moreover, in subsection 3.2, the features of the language are presented; in subsection 3.2.1, a proposed theory related to *Query Rewriting* and *Query Execution Plans* is reviewed; in subsection 3.2.2, the conceptual scheme of the language for query evaluation based on the abstract definition of a query is briefly reviewed; and in subsection 3.2.3, we explain how the different representations of a query answer can be produced. In section 4, we give a detailed description of a system that we implemented, which includes all the contributions described above, and we explain what this system provides. In subsections 4.1 and 4.2, the mapping of the HIFUN conceptual scheme to physical level mechanisms using two APIs of the Apache Spark is given. In the last subsections of the main section, 4.3 and 4.4, we describe how we designed an algorithm which reduces the queries evaluation cost based on the theory of *Query Rewriting* and *Query Execution Plans*, and how a user-friendly interface is implemented which allows any user to perform an analysis on a data set and explore useful information in its answer. Finally, in the last two sections, we evaluate the effectiveness of the query rewritings at reducing queries evaluation cost, present directions of future work and make some concluding remarks.

Chapter 2

Background

In this section, we describe some of the popular frameworks and technologies developed for big data processing.

2.1 The MapReduce Programming Model

MapReduce, seen from a high level, is a programming model for processing data sets of large volume in a parallel manner. A programmer of the MapReduce library expresses a computation as two functions, the Map and the Reduce function. Firstly, the Map function takes an input pair and produces a set of intermediate key/value pairs (K, V). All intermediate values associated with the same intermediate key are passed to the Reduce function. The latter accepts an intermediate key K_i and a set of values for that key and merges these values to form a possibly smaller set of values (usually zero or one output value is produced per Reduce invocation). These are depicted in Figure 2.1. A big amount of real world tasks can be expressed using this model and programs following it can be automatically parallelized and executed on a large cluster of machines achieving high scalability. Each key-value pair can be mapped or reduced independently and this means that many different processors, or even machines, can each take a section of the data and process it separately [10][11][12][13][14][15].

This model needs a framework that runs these programs in parallel, automatically handling the details of distribution, synchronization, and fault-tolerance. The model and the framework work together to make programs that are scalable, distributed, and fault-tolerant. Some of the framework's jobs include 1. constructing the input pairs of the mapper from the input files, 2. collecting, sorting and grouping the intermediate key-value pairs by key and 3. collecting the output pairs of the reducer and storing them in output files. In general, the framework handles all the needed synchronization between the steps of Map and Reduce [12].

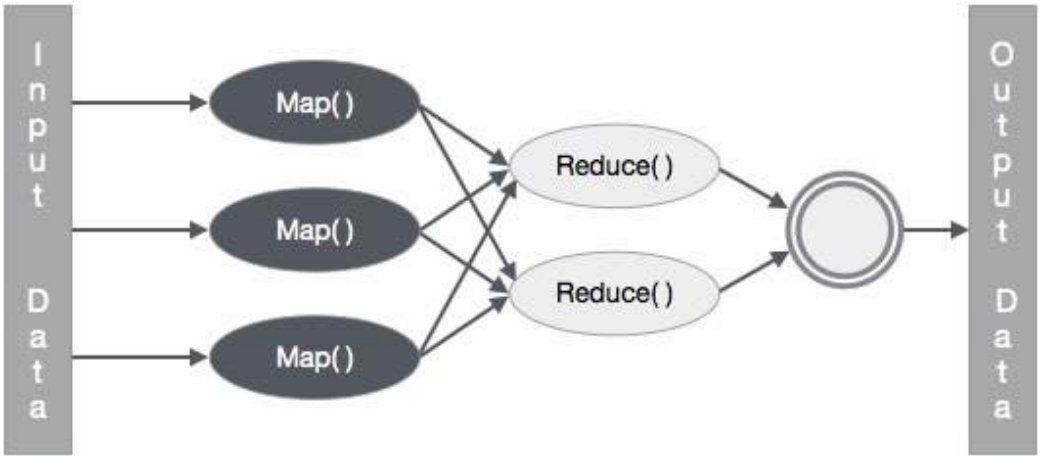


Figure 2.1: The MapReduce Programming Model

2.2 Apache Hadoop

Hadoop is a programming framework developed to support the processing of data sets of large volume in a distributed computing environment. It consists of computer clusters built from commodity hardware. All the modules in Hadoop are designed with an assumption that hardware failures are common occurrences and should be automatically handled by the framework. The fundamental parts of the *Hadoop* ecosystem are the *Hadoop Distributed File System (HDFS)*, which is its storage part, and the *Hadoop MapReduce*, which is an implementation of the Google's *MapReduce* programming model as described above. Some additional parts of the ecosystem are 1. the basic modules: *Hadoop Common* which contains libraries and utilities needed by other Hadoop modules, the *Hadoop Yarn* which is a platform responsible for managing computing resources in clusters, and 2. a number of various components that can be installed on top of or alongside Hadoop like Apache Hive, Base and Zookeeper.

The architecture of a multi-node Hadoop cluster can be seen in Figure 2.2. *Hadoop* creates clusters of machines, distributes data across the nodes in the cluster and transfers packaged code into nodes to process the data in parallel. This allows the data set to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking. A *Hadoop* cluster is consisted of two layers, the *HDFS* layer and the *MapReduce* layer. It includes a single master and multiple worker nodes. The master node consists of a Job Tracker, a Task Tracker, a NameNode, and a DataNode. A slave or worker node acts as both a DataNode and a TaskTracker. Briefly, a Job Tracker pushes work to available Task Tracker nodes in the cluster to be executed, trying to keep the work as close to the data as possible in order to avoid network traffic [16][17][18].

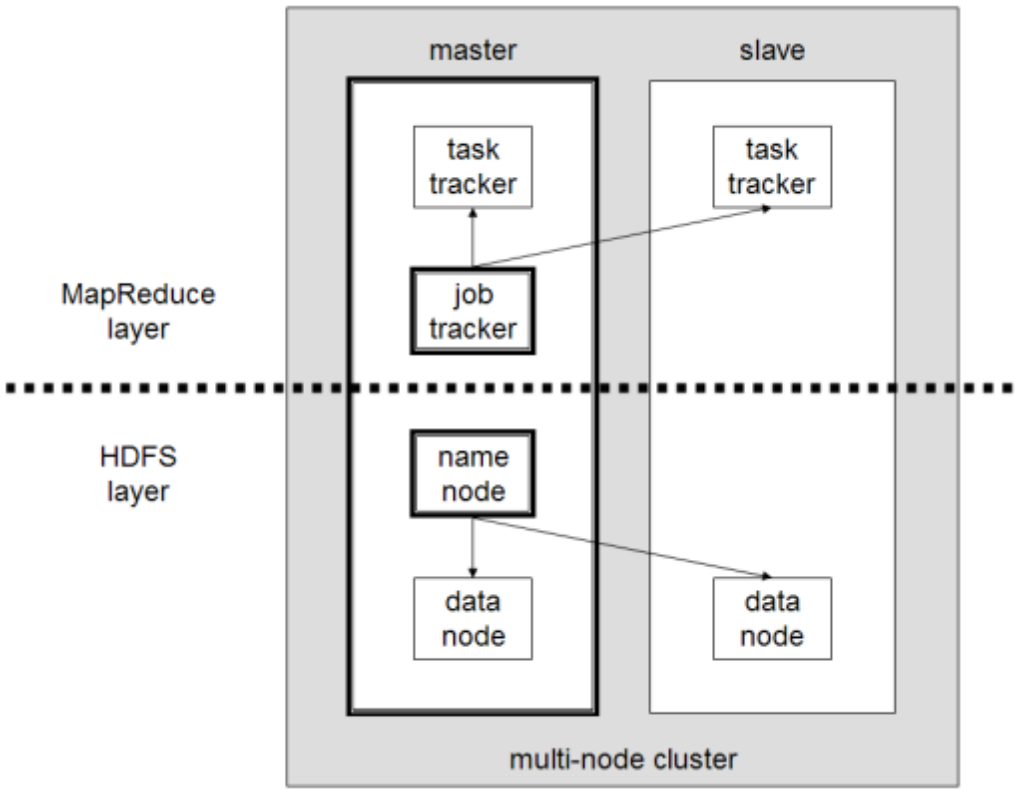


Figure 2.2: A multi-node Hadoop cluster

2.2.1 Hadoop Distributed File System (HDFS)

The *Hadoop Distributed File System (HDFS)* is a fault-tolerant, distributed and scalable storage system, part of the *Apache Hadoop* ecosystem. It can store a massive amount of data and scale up incrementally by adding more DataNodes to the cluster. Hadoop splits files into blocks and distributes them across the nodes of the cluster. A strong benefit of this filesystem is that if a node of a cluster fails, Hadoop can continue to operate the cluster using the remaining nodes without losing data or interrupting work. Other benefits of it are high portability across heterogeneous hardware and software platforms, a simple coherency model and network congestion minimization.

It has a master/slave architecture which is depicted in Figure 2.3. An *HDFS* cluster consists of a master server that manages the file system namespace and controls access to files by the clients, that is the NameNode, and a number of DataNodes which manage storage attached to the nodes that they run on. *HDFS* splits data files into blocks and stores these blocks in a set of DataNodes. The NameNode performs operations on files and directories, and determines the mapping of blocks to DataNodes. The DataNodes follow commands from the NameNode

performing block-related operations and are responsible for the read/write requests from the clients. The fault-tolerance is achieved by replicating the blocks of a file across the DataNodes [19].

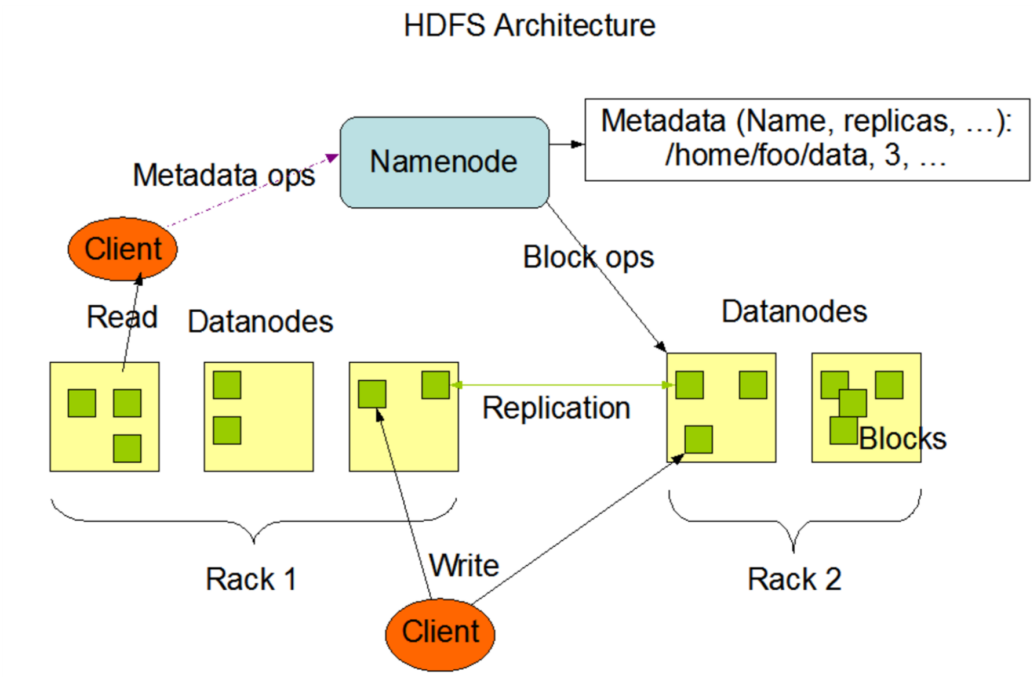


Figure 2.3: The Hadoop Distributed File System Architecture

2.3 Apache Spark

Apache Spark is a fast and general engine for large-scale data processing. Some of its goals include: 1. Generality: designed to cover a wide range of workloads (e.g., job batches, iterative algorithms, interactive queries and streaming) that previously required separate distributed systems, 2. Low Latency: designed for speed, operating both in memory and on disk, 3. Fault Tolerance, 4. Simplicity: its capabilities are accessible via a set of rich APIs, 5. Existing technologies exploitation: exploits existing distributed file systems and cluster managers e.g. *HDFS, Yarn*. It is a unified ecosystem consisted of a number of main components including Spark core and some upper-level libraries: *MLib* for machine learning, *GraphX* for graph analysis, *Spark Streaming* for stream processing and *Spark SQL* for structured data processing [20][21].

Apache Spark has now emerged as the de facto standard for big data analytics after Hadoop's MapReduce. As a framework, it combines a core engine for distributed computing with an advanced programming model for in-memory processing. Although it has the same linear scalability and fault tolerance capabilities

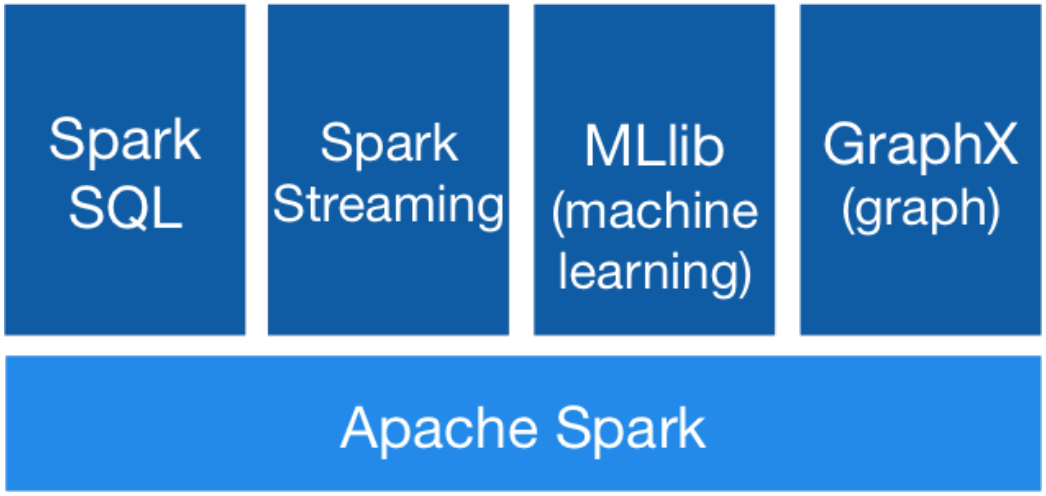


Figure 2.4: Apache Spark Stack

as those of MapReduce, it comes with a multistage in-memory programming model comparing to the rigid map-then-reduce disk-based model. With such an advanced model, Apache Spark is much faster and easier to use. It comes with rich APIs for performing complex distributed operations on distributed data. In addition, Apache Spark leverages the memory of a computing cluster to reduce the dependency on the underlying distributed file system, leading to dramatic performance gains in comparison with Hadoop's MapReduce. It is built upon the Resilient Distributed Datasets (RDDs) abstraction which provides an efficient data sharing between computations. Previous data flow frameworks lack such data sharing ability although it is an essential requirement for different workloads [21].

2.3.1 Spark Applications

A Spark application is consisted of the following entities: a driver program, a cluster manager, workers, executors and tasks. A *driver program* is an application that uses Spark as a library and defines a high-level control flow of the target computation. While a *worker* provides CPU, memory and storage resources to a Spark application, an *executor* is a Java Virtual Machine process that Spark creates on each worker for that application. A job is a set of computations that Spark performs on a cluster to get results to the driver program and a Spark application can launch multiple jobs. Spark splits a job into a directed acyclic graph of stages where each stage is a collection of tasks. A *task* is the smallest unit of work that Spark sends to an executor [21].

Figure 2.6 gives an overview of how Spark runs on clusters. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the driver program. More specifically, the SparkContext can connect to several types of cluster managers, which allocate resources across applications.

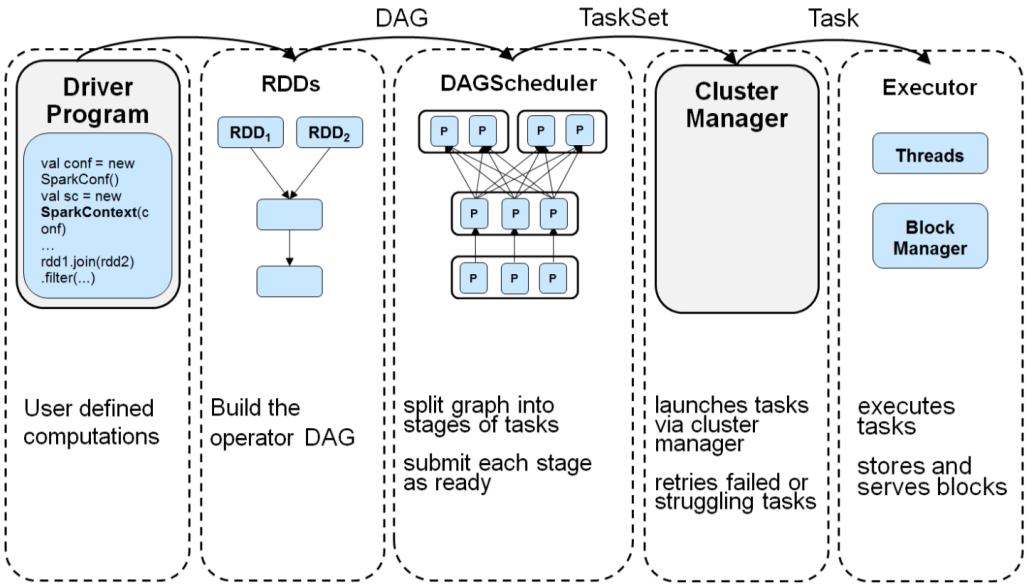


Figure 2.5: Apache Spark Execution Workflow

Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for an application. Next, it sends the application code to the executors. Finally, SparkContext sends tasks to the executors to run. This architecture allows Spark applications to run with high efficiency on the cluster [22].

As we describe in detail in the next sections, we designed and implemented a big data analytics system using the *Apache Spark* framework. Our system evaluates analytic queries expressed by the users by mapping them to lower level mechanisms of this framework. The data sets that need to be analyzed in our system are stored onto the *HDFS* and a number of our queries evaluation processes are designed based on the *MapReduce* programming model.

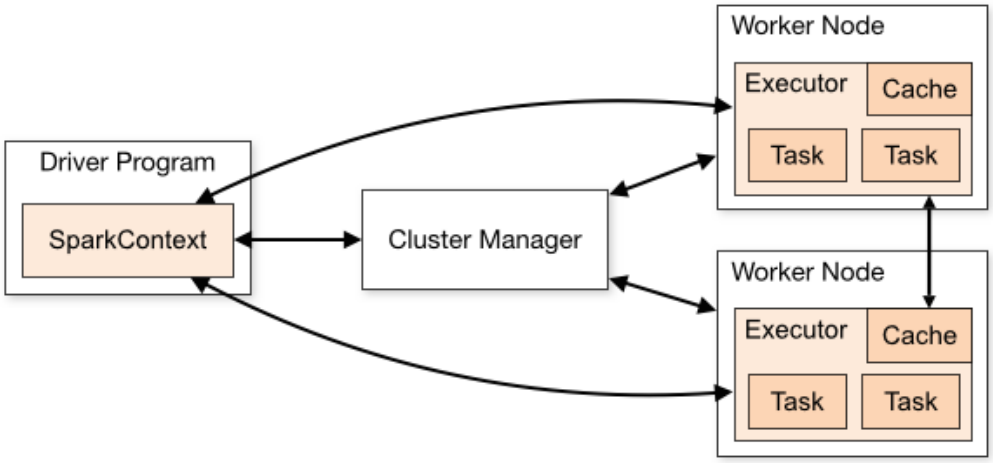


Figure 2.6: Key entities for running a Spark application

Chapter 3

The HIFUN Language

In this section, we are going to briefly describe the formal model of the high level query language called *HIFUN*, proposed for expressing analytic queries over big data sets. For a detailed description, the reader is referred to [1][2].

Firstly, the main benefit of *HIFUN* is that it separates the conceptual and the physical level, which means that it can be used to express analytic queries regardless of the nature of the data sets. This language can be used to express analytic queries on a data set D if:

1. D consists of data items that can be uniquely identified.
2. the items in D share a common set of attributes and each attribute can be seen as a function associating each data item of D with a value, in some set of values.

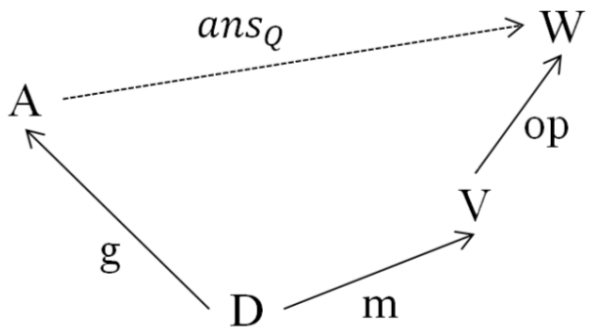
3.1 Definitions

3.1.1 The definition of an analytic query and its answer

An analytic query in HIFUN is defined to be a triple *(g, m, op)*, such that g and m are attributes of the data set D, and op is an aggregate operation applicable on m-values. The attribute g is called the "grouping attribute" and the attribute m is called the "measuring attribute". The query Q is evaluated in three steps:

1. **Grouping:** The items of D are grouped using the values of the grouping attribute g.
2. **Measuring:** The value of the measuring attribute m is extracted from each item contained in each group created in the previous step.
3. **Reduction** The values of the measuring attribute m are aggregated in each group to get a final value v_i. This aggregate value is defined to be the answer of the query$Q$ on g_i, that is $ans_Q(g_i) = v_i$

To formally define the query and its answer, some basic mathematical concepts, the functions and the partitions, are needed.



$$Q = (g, m, op)$$

$$ans_Q(a_i) = (\text{red}(\text{m}/g^{-1}(a_i), \text{op}))$$

Figure 3.1: The definition of an analytic query and its answer

The formal definition is depicted in Figure 3.1. Firstly, D is a finite set of data items, that is $D = \{d_1, ..., d_n\}$. An analytic query over D is a triple $Q = (g, m, op)$, where g is a function with domain the set D and range a set A, m is a function with domain the set D and range a set V and op is an operation over V taking its values in a set W. If $\{a_1, ..., a_n\}$ is the set of values of g, then we call grouping of D by g, the partition $\pi_g = \{g^{-1}(a_1), ..., g^{-1}(a_k)\}$ induced by g on D. The reduction of m with respect to op, denoted $red(m, op)$, is a value of W defined as $red(m, op) = op(\langle m(d_1), ..., m(d_n)\rangle)$. Based on the above, the answer to Q, denoted ans_Q, is a function from the set of values of g to W defined by:

$$ans_Q(a_i) = red(m/g^{-1}(a_i), op), i = 1, 2, ..., k$$

The following restricted queries can also be defined over D:

1. **Attribute-Restricted Query**: $(g/E, m, op)$, where E is any subset of D. To evaluate this query, we compute the restriction g/E and then evaluate the query over E.

2. **Result-Restricted Query**: $(g, m, op)/F$, where F is any subset of the target of g. To evaluate this query, we evaluate the query $Q = (g, m, op)$ over D and then we compute the restriction ans_Q/F.

3.1.2 Analysis Context

An analysis context over a data set D is any subset of attributes of the set of all attributes of D. It is the interface between an analyst and the data set. An example can be seen in Figure 3.2. The attributes with domain the data items of D are called *"direct"* and the attributes that can be derived from the *direct* attributes are called *"derived"*. It is a directed labelled acyclic graph whose nodes represent data sets, and whose edges are functions between the data sets. An analyst can express queries by choosing any node of the context and can also form complex grouping functions using the *functional algebra*. More specifically:

1. The *composition* operation can be used to compose one or more attributes to allow grouping by *"derived"* attributes. *e.g* $(r \circ b, q, sum)$

2. The *pairing* operation can be used to allow grouping by more than one attributes. *e.g* $(b \wedge p, q, sum)$

3. The *restriction* operation can be used to express restricted queries.

4. The *Cartesian product projection* operation can be used to express projection queries which are useful for *query rewritings* explained in the next subsection.

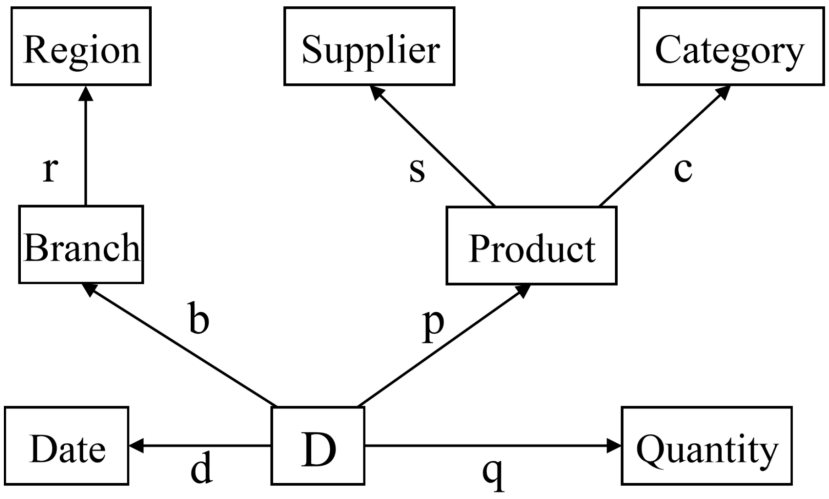


Figure 3.2: An Analysis Context example

Given a context C, a functional expression over it is either an edge of C or a well formed expression whose operands are edges and whose operations are those of the functional algebra. Based on this, a query over C is a triple *(e, e', op)*, such that e and e' have a common source and *op* is an operation over the target of e'. Finally, the query language of C is the set of all queries over C.

3.2 The Main Features of HIFUN

3.2.1 Query Rewriting and Query Execution Plans

In this subsection, we briefly describe how a given query can be rewritten in terms of one or more other queries to reduce its evaluation cost. *Query Rewriting* has two applications:

1. Optimizing the evaluation of a query which is done by taking advantage of results of queries that have already been evaluated.

2. Optimizing the evaluation of a set Q of queries.

   The rewriting rules are the following:

   (a) $Q = \{(g, m, op_1), ...(g, m, op_n)\} \Rightarrow Q = ((g, m), \{op_1, ..., op_n\})$

   (b) $Q = \{(g, m_1, op_1), ...(g, m_n, op_n)\} \Rightarrow Q = (g, (\{m_1, op_1\}, ..., \{m_n, op_n\}))$

   (c) $Q = \{(g_1, m, op), ..., (g_n, m, op)\}$, where op is a distributive operation. Queries of this form can be rewritten according to a number of rules:

      i. A set $Q = \{(f, m, op), ..., (h, m, op)\}$ can be rewritten as $Q = \{(f \wedge ... \wedge h, m, op), (proj_F, (f \wedge ... \wedge h, m, op), op), ..., (proj_H, (f \wedge ... \wedge h, m, op), op)\}$.

      ii. A set $Q = \{(g \circ f, m, op), ..., (h \circ f, m, op)\}$ can be rewritten as $Q = \{((g \wedge ... \wedge h) \circ f), m, op), (proj_G, ((g \wedge ... \wedge h) \circ f), m, op), op), ..., (proj_H, ((g \wedge ... \wedge h) \circ f), m, op), op)\}$.

      iii. A set $Q = \{(f, m, op), (g \circ f, m, op)\}$ can be rewritten as $Q = \{(f, m, op), (g, (f, m, op), op)\}$. This is called the *Basic Rewriting Rule*.

In case there are equality constraints over a context, they can be used to increase the possibilities of query rewritings in both applications.

Since some queries can be evaluated using the answers of other queries with lower cost, a *Query Execution Plan* (ordered set of steps of query evaluation) has to be created for every set of queries Q which guarantees that each query is evaluated only once and the evaluation order implied by rewriting is maintained. This is done by creating a *Query Rewriting Graph* for every set of queries Q, which is a directed labeled graph where the nodes are the queries of Q and there is an edge from a node Q to node Q' if Q' can be rewritten in terms of Q. The label of the edge $Q \rightarrow Q'$ is the function on which is based the rewriting of Q' in terms of Q. A *Query Execution Graph EG* for a set of queries Q is a subgraph of the *Query Rewriting Graph*, where the nodes of the subgraph are the queries of Q and each node has at most one predecessor. Finally, the *Query Execution Plan* is a *Query Execution Graph* together with an external order $\leqslant_e$ compatible with the rewriting order $\leqslant_w$. More specifically, the external order $\leqslant_e$ is the ordering of the queries to be evaluated and the rewriting partial order $\leqslant_w$ is defined by the rewriting rules

as described above. The rewriting order carries semantic information in addition to ordering information and that is the main difference between the two partial orders. An example for the above-mentioned can be seen in Figure 3.3. In the query execution plans depicted, the rewriting order $\leqslant_w$ is indicated by the edges in solid line, whereas the external order $\leqslant_e$ is indicated by the edges in dotted line.

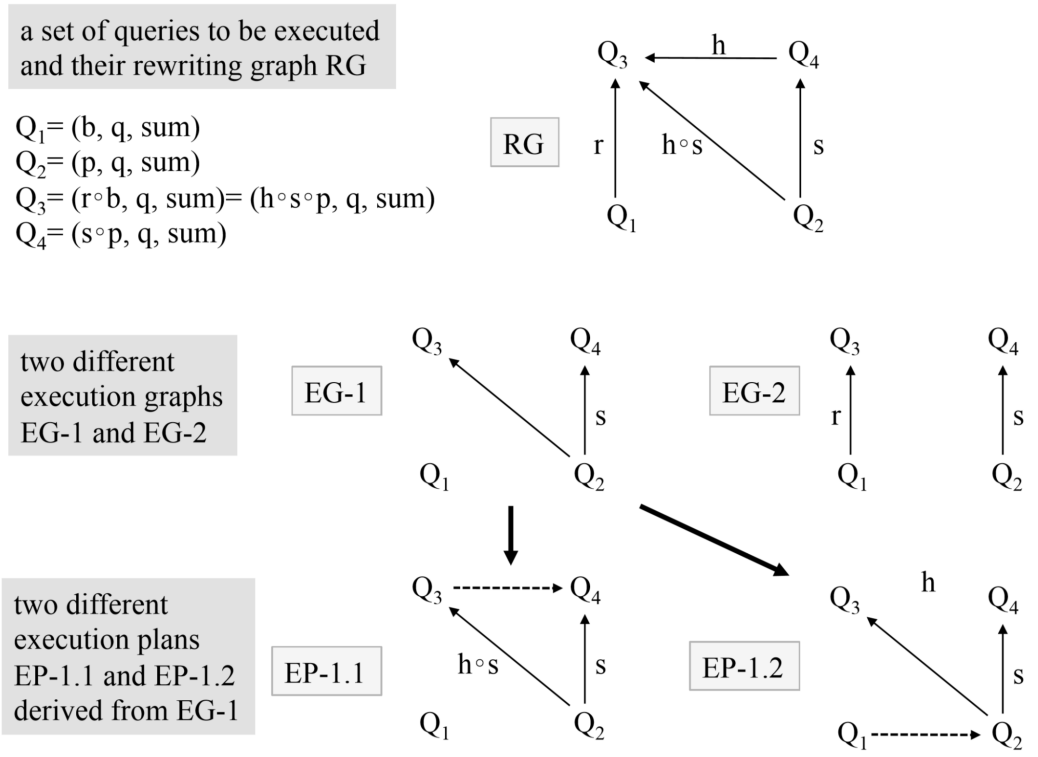


Figure 3.3: Query Rewriting Graph, Query Execution Graphs and Query Execution Plans Example

3.2.2 Conceptual Query Evaluation Scheme

The HIFUN language proposes a conceptual evaluation scheme which consists of four steps as follows:

1. **Query input preparation** Query input, denoted by *IN(Q)*, is the set of tuples which contain the useful information necessary for evaluating a query *Q*. That is the identifier *i* together with the values *g(i)* and *m(i)*, as well as the values A(i) of any attributes A contained in the restriction, in case of a restricted query. This step returns this set *IN(Q)*.

2. **Filtering (if needed)** This step is skipped in case there are no attribute restrictions. In case we have an Attribute-Restricted Query, this step filters the tuples that don't conform to the query restrictions, contained in the set *IN(Q)*.

3. π_g **construction** This step constructs the grouping partition $\pi_g = \{G_1,..., G_n\}$, as it was previously defined in the query definition. The reduction of π_g will give the answer to the query.

4. π_g **reduction** This step returns the answer of the query on each value g_j of g, previously defined as $ans_Q(g_j)$=*red(m/*G_j*, op)*, j = *1,..., n*, for every block G_j of the grouping partition π_g.

This scheme can be mapped to different lower level evaluation mechanisms supporting a big range of data set formats.

3.2.3 Query Answer Representations

The HIFUN language proposes a formal method for creating multiple representations of a query result based on Currification [23]. Briefly, when the target of the grouping g of a query is a Cartesian product, the answer of the query has more than one representation. Each different representation of a query result can reveal new "patterns" of information that might not be visible in a different representation. For example, consider the query Q = *(region* $\wedge$ *category, quantity, sum)*. The answer ans_Q of the query is the function: $ans_Q : Region \times Category \rightarrow TotQty$. The standard way of representing the answer ans_Q can be seen on the left side of Figure 3.4. However, additional representations of the answer can be produced using Currification. For example, we can generate a function $r_i : Category \rightarrow TotQty$ for each distinct value r_i of the function r. The new representation of the answer of the result can be seen on the right side of Figure 3.4. As we can see, the new representation reveals information of the answer not visible in the standard one.

The formal model of HIFUN was leveraged to develop a big data analytics system using *Apache Spark* and prove that this language is useful in practice. In the next section, we describe in detail how we implemented this system which allows a user to analyse, visualize and discover information useful for decision making, "hidden" in large-scale data sets. This system is based on HIFUN's definitions and features and performs optimizations though query rewritings and query execution plans, as proposed by HIFUN, to reduce computational costs.

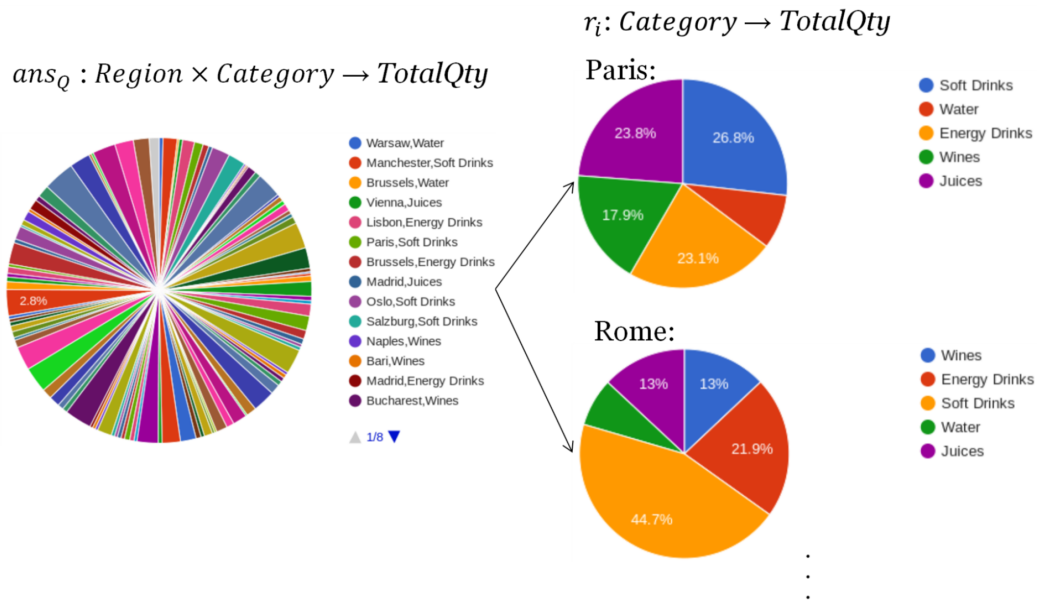


Figure 3.4: Different representations of a query answer

Chapter 4

System Implementation

As we made clear in the previous sections, an analytic query and its answer in HIFUN are defined at the conceptual level independently of the nature and the location of the data. This allows us to use it to evaluate analytic queries on more than one type of data sets, mapping analytic queries to available evaluation mechanisms. In the following paragraphs, we firstly explain in detail how we mapped HIFUN queries to existing mechanisms of Apache Spark in the physical level to evaluate analytic queries. This was done based on the HIFUN conceptual evaluation scheme, which was presented in the previous section using two different APIs: 1: the RDDs API, 2: the SPARK SQL module and the Dataset API [21]. Please note that each API supports different types of data sets and is more efficient in different cases. In any case, each data set must conform with the 2 prerequisites of the formal model as described above. Secondly, we explain how the query rewritings are applied and the query execution plans are generated, which both reduce the evaluation cost of queries. Finally, we explain how we allow a user to express analytic queries, visualise the results and explore the query answers through a user-friendly interface.

4.1 Apache SPARK RDDs API

4.1.1 Query Evaluation

In this subsection we will describe how we mapped the steps of the HIFUN conceptual evaluation scheme to existing physical level mechanisms of the Apache Spark RDDs API. The Resilient Distributed Datasets (RDDs) is the fundamental data structure in Apache Spark. Briefly, RDDs is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner [24]. Before the evaluation begins, a data set is saved in a Hadoop Distributed File System(HDFS) cluster, split in blocks saved possibly in different DataNodes.

The evaluation steps are as follows: Firstly, the analytic query is parsed and dis-

tributed to all the cluster nodes as a broadcast variable [25]. This ensures that the query is copied to each worker only once which reduces the evaluation cost. A broadcast variable was chosen because a query is read-only and useful to every worker. After this, the data set is saved as an RDD and the query evaluation scheme steps are executed.

1. **Query input preparation:** In this step, the *IN(Q)* set is computed, which is consists of tuples that contain the useful attributes of each data item. This is done using the *map()* function of Spark [26], keeping the useful attributes and storing them in Tuple objects. The Tuple object was created and serialized using the Kryo serialization. Kryo is a fast and efficient object graph serialization framework which is faster and more efficient than the standard serializer [27]. This once again results to a smaller evaluation time. The output of this step is a new RDD which contains the set of tuples as described above.

2. **Filtering:** This step filters the tuples of the RDD that don't conform to the query restrictions, if any. This is done using the filter() function of Spark [26] and the output of this step is a new RDD which contains the filtered tuples.

3. π_g **construction and** π_g **reduction:** Firstly, to construct the grouping partition π_g and the π_g reduction, the *mapToPair()* function [28] was used. The result of this function, after being applied to the tuples RDD is a new Pair RDD which contains key-value pairs *(K,V)*, where the Key of the pair is the value of the grouping attribute of each data item i (or the values of the grouping attributes if the domain of ans_Q is a cartesian product of two or more grouping attributes) and the Value of the pair is the value of the measuring attribute of each data item i. After that, the function reduceByKey() [28] is applied on the Pair RDD. This function constructs both the grouping partition π_g and the π_g reduction and creates a final Pair RDD which contains the answer of the query ans_Q. Finally, the answer is saved on a text file and can be used for visualization and further exploration.

We will now show an example of a query evaluation step-by-step. Let D be a data set whose analysis context is depicted in Figure 4.1 and let Q be an attribute-restricted query that needs to be evaluated where Q = *(p∧r/E, q, sum), E={x in D/ c(x)='Soft-Drinks'}*. Please note that the structure of the data set is just indicative. Datasets of varying forms can be analyzed using the RDDs API implementation changing only the *Input preparation* step.

Firstly, in the query input preparation step, we choose the attributes useful for the analysis. Those are the grouping attributes p and r, the measuring attribute q and the attribute c contained in the query restriction. The result of this step is the *Input Preparation RDD* which contains tuples of the form *(p(i), r(i), c(i), q(i))*.

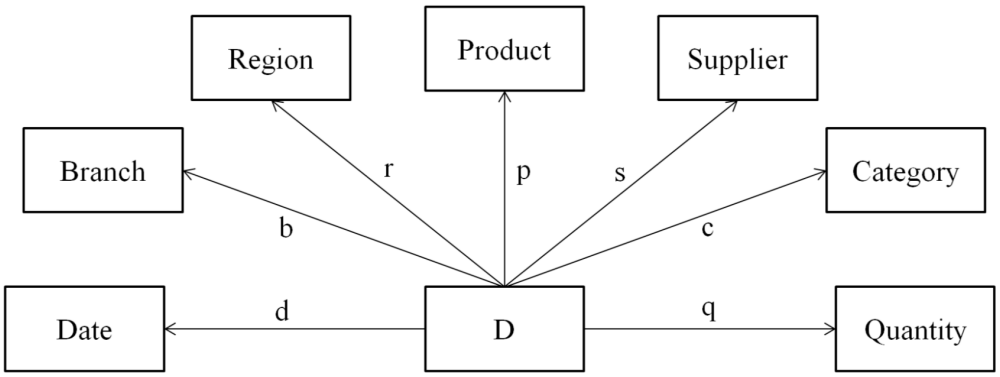


Figure 4.1: Analysis Context of the data set of the RDDs API evaluation example

This is depicted in Figure 4.2.

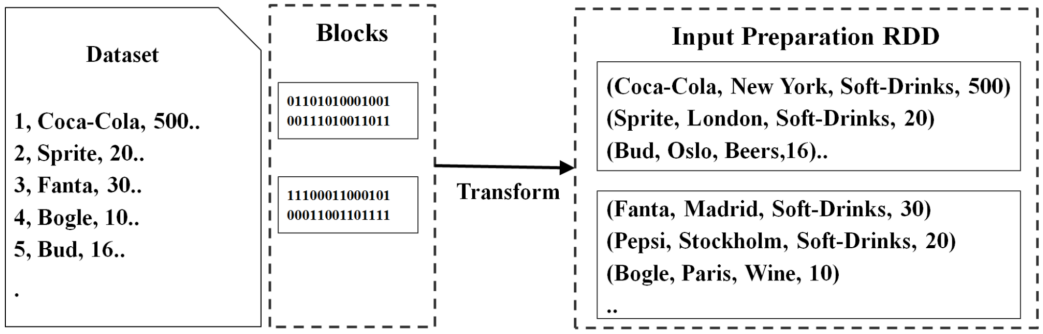


Figure 4.2: Query Input Preparation Step

Secondly, in the *Filtering step*, the tuples of the *Input preparation RDD* are filtered according to the restriction of the query: $c(x)$ = *'Soft Drinks'*. A new *RDD*, the *Filtered RDD* is created, which contains only the tuples conforming to this restriction. This is depicted in Figure 4.3. This step is skipped in the evaluation of non Attribute-Based Restricted queries.

The last step is depicted in Figures 4.4 and 4.5. The *Key-Value Pairs RDD* is created which contains (K ,V) pairs, where K the "$p(i), r(i)$" pairs of each tuple and V the $q(i)$ value of each tuple. Finally, the *Result RDD* contains the query answer $ans_Q : Product \times Region \rightarrow TotQty$ consisted of (K',V') tuples. For each distinct value that the pairing of the functions p and $r(p \wedge r)$ gives when applied to each row of D, we have one such tuple and the value V' of each such tuple is the value that the ans_Q function gives, when applied to K'.

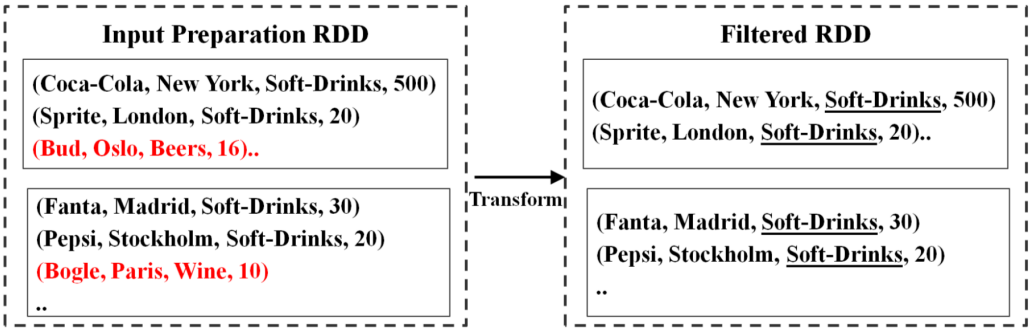


Figure 4.3: Filtering Step

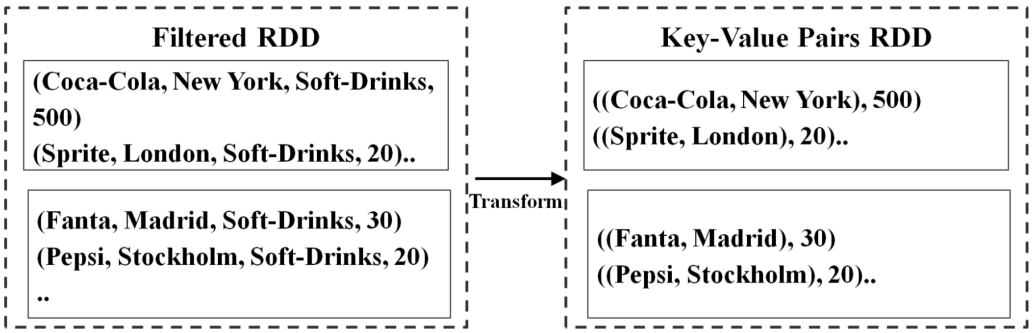


Figure 4.4: π_g construction and π_g reduction Step Part 1

4.1.2 Rewritten Set Evaluation

As we saw in the formal definition of the query language, a set Q of HIFUN queries can be rewritten according to some rules. In this subsection, we will describe how the evaluation process changes in the RDDs API for each rewriting rule.

1. $Q = \{(g, m, op_1), ...(g, m, op_n)\} \Rightarrow Q = ((g, m), \{op_1, ..., op_n\})$

   For this rewriting rule, the *Query input preparation* step and the *Filtering* step remain the same. As far as the π_g construction and π_g reduction steps are concerned, the *mapToPair()* function creates an RDD which contains key-value pairs of the form *(K, $(V_1, ..., V_n)$)*, where *$(V_1, ..., V_n)$* is the list consisted of the measuring attribute value *m(i)* of each data item i, repeated n times. After that, the function *reduceByKey()* applies the n operations on the list of values to produce the query answer.

2. $Q = \{(g, m_1, op_1), ...(g, m_n, op_n)\} \Rightarrow Q = (g, \{m_1, op_1\}, ..., \{m_n, op_n\})$

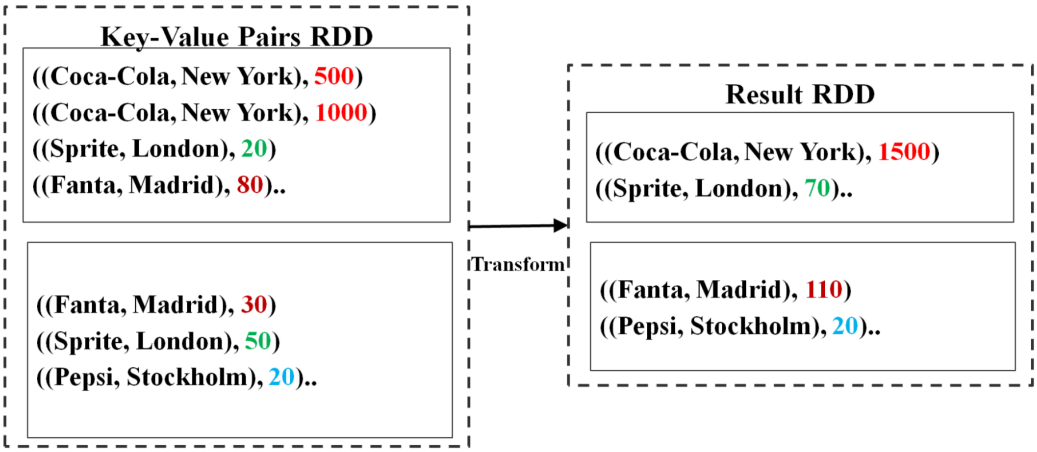


Figure 4.5: π_g construction and π_g reduction Step Part 2

The evaluation of the rewritten set of the second rewriting rule is very similar to the evaluation of the previous one. In the *Query input preparation* step, the values of the distinct measuring attributes of each data item are added to the tuples of the *IN(Q)* set. The π_g construction and π_g reduction steps are identical to the steps of the previous rule. The *mapToPair()* function creates an RDD which contains key-value pairs of the form *(K, $(V_1, ..., V_n)$)*, where *$(V_1, ..., V_n)$* is the list consisted of the values of the n measuring attributes of each data item i and the function *reduceByKey()* applies the n operations on the list of values to produce the query answer. The *Filtering* step once again remains the same.

3. $Q = \{(g_1, m, op), ...(g_n, m, op)\} \Rightarrow Q = \{(g_1 \wedge ... \wedge g_n, m, op), (proj_{G_1}, (g_1 \wedge ... \wedge g_n, m, op), op), ..., (proj_{G_N}, (g_1 \wedge ... \wedge g_n, m, op), op)\}$

For the last rewriting rule, the evaluation of the base query *$(g_1 \wedge ... \wedge g_n, m, op)$* is done as described above. Since the result of the base query is traversed n times, one for each projection query, it is cached in memory. For each projection query, we use the *mapToPair()* function to create key-value pairs of the form (K, V), where K contains the values of the subset of the grouping attributes values of the base query, which are contained in the projection query. Once again, the *reduceByKey()* function gives the projection query answer.

4.2 Apache SPARK SQL Datasets API

4.2.1 Query Evaluation

In this subsection we will describe how the HIFUN conceptual evaluation scheme can be implemented using existing physical level mechanisms of Apache Spark SQL, which is a Spark module for structured data processing. This module provides two APIs, the DataFrames and the Datasets. A DataFrame is conceptually equivalent to a table in a relational database, but it comes with richer optimizations as Spark evaluates transformations lazily. More specifically, it is a distributed collection of data, like RDD, but organized into named columns. This provides Spark with more information about the structure of both the data and the computation. Such information can be used for extra optimizations. As far as the Datasets API is concerned, it is a strongly typed, immutable collection of objects that are mapped to a relational schema. It provides the benefits of RDDs with the benefits of Spark SQL's optimized execution engine and fast in-memory encoding [21]. The latter was used for the evaluation of the HIFUN queries.

The evaluation steps follow the conceptual mapping of HIFUN queries to SQL group-by queries as given in [1][2]. This conceptual mapping is shown in Figure 4.6 and it is implemented in our system as follows: the SPARK SQL provides the option of running SQL queries programmatically on Datasets. As a result, a HIFUN analytical query can be translated to an SQL query, which can give the query result when executed [29].

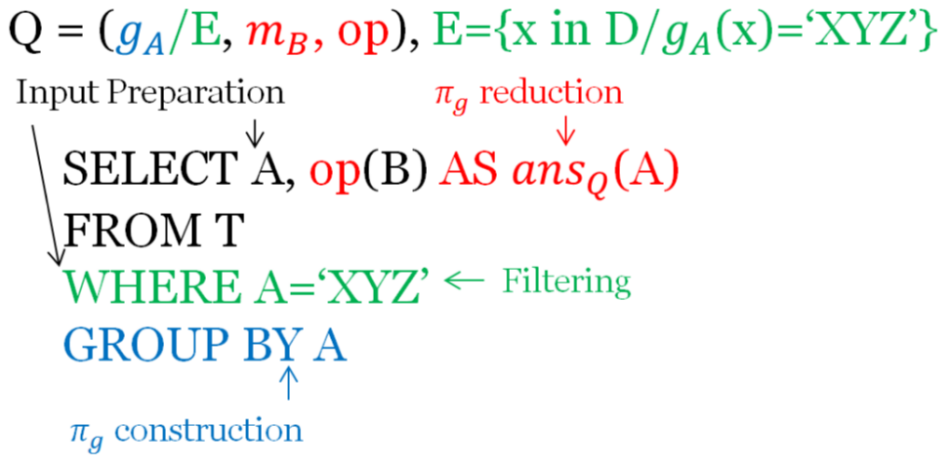


Figure 4.6: Correspondence between the conceptual evaluation scheme and the evaluation using an SQL query

We will now show an example of a query evaluation step-by-step. Let D be a data set whose analysis context is depicted in Figure 4.7 and let Q be an attribute-restricted query that needs to be evaluated where $Q=((r \circ b) \wedge (c \circ p)/E, q, sum)$, $E=\{x \text{ in } D/\ r \circ b(x)!='Athens'\}$. The data set D is consisted of 3 relational tables, as shown in the figure. The mapping of the query Q to an SQL query is shown in the Figure 4.7. Firstly, in the *Input preparation* step the useful attributes are selected which are the grouping attributes *Region*, *Category* and the measuring attribute *Quantity*. The *Region* attribute is also included in the query restriction. Some of these attributes are contained in the upper-level tables of the analysis context so the tables are joined accordingly. Secondly, in the *Filtering* step, the HIFUN restrictions are translated to SQL restrictions using the *"Where"* clause in the obvious way. Thirdly, in the π_g *construction* step, the grouping partition as defined in the formal model is constructed using the *"GROUP BY"* clause, grouping by the two grouping attributes *Region* and *Category*. Finally, the π_g *reduction* step is implemented by applying the query operation *SUM* on the measuring attribute *Quantity*. The answer of the constructed *SQL* query is the answer of the HIFUN query Q, ans_Q.

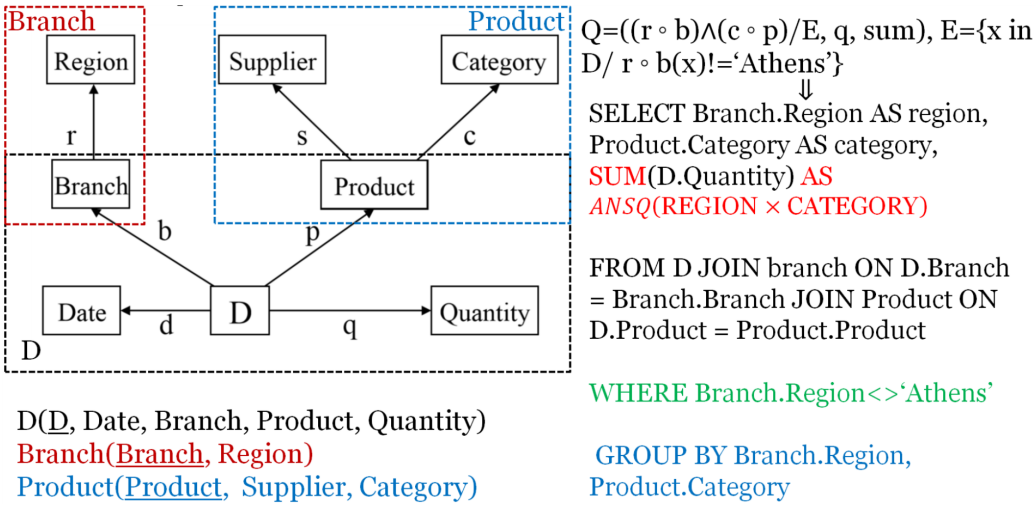


Figure 4.7: Apache Spark SQL Dataset API query evaluation example

4.2.2 Rewritten Set Evaluation

Once again, a set Q of HIFUN queries can be rewritten according to some rules. In this subsection, we will describe how the evaluation process changes in the Datasets API for each rewriting rule.

1. $Q = \{(g, m, op_1), ...(g, m, op_n)\} \Rightarrow Q = ((g, m), \{op_1, ..., op_n\})$

For this rewriting rule, the only step changing is the π_g *reduction* step making small changes on the SQL query. More specifically, we apply the aggregate functions related to the n operations on the measuring attribute m. This is depicted in Figure 4.8.

2. $Q = \{(g, m_1, op_1), ...(g, m_n, op_n)\} \Rightarrow Q = (g, \{m_1, op_1\}, ..., \{m_n, op_n\})$

The evaluation of the rewritten set of the second rewriting rule is very similar to the evaluation of the previous one. Once again, we change the π_g *reduction* part of the SQL query, applying the n aggregate functions on the corresponding measuring attributes. This is depicted in Figure 4.8.

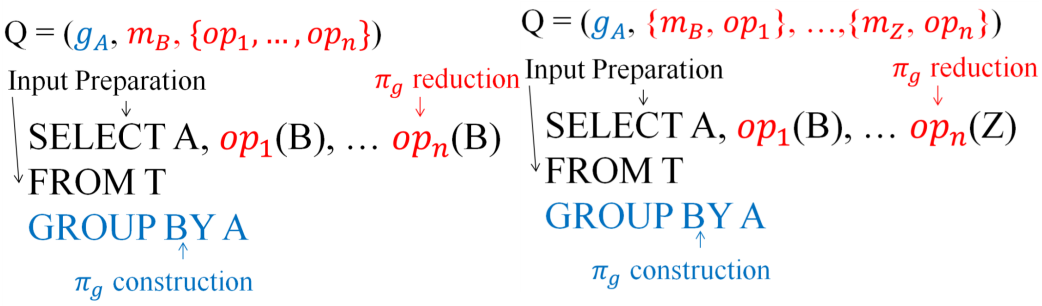


Figure 4.8: First and Second Rewriting Rule Datasets API

3. $Q = \{(g_1, m, op), ...(g_n, m, op)\} \Rightarrow Q = \{(g_1 \wedge ... \wedge g_n, m, op), (proj_{G_1}, (g_1 \wedge ... \wedge g_n, m, op), op), ..., (proj_{G_N}, (g_1 \wedge ... \wedge g_n, m, op), op)\}$

For this rewriting rule, the result table of the base query $(g_1 \wedge ... \wedge g_n, m, op)$ is cached for faster evaluation of the projection queries. The mapping of both the base query and the projection queries to SQL queries is shown in Figure 4.9.

4. $Q = \{(f, m, op), (g \circ f, m, op)\} \Rightarrow Q = \{(f, m, op), (g, (f, m, op), op)\}$

In this case, for the evaluation of the second query, the answer table of the first query, *(f, m, op)*, is joined with the table containing the grouping attribute g and the aggregation function is applied on the column containing the result of the aggregation of the first query. The mapping of both queries to SQL queries is shown in Figure 4.10.

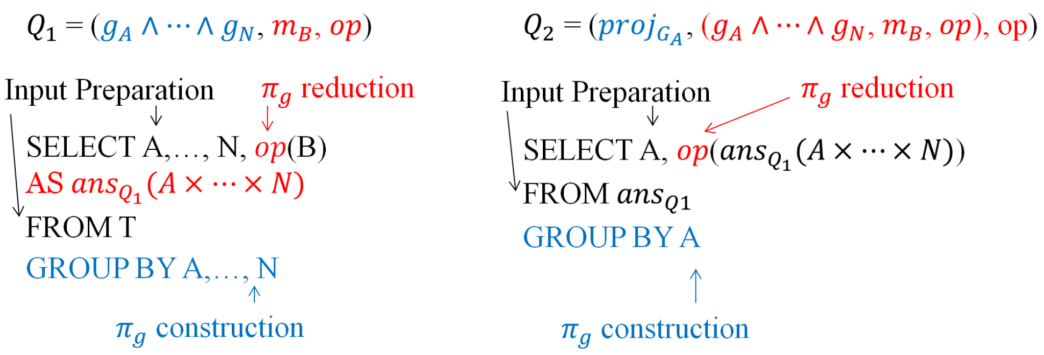


Figure 4.9: Third Rewriting Rule Datasets API

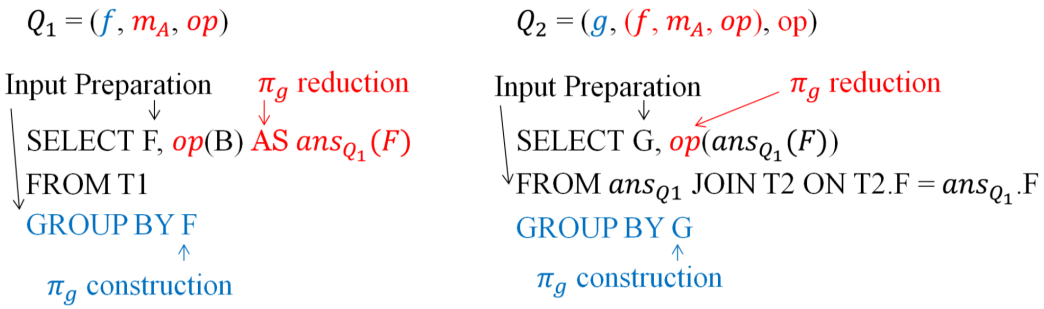


Figure 4.10: Basic Rewriting Rule Datasets API

4.3 Query Rewriting and Query Execution Plans Algorithm

In the formal model of the HIFUN language, the concepts of the *Query Rewriting Graph*, the *Query Execution Graph* and the *Query Execution Plan* were introduced [1]. Briefly, these concepts were proposed to reduce the evaluation cost of a set of queries Q. However, the problem of choosing the most effective rewriting when a given query has two or more different rewritings in terms of other queries in the set Q, and the problem of choosing between possible *Query Execution Graphs* is not tackled. In this subsection, we explain how we tackled these problems and describe an algorithm which applies the proposed *Query Rewritings* and generates the *Query Execution Plan* of a set Q of queries.

Firstly, the rewritten queries of the first and the second rewriting rule are evaluated in the same way as described above, so the effectiveness of them in reducing the evaluation cost is the same. This can be also proved experimentally as shown in the next section. For this reason, every time we have two or more queries in the rewriting set with the same grouping, some common and some different measuring attributes and we have to choose between the first and the second rewriting rule, we always choose the second one. For example, let Q be a set of queries where Q

= (g, m_1, op_1),(g, m_1, op_2), (g, m_2, op_3). In this case, the first two queries could be rewritten as (g, $m_1, \{op_1, op_2\}$) but this would let the third query to be evaluated individually which has a higher cost. For this reason, in these cases, we always choose the second rewriting rule to make sure none of the queries is evaluated individually. In our example, Q would be rewritten as Q = {(g, $\{m_1, op_1\}$, $\{m_1, op_2\}$, $\{m_2, op_3\}$)}.

Secondly, the first two rewriting rules are more effective in reducing the evaluation cost than the third rewriting rule. This once again is proved experimentally and can be seen in the next section. This is explained as follows: the third rewriting rule performs one grouping for each projection query of the set. A grouping is a shuffle operation which creates network traffic and has high cost [21]. This is more expensive than performing more than one operations on the measuring values, as this does not depend on the network and executes fast in the memory of the nodes of the cluster. For this reason, when we have two or more queries and a query can be rewritten using the remaining queries, then if we have to choose between the first and the third or the second and the third rewriting rule, we always keep the third one as the last choice. The third rewriting rule is chosen only if we have no alternatives and this reduces the evaluation cost. For example, let Q be a set of queries where Q = {(g, m_1, op_1),(g, m_1, op_2), (f, m_1, op_1)}, where op_1 a distributive operation. In this case, Q would be rewritten as Q = {(g, $m_1, \{op_1, op_2\}$), (f, m_1, op_1)}.

We will now describe the steps which are followed to apply the *Query Rewritings* and produce the *Query Execution Plan* for a set Q of queries.

Algorithm: Query rewritings and execution plan generation

Input: A set Q of queries

Output: The possibly rewritten queries and the execution plan for the set Q

1. The queries of the set Q are rewritten based on the rewriting rules excluding the basic rewriting rule.

   (a) The set Q is traversed once, keeping the number of distinct measuring attributes and distinct operations per grouping g. This will help to ensure that the choice rules as described above are followed. Moreover, in case there are equality constraints in the analysis context, if the grouping attributes or the measuring attribute have alternatives according to the equality constraints then we change them accordingly while traversing so that these queries can be used for rewritings using equality constraints which were proposed in the formal model.

(b) A new empty set *Q'* is created

(c) For every query q of Q: If *Q'* is empty we add the query q to Q'. If not, we start traversing *Q'*:

-If a query *q'* with the same grouping g and the same measuring attribute m is found then we check if this grouping has a single measuring attribute in the set. If yes, then we choose the first rewriting rule and add the operation of q to *q'*. If not, we choose the second rewriting rule and add the measuring attribute m and the operation of q to *q'*.

-If a query *q'* with the same grouping g and different measuring attributes is found then we choose the second rewriting rule and add the measuring attribute m and the operation of q to *q'*.

-If a query *q'* with the same measuring attribute m, the same distributive operation *op* and different groupings g_1, g_2 with the same source is found then we check if both groupings have single measuring and single operation attributes in the set. If yes, we add the grouping attributes of grouping g_1 not contained in g_2 to g_2 and add the corresponding projection queries in the set Q'. If not, we keep traversing the set *Q'*.

-If a query is already rewritten and q can join the rewriting without violating the choice rules, then it is done accordingly.

-If none of the above hold, we add the query q to *Q'*.

2. We traverse *Q'* and check every grouping g . If grouping g has alternatives according to possible equality constraints then if g cannot be used to allow any rewritings based on the basic rewriting rule, we search if the grouping alternatives can be used to allow more basic rule rewritings. The difference between this step and step *1.a* is that in this step we check equality constraints to allow *basic rewriting rule* rewritings only. In case there are different measurings which are equal according to the equality constraints and they can be used for *basic rewriting rule* rewritings, they are already changed in step *1.a*.

3. A direct acyclic graph G is created, whose nodes are the queries of the set *Q'*. In case there are projection queries, edges are added from the base queries to the projection queries accordingly.

4. The possibly rewritten queries of *Q'* are partitioned into subsets S_i, based on the depth i of the grouping g of each query. This depth is equal to the number of compositions contained in a grouping g of a query. In case a query

has more than one grouping attributes then we add the query to a subset S_i, only if the grouping attributes have common source. If not, the queries cannot be rewritten based on the basic rewriting rule.

5. Starting from the subset S_m, where m is the maximum depth of the queries of the set Q and going back to S_l, where l is the first depth bigger that the minimum depth of the queries of the set Q: for every query *q"* in each subset we start searching all the previous subsets from the highest to the lowest depth, for queries *q"'* that can be used to rewrite *q"*. In case we find one, we rewrite the queries, add a labelled edge in the graph G from the node representing the query *q"'* to the node representing the node *q"* and continue with the next query of the subset. More specifically, if C is an analysis context with f : A $\rightarrow$ B, g : B $\rightarrow$ C, h : B $\rightarrow$ D three edges of C, the queries are rewritten as follows:
-If *q" = (g ∘ f, m, op)* and *q"'= (f, m, op)*, then *q"* is rewritten as *(g, $ans_{q'''}$, op)*
-If *q" = (g ∘ f ∧ ... ∧ h ∘ f, m, op)* and *q"'= (f, m, op)*, then *q"* is rewritten as *(g∧... ∧ h, $ans_{q'''}$, op)*
-If *q" = (g ∘ f, m, op_x)* and *q"'= (f, m, { op_1..., op_n})*, where op_x one of the operations of *q"'* , then *q"* is rewritten as *(g, $ans_{q'''}$, op_x)*
-If *q" = (g ∘ f, m_x, op_x)* and *q"'= (f, $\{m_1, op_1\}, ..., \{m_n, op_n\}$)*, where m_x one of the measuring attributes of *q"'* and op_x one of the operations of *q"'* applied on m_x in *q"'*, then *q"* is rewritten as *(g, $ans_{q'''_{m_x}}$, op_x)*
A query *q"'* can also be a projection query produced by the third rewriting rule and in this case *q"* can also be rewritten using *q"'* in the obvious way. We assume that every *op* in this step is distributive.

6. The topological sorting of the direct acyclic graph created is used to compute the query execution plan of the set Q [1].

We will now show an indicative example of how this algorithm works to produce the execution plan for a specific set of queries. Let Q be a set of queries where $Q = \{(a, m_1, op_1), (d, m_1, op_1), (a, m_1, op_2), (b, m_1, op_1), (b, m_1, op_2), (b, m_2, op_3), (c, m_1, op_1), (e \circ f, m_1, op_3), (g \circ a, m_1, op_1), (h \circ b, m_2, op_3), (i \circ b, m_2, op_3), (j \circ k \circ l, m_1, op_1)\}$ and let op_1, op_3 distributive operations with the following equality constraints holding: $e \circ f = a$ and $j \circ k \circ l = m \circ d$.

The steps of the algorithm are the following:

1. (a) The number of distinct measuring attributes and distinct operations per grouping g is kept. For example, grouping b has two distinct measuring attributes: m_1 and m_2, and three operations: op_1, op_2 and op_3. Checking the first equality constraint we notice that there are two queries with grouping a and one query with grouping $e \circ f$. The grouping of the query $(e \circ f, m_1, op_3)$ is changed to a while traversing to allow

more rewriting rules. The second constraint does not produce a useful rewriting according to the rewriting rules excluding the basic rewriting rule.

(b) Q'={}

(c) We start traversing Q:

i. (a, m_1, op_1): Q' = $\{(a, m_1, op_1)\}$
Q' is empty so we add the query to the set.

ii. (d, m_1, op_1): Q' = $\{(a, m_1, op_1), (d, m_1, op_1)\}$
The query (a, m_1, op_1) is found which has the same measuring, the same distributive operation and different grouping with the same source. Grouping a has three distinct operations so we keep traversing and since there are no other queries inside, the query is added to *Q'*.

iii. (a, m_1, op_2): Q' = $\{(a, m_1, \{op_1, op_2\}), (d, m_1, op_1)\}$
The query (a, m_1, op_1) is found which has the same grouping and measuring. Grouping a has a single measuring attribute, so op_2 is added to the query.

iv. (b, m_1, op_1): Q' = $\{(a, m_1, \{op_1, op_2\}), (d, m_1, op_1), (b, m_1, op_1)\}$
The query (d, m_1, op_1) is found which has the same measuring, the same distributive operation and different grouping with the same source. Grouping b has two distinct measurings so it is skipped.

v. (b, m_1, op_2): Q' = $\{(a, m_1, \{op_1, op_2\}), (d, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}))\}$
The query (b, m_1, op_1) is found which has the same grouping and the same measuring. Grouping b has two distinct measurings so the second rewriting rule is chosen and the measuring m1 and the operation op2 are added to the query.

vi. (b, m_2, op_3): Q' = $\{(a, m_1, \{op_1, op_2\}), (d, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}))\}$
The query $(b, (\{m_1, op_1\}, \{m_1, op_2\}))$ is found and (b, m_2, op_3) can join the rewriting so its measuring and operation are added.

vii. (c, m_1, op_1): Q' = $\{(a, m_1, \{op_1, op_2\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1))\}$
The query (d, m_1, op_1) is found which has the same measuring, the same operation and different grouping with the same source. Grouping c is added to $((d, m_1, op_1)$ and the two projection queries are added to the set *Q'*.

viii. (a, m_1, op_3): Q' = $\{(a, m_1, \{op_1, op_2, op_3\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1))\}$
The query $(a, m_1, \{op_1, op_2\})$ is found and since it is already rewritten according to the first rewriting rule and (a, m_1, op_3) can join the

rewriting, op_3 is added. We remind that the grouping of this query was changed using the equality constraints to allow this rewriting.

ix. $(g \circ a, m_1, op_1)$: Q' = $\{(a, m_1, \{op_1, op_2, op_3\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1)), (g \circ a, m_1, op_1)\}$
No query with common grouping or different grouping with common source.

x. $(h \circ b, m_2, op_3)$: Q' = $\{(a, m_1, \{op_1, op_2, op_3\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1)), (g \circ a, m_1, op_1), (h \circ b, m_2, op_3)\}$
No query with common grouping or different grouping with common source.

xi. $(i \circ b, m_2, op_3)$: Q' = $\{(a, m_1, \{op_1, op_2, op_3\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\}), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1)), (g \circ a, m_1, op_1), ((h \wedge i) \circ b, m_2, op_3), (proj_H, ((h \wedge i) \circ b, m_2, op_3), op_3), (proj_I, ((h \wedge i) \circ b, m_2, op_3), op_3)\}$
The query $(h \circ b, m_2, op_3)$ is found which has the same measuring, the same distributive operation and different grouping with the same source. The grouping $(i \circ b)$ is added to the query and two projection queries are added to the set.

xii. $(j \circ k \circ l, m_1, op_1)$: Q' = $\{(a, m_1, \{op_1, op_2, op_3\}), (d \wedge c, m_1, op_1), (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\})), (proj_D, (d \wedge c, m_1, op_1), op_1), (proj_C, (d \wedge c, m_1, op_1), op_1)), (g \circ a, m_1, op_1), ((h \wedge i) \circ b, m_2, op_3), (proj_H, ((h \wedge i) \circ b, m_2, op_3), op_3), (proj_I, ((h \wedge i) \circ b, m_2, op_3), op_3), (j \circ k \circ l, m_1, op_1)\}$. No query with common grouping or different grouping with common source.

2. The set *Q'* is traversed and we notice that the query $(j \circ k \circ l, m_1, op_1)$ has a grouping alternative according to the equality constraints of the context and the current grouping $j \circ k \circ l$ cannot be used for any *basic rewriting rule* rewriting. For this reason, the grouping of the query is changed to $m \circ d$ since there is a query in the set *Q'* with the same measuring, the same distributive operation and a grouping contained in the source of $m \circ d$.

3. A direct acyclic graph G is created, whose nodes are the queries of the set *Q'*. Labels are added from the base queries to the projection queries accordingly.

4. Two subsets are created, S_0 containing the queries of *Q'* with grouping depth 0 and S_1 containing the queries of *Q'* with grouping depth 1. More specifically: $S_0 = \{q'_1 = (a, m_1, \{op_1, op_2, op_3\}), q'_2 = (d \wedge c, m_1, op_1), q'_3 = (b, (\{m_1, op_1\}, \{m_1, op_2\}, \{m_2, op_3\})), q'_4 = (proj_D, ans_{q'_2}, op_1), q'_5 = (proj_C, ans_{q'_2}, op_1)\}$ and $S_1 = \{q'_6 = (g \circ a, m_1, op_1), q'_7 = ((h \wedge i) \circ b, m_2, op_3), q'_8 = (proj_H, ans_{q'_7}, op_3), q'_9 = (proj_I, ans_{q'_7}, op_3), q'_{10} = (m \circ d, m_1, op_1)\}$

5. We check if any queries in subset S_1 can be rewritten using any queries in S_0. The query $q'_6 = (g \circ a, m_1, op_1)$ can be rewritten using the query q'_1 so it is rewritten as $q'_6 = (g, ans_{q'_1}, op_1)$. An edge is added from the node representing the query q'_1 to the node representing the query q'_6 with a label g. The query $q'_7 = ((h \wedge i) \circ b, m_2, op_3)$ can be rewritten using the query q'_3 so it is rewritten as $q'_7 = (h \wedge i, ans_{q'_{3_{m_2}}}, op_3)$. An edge is added from the node representing the query q'_3 to the node representing the query q'_7 with a label $h \wedge i$. The query $q'_{10} = (m \circ d, m_1, op_1)$ can be rewritten using the query q'_4 so it is rewritten as $q'_{10} = (m, ans_{q'_4}, op_1)$. An edge is added from the node representing the query q'_4 to the node representing the query q'_{10} with a label m.

6. The topological sorting of the graph gives three query levels $L_i, i = 0, 1, 2$ [1]. The queries contained in a level i + 1 are evaluated after the evaluation of the queries contained in a level i. More specifically: $L_0 = \{q'_1, q'_2, q'_3\}$, $L_1 = \{q'_4, q'_5, q'_6, q'_7\}$ and $L_2 = \{q'_8, q'_9, q'_{10}\}$. The final result is depicted in Figure 4.11.

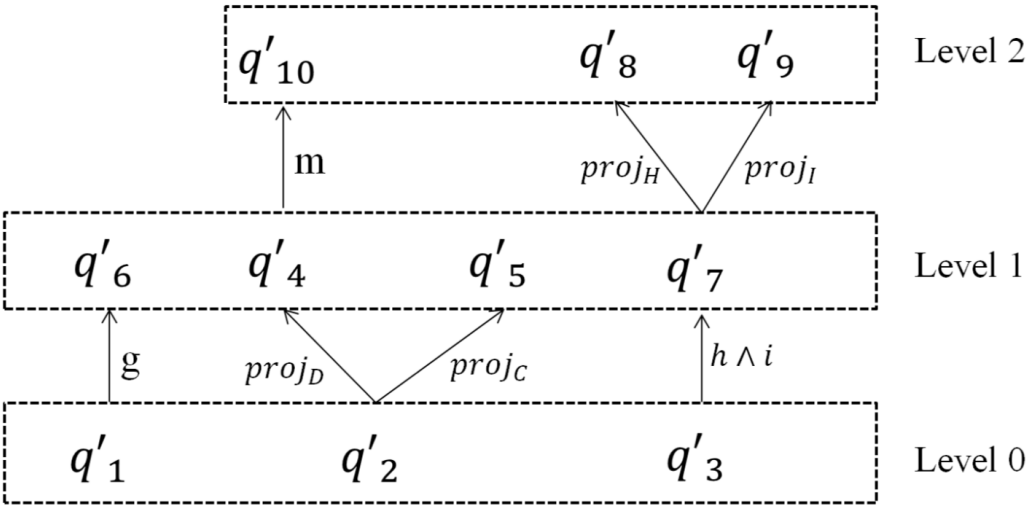


Figure 4.11: Final Result of the Query Execution Plans Generation Algorithm Example

4.4 User Interface

In this subsection, we are going to describe the user interface implemented for this system, which allows a user to express analytic queries, visualise the results and

explore the query answers.

Firstly, as we made clear, a HIFUN query consists of three parts only: the grouping part, the measuring part, and an operation applicable on the measuring attribute. Based on this simple structure, a very simple user interface can be created which allows a user of any familiarity with data analytics to express an analytic query. Our system provides a step-by-step query constructor which allows a user to select the attributes and the operation needed for the analysis, and possibly some restrictions on these attributes in a user-friendly way. By doing this, we allow anyone interested in analysing some data to reveal information useful for decision making, which may be "hidden" in them. For example, a financial advisor of an airline who has data related to ticket bookings, would like to maximise the profits of the company by proposing the introduction of new flights to popular destinations. This system could allow him to reveal useful information hidden in these data(e.g."Which flight of the airline has the most profits per month?", "Which airport gets the most visitors per year?"), which can help him with decision-making. Our system uses the language as a tool for the expression of analytic queries but does not depend on a user being familiar with it in order to be able to use it. Moreover, it automates the process of query rewriting and the execution plans generation, providing to the user the answers of a set of queries as if they were evaluated individually. For the visualisation of the answer of an analytic query, charts from the *Google Charts API* were used.

4.4.1 Query Answer Visual Exploration

In this last subsection, we describe how our system allows a user to explore the answer of a query. Firstly, in order to allow the exploration of all the different representations of an answer of a query, the method of *Currification*[23] was used, which was explained in section 3. A user can switch from one representation to another by selecting the grouping attributes of interest through the system's interface.

Secondly, we allow a user to filter the answer ans_Q of a query Q, according to some restrictions of the grouping attributes. This is equivalent to the Result-Restricted queries as defined in the formal model. We implement this by running Spark, caching the answer and applying the filters on the answer. The user can apply multiple filters, meaning that a filtered answer can be re-filtered using additional filters. The user can also undo the filtering and apply new restrictions from scratch. The Spark application is terminated only when the user finishes with answer exploration and this is done to ensure that the answer remains in cache and is filtered efficiently. Our system allows the user to combine *Filtering* and *Currification* to help him reveal useful information "hidden" in the answers of the queries as much as possible.

We end this section with a remark regarding the query rewritings. Our system

implements all of the rewritings proposed by HIFUN's formal model. However, no evaluation of the benefits resulting from the formal rewritings has been conducted before in the context of HIFUN. We do this in the following section.

Chapter 5

Evaluation

In this section, we describe the experiments that we conducted to evaluate the effectiveness of the query rewritings, as described above. By effectiveness, we mean the percentage decrease of the time needed to evaluate a set of queries using the proposed rewriting rules. These experiments were performed over synthetic data sets stored in the Hadoop Distributed File System (HDFS). Their schema and their contents are described below in detail. The procedures were ran over a cluster consisted of 4 nodes. Each node had 38 cores (19 physical and 19 virtual) at 2.2 Ghz, 245 GB RAM and storage capacity of 415 GB.

5.1 Apache Spark RDDs API Rewritings Evaluation

In order to evaluate the effectiveness of the query rewritings in the RDDs API implementation, a synthetic data set D was used. The analysis context of this data set is depicted in Figure 5.1 and it contains delivery invoices in a distribution center. More specifically, there are 7 attributes in D and they follow a uniform distribution. The size of the data set D is 15 GB and contains 225M data items. Finally, each attribute's target is a set of around 50 distinct values.

1. $Q = \{(g, m, op_1), ...(g, m, op_n)\} \Rightarrow Q = ((g, m), \{op_1, ..., op_n\})$

   In order to evaluate this rewriting rule, we randomly chose one attribute of the data set D as the grouping attribute g, one attribute as the measuring attribute m and n operations op_i applicable on the attribute m, to create a set Q of n queries. Firstly, we evaluated the queries of Q one by one and then we repeated the evaluation according to the rewriting rule. We ran the experiments for a set Q of cardinality n=2, gradually increasing it to cardinality n=5, to observe how the effectiveness of the rewriting rule changes as more queries participate in the rewriting of the set Q.

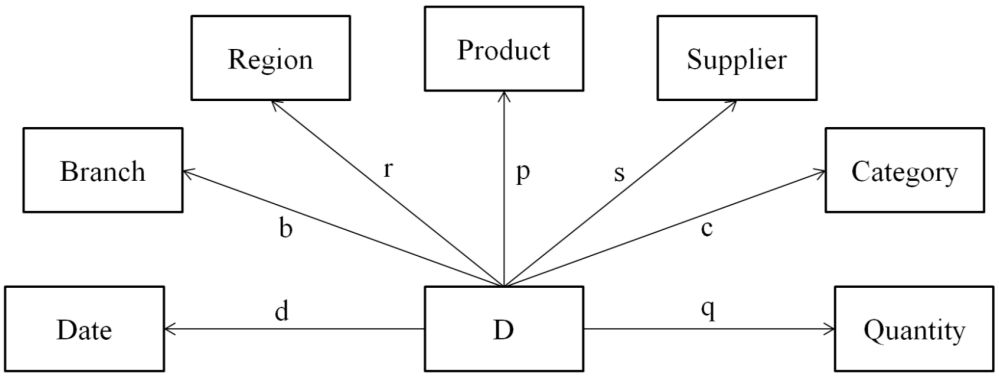


Figure 5.1: Analysis Context of the data set of the Apache Spark RDDs API evaluation

The results of this evaluation are shown in detail in Tables 5.1 and 5.2.

Table 5.1 shows the evaluation time of the queries, when evaluated individually. The first column shows the id of each query, the second shows the evaluation time of each query, and each row in the last column shows the sum of the evaluation time of all the queries until that row. The last column was compared to the evaluation time of the queries after the rewriting to calculate the percentage decrease of the queries' evaluation time.

Query ID	Evaluation Time	Evaluation Time Total
1	8.8s	8.8s
2	8.78s	17.58s
3	8.76s	26.34s
4	8.68s	35.02s
5	12.12s	47.14s

Table 5.1: First Queries Evaluation Results RDDs API

Table 5.2 shows the evaluation time of the queries, when evaluated according to the rewriting rule. The first column shows the cardinality of the set Q (the number of queries participating in the rewriting). The elements of Q are the queries that were evaluated individually before, whose results are shown in Table 5.1. The second column shows the evaluation time of the set after the rewriting. Finally, the last column shows the percentage decrease of the evaluation time that was achieved through this rewriting rule.

The startup time of Apache Spark was fixed and equal to 9.5 seconds and was not included in the evaluation time of the queries.

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	9s	48.8%
3	9.44s	64.2%
4	9.84s	71.9%
5	12.75s	73%

Table 5.2: First Rewriting Evaluation Results RDDs API

As we can see from the evaluation results in Table 5.2, a remarkable percentage decrease in time is achieved. Moreover, it can be easily observed that the more queries participating in the rewriting set, the more effective the rewriting rule is. This was expected as the grouping and the measuring step is done only once and the n reduction operations are applied to the result. Reducing the number of times that the data set D is read to the absolute minimum is of great benefit to the reduction of the evaluation cost.

2. $Q = \{(g, m_1, op_1), ...(g, m_n, op_n)\} \Rightarrow Q = (g, \{m_1, op_1\}, ..., \{m_n, op_n\})$

For this rewriting rule, we randomly chose one attribute of the data set D as the grouping attribute g, n measuring attributes m_i and n operations op_i, where op_i is applicable on the corresponding measuring attribute m_i, to create a set Q of n queries. The evaluation process which was followed was identical to the process of the previous rewriting rule.

The results of this evaluation are shown in detail in Tables 5.3 and 5.4

Similarly, with the help of the second rewriting rule, we managed to achieve a percentage decrease in time which cannot be easily ignored. Like in the previous rewriting rule, the effectiveness of this rule is increasing as we add more queries to the set Q. Once again, this was expected as the grouping

Query ID	Evaluation Time	Evaluation Time Total
1	8.52s	8.52s
2	8.52s	17.04s
3	8.8s	25.84s
4	8.74s	34.58s
5	8.42s	43s

Table 5.3: Second Queries Evaluation Results RDDs API

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	8.84s	48.12%
3	9.6s	62.85%
4	10.26s	70.33%
5	10.74s	75.02%

Table 5.4: Second Rewriting Evaluation Results RDDs API

is done only once whereas the n measuring and reduction steps are applied to the result of grouping. Comparing the percentage decrease in time of the first two rewriting rules, we can easily notice an obvious similarity between the two results. This is due to the fact that the two rewriting rules are in practice evaluated in the same way.

3. $Q = \{(g_1, m, op), ...(g_n, m, op)\} \Rightarrow Q = \{(g_1 \wedge ... \wedge g_n, m, op), (proj_{G_1}, (g_1 \wedge ... \wedge g_n, m, op), op), ..., (proj_{G_N}, (g_1 \wedge ... \wedge g_n, m, op), op)\}$

   For this rewriting rule, we randomly chose n attributes of the data set D as grouping attributes g_i, one measuring attribute m and one distributive operation op applicable on the measuring attribute, to create a set Q of n queries. The evaluation process which was followed was identical to the process of the previous rewriting rule and the set Q was of size n=2, gradually increased to n=4. The size of the result of the intermediate query $(g_1 \wedge ... \wedge g_n, m, op)$ was around 100K tuples. The size of this result depends on the number of the distinct values of the co-domain of the n grouping attributes.

   The results of this evaluation are shown in detail in Tables 5.5 and 5.6.

Once again, the third rewriting rule's results were satisfactory. The results of the evaluation show that this rewriting rule succeeds in reducing the evalua-

Query ID	Evaluation Time	Evaluation Time Total
1	8.56s	8.56s
2	8.88s	17.44s
3	8.86s	26.3s
4	8.44s	34.74s

Table 5.5: Third Queries Evaluation Results RDDs API

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	11.86s	32%
3	15.8s	39.92%
4	16.7s	51.93%

Table 5.6: Third Rewriting Evaluation Results RDDs API

tion time of the queries. Like in the previous rewriting rules, increasing the cardinality of the set Q increases the percentage reduction in the evaluation time. This was once again expected, as in this rewriting rule the data set D is traversed only once instead of n times. Applying n projections on the result of the intermediate query reduces the evaluation cost as the result is smaller than the whole data set D.

To sum up, the preceding experiments that were conducted to evaluate the effectiveness of the proposed rewritings prove that the theory is verified in practice using this implementation. The reductions in the evaluation costs show that these rewritings were worth the time taken to implement. The percentage reductions are remarkable, keeping in mind that the data sets we are dealing with can have exceedingly high volume and any unnecessary computations should be avoided at any cost.

5.2 Apache Spark SQL Datasets API Rewritings Evaluation

As we made clear in the previous sections, each HIFUN analytic query and its answer are defined at the conceptual level independently of the nature and location of data, and our system offers two implementations to support a wide range of data set formats. As the evaluation process is different in each implementation, we also need to evaluate whether the query rewritings are effective in case we use the Spark SQL Datasets API implementation. For this reason, we need to repeat the former experimental evaluation and for this purpose we used a data set structured according to a relational schema. For the new experiments, a new

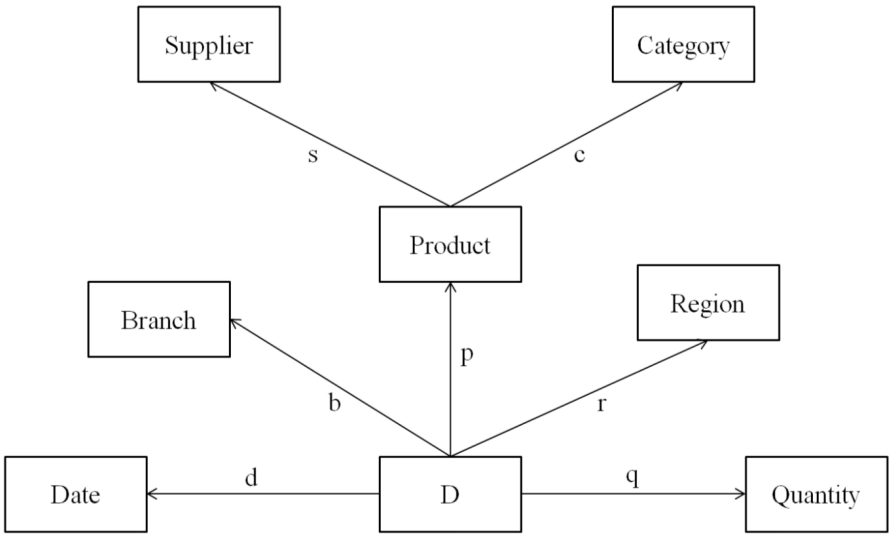


Figure 5.2: Analysis Context of the data set of the Apache Spark SQL Datasets API evaluation

synthetic data set D was used which is consisted of a number of files. Each file represents a relational table. The analysis context of the new data set is depicted in Figure 5.2 and once again contains delivery invoices in a distribution center. More specifically, it contains 2 relational tables, the base table D and one extra table *Product*. The primary key of the D table is the delivery id D and the primary key of the *Product* table is the product id *Product*. The first table contains five attributes, the second contains two attributes and the attributes' co-domain follow a uniform distribution. The size of the base table D is 15GB and contains 130M tuples. All of the attributes were chosen from the base table for the following experiments. Finally, the evaluation steps followed were the same for each rewriting rule as described above.

1. $Q = \{(g, m, op_1), ...(g, m, op_n)\} \Rightarrow Q = ((g, m), \{op_1, ..., op_n\})$

   The results of this evaluation are shown in detail in Tables 5.7 and 5.8

Query ID	Evaluation Time	Evaluation Time Total
1	8.85s	8.85s
2	8.83s	17.68s
3	9.01s	26.69s
4	8.83s	35.52s
5	9.04s	44.56s

Table 5.7: First Queries Evaluation Results Datasets API

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	8.92s	49.55%
3	9.02s	66.2%
4	9.1s	74.38%
5	9.14s	79.49%

Table 5.8: First Rewriting Evaluation Results Datasets API

2. $Q = \{(g, m_1, op_1), ...(g, m_n, op_n)\} \Rightarrow Q = (g, \{m_1, op_1\}, ..., \{m_n, op_n\})$

   The results of this evaluation are shown in detail in Tables 5.9 and 5.10

Query ID	Evaluation Time	Evaluation Time Total
1	8.85s	8.85s
2	9.74s	18.59s
3	9.14s	27.73s
4	9.42s	37.15s
5	8.5s	45.65s

Table 5.9: Second Queries Evaluation Results Datasets API

3. $Q = \{(g_1, m, op), ...(g_n, m, op)\} \Rightarrow Q = \{(g_1 \wedge ... \wedge g_n, m, op), (proj_{G_1}, (g_1 \wedge ... \wedge g_n, m, op), op), ..., (proj_{G_N}, (g_1 \wedge ... \wedge g_n, m, op), op)\}$

   The results of this evaluation are shown in detail in Tables 5.11 and 5.12.

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	10.2s	45.13%
3	10.63s	61.67%
4	12.04s	67.59%
5	12.54s	72.53%

Table 5.10: Second Rewriting Evaluation Results Datasets API

Query ID	Evaluation Time	Evaluation Time Total
1	8.43s	8.43s
2	8.5s	16.93s
3	8.33s	25.26s
4	8.43s	33.69s

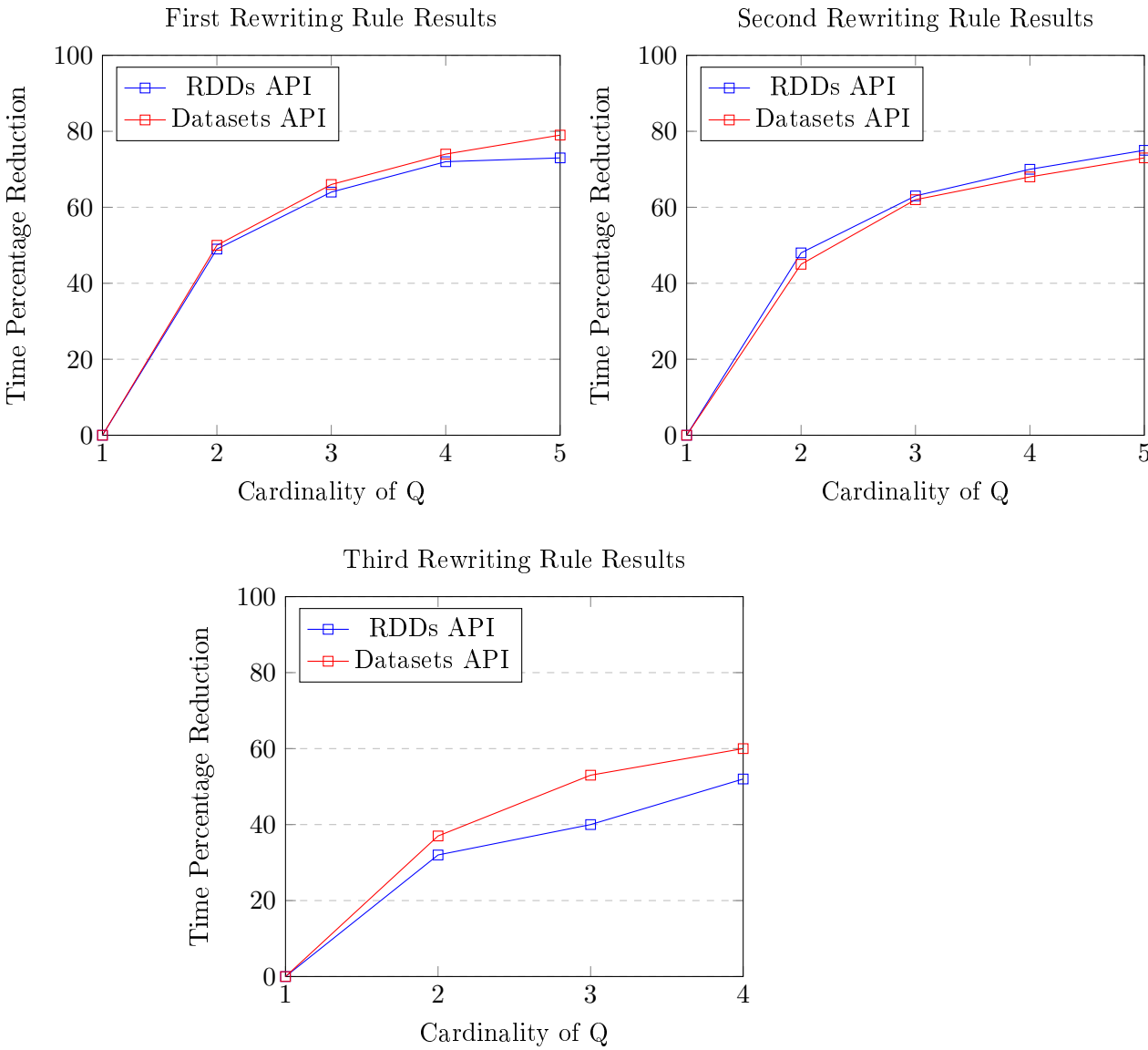
Table 5.11: Third Queries Evaluation Results Datasets API

Q Cardinality	Evaluation Time	Percentage Decrease in Time
2	10.56s	37.6%
3	11.8s	53.3%
4	13.27s	60.6%

Table 5.12: Third Rewriting Evaluation Results Datasets API

Once again, the results of the experiments presented in the previous tables show that the rewriting rules succeed in reducing the evaluation cost of the queries in the Datasets API implementation. The exact same remarks can be made on the results for each rewriting rule. The results have much in common and are visualised more clearly in the next subsection.

5.3 Visualization of the Results of the Rewritings Evaluations



Visualising the performance of each rewriting rule for both evaluations, it is clear that the results of the experiments are satisfactory. This type of visualization makes the observation that the percentage reduction is increasing as the cardinality of the Q is increasing even more obvious. Our experimental evaluation proved the effectiveness of the theory about query rewritings and the fact that it is useful in practice regardless of the format of the data sets and the evaluation mechanisms used to analyze them. However, it is worth mentioning that this effectiveness may fluctuate and is influenced by a number of factors including the number of

available processors and nodes in the cluster, the size of the data set, the number of attributes and their distinct values etc.

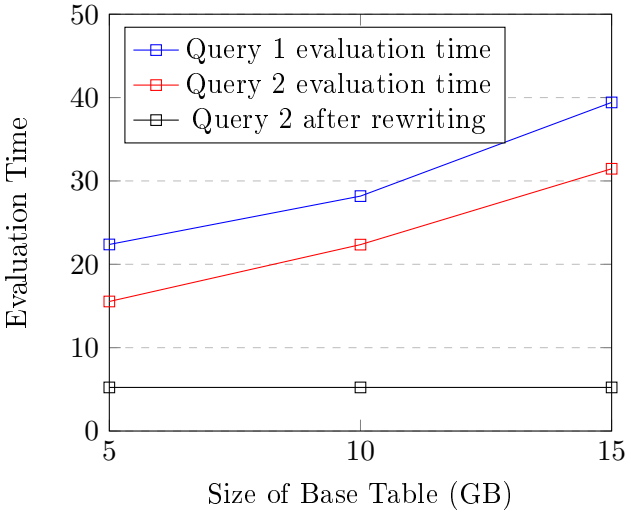
5.4 Basic Rewriting Rule

The last rule that needs to be evaluated is the basic rewriting rule:

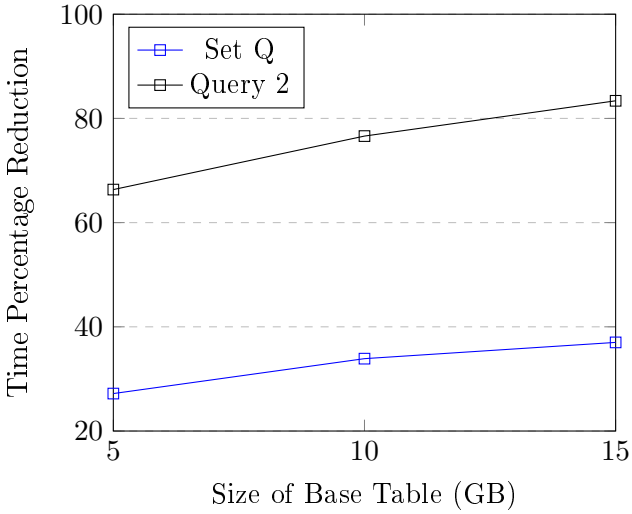
$$\mathbf{(g \circ f,\ m,\ op) = (g,\ (f,\ m,\ op),\ op)}$$

For these series of experiments, a synthetic structured data set was used. The analysis context of this data set is depicted in Figure 5.3 and contains two relational tables, the base table D and one extra table Product. The attributes q and p in the base table and the attribute c in the *Product* table follow the uniform distribution. To evaluate this rewriting rule, two HIFUN queries are created. Both of them have the same measuring attribute (the *quantity* of the delivery invoice) and the same distributive operation applicable on the measuring attribute. The attribute p was selected as the grouping attribute of the first query and the attribute $c \circ p$ was selected as the grouping attribute of the second query. Two kind of experiments were conducted: Firstly, we kept the size of the top-level table *Product* fixed (1GB size and 20M tuples) and gradually increased the size of the base table D (increased the number of tuples of D while keeping the distinct values of products fixed according to the number of products in the *Product* table). Secondly, we kept the size of the base table D fixed (15GB and 255M tuples) and gradually increased the size of the top-level table *Product*(as we increased the number of products, we increased the distinct values of the p attribute in the base table without changing the number of tuples in it). The queries were firstly evaluated individually and then according to the rewriting rule. The results of this evaluation are shown in detail in the following plots. The Set Q now contains the two queries as described above.

Basic Rewriting Rule - First Series of Experiments



Rewriting Results



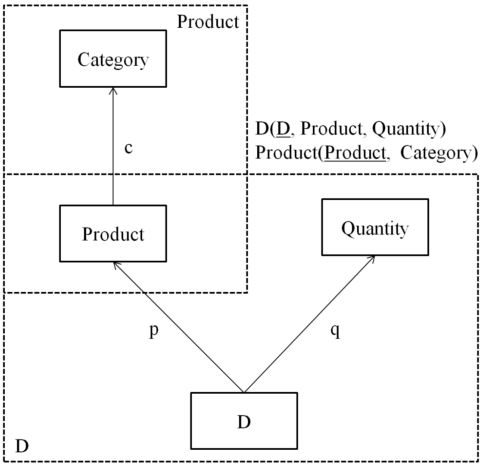
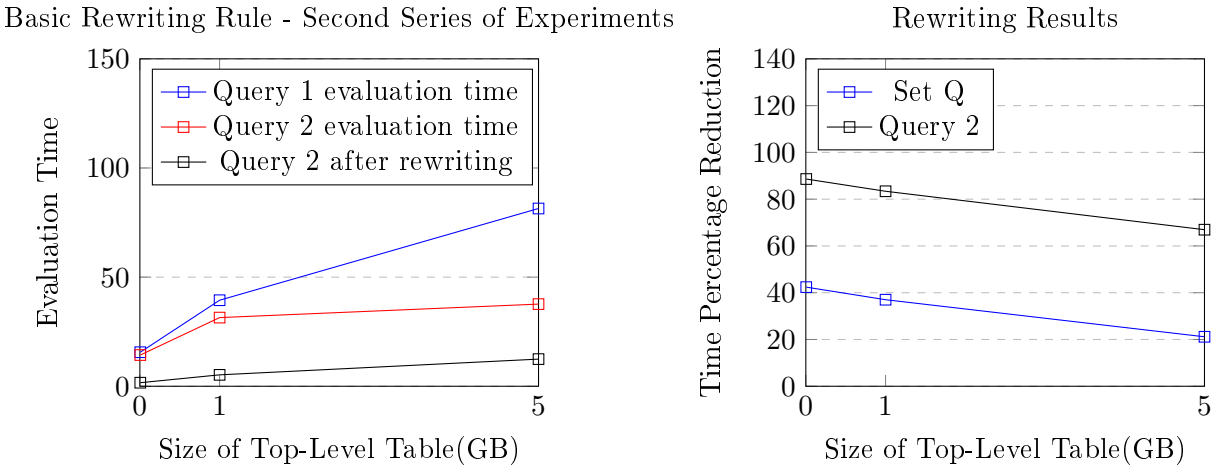


Figure 5.3: Analysis Context of the data set of the Basic Rewriting Rule evaluation



Taking a closer look at the results of the first series of experiments, we notice that as the size of the base table increases, the effectiveness of the basic rewriting rule increases. The bigger the tables that have to be joined, the higher the cost of the join. This increases the evaluation time of the second query when we don't use the basic rewriting rule. By avoiding the join of the two big tables (for the evaluation of the second rewritten query, the top-level table is joined with the answer table of the first query which is much smaller than the base table), the evaluation cost of the rewritten query stays fixed even when we increase the base table's size as the number of tuples in the answer table of the first query is fixed and equal to the number of the distinct values of the products. These explain the increment in the percentage reduction of both the set Q and the Query 2.

As far as the results of the second series of experiments are concerned, we notice that as the size of the top-level table increases, the effectiveness of the basic rewriting rule decreases. Again, the bigger the size of the top-level table, the bigger the cost of the join. When using the basic rewriting rule, the size of the answer table is increased when we add more tuples on the top-level table as the distinct values of the products are increased. As we can see from the plots, the evaluation time of the rewritten query increases for two reasons: 1: the join cost as described above, 2: applying an operation on a bigger table takes more time. Moreover, the results of the experiments show that the more the distinct values of the grouping attribute, the higher the cost of the evaluation of a query. These explain the reduction in the percentage reduction in time of both the set Q and the Query 2 as the size of the top-level table increases.

To sum up, the effectiveness of the basic rewriting rule fluctuates according to a couple of factors like the size of the tables and the number of the distinct values of the common attribute. In any case, the basic rewriting rule once again succeeds in reducing the evaluation time of the queries.

Before ending the evaluation section, we would like to make a final remark. The fact that we managed to achieve computational cost reductions using the Apache Spark framework gives hope for further optimizations. Technology is constantly evolving and new big data analytics tools will appear in the future. Mapping analytic queries to new lower level mechanisms may help in both reducing the evaluation cost of a single query and increasing the percentage reduction of the evaluation cost of a set of queries through rewritings.

Chapter 6

Future Work and Conclusions

In this section, we describe a number of research items that can be used for the extension of our system.

The first research item concerns change in the data set D. Indeed, throughout this work we have tacitly assumed that the data set is "static". While this might be a reasonable assumption in several application environments, it is by no means true for big data sets in general. Indeed, there are application environments where the data set changes frequently and where the results to some important, continuous queries have to be updated frequently as well. The objective here is to study incremental algorithms that take as input the increment in data and produce the increment in the query result. Our system can then be extended by implementing those algorithms [1].

The second research item concerns the extension of rewriting based on restrictions. Think of the staff working at an airport. Each day a big amount of flights take place there and a member of the staff would like to ask questions like "How many flights per hour took place between time X and time Y on day Z?". In this case, if the result of the query "How many flights per hour took place during the whole day Z?" was pre-evaluated and stored as a materialized view, then this query could be evaluated faster using this view. In general, our system can be extended to compute such queries with restrictions using the answers of pre-evaluated queries.

The third research item concerns the query answer visualization. As the grouping attributes of a query are increased, meaning that the domain of the answer of the query gets more complex, the understanding and the visual representation of the answer gets more difficult. However, there are tools available that can be exploited to make the visual exploration of query answers even more simple. Our system could be extended with additional visualization tools to become user-friendlier.

The fourth research item concerns the execution plans generation complexity. This work presents an algorithm which takes a query set as an input, applies query rewritings and generates query execution plans but makes no remarks about its complexity.

The last research item is related to the *Self-Service Analytics* field. Briefly, this field aims to enable business agents to perform analytic queries on business information on their own. This field can be supported by tools which allow a professional to perform an analysis using a data model which has been simplified for understanding purposes. The HIFUN data model is simple enough to allow anyone interested in business information analytics to express analytic queries and can be leveraged by this field.

To conclude, in this thesis we showed that the HIFUN language is indeed of great benefit in practice. We leveraged the theory of this language and the mechanisms of the Apache Spark framework to create a system which allows a user to discover useful patterns hidden in data sets. The theory around *Query Rewriting* and *Query Execution Plans* proved to be of great importance to reducing evaluation costs of queries and they helped in the design of an algorithm for execution plans generation. The rewriting rules proposed are shown to exhibit excellent performance gains avoiding a lot of unnecessary computations. Our system provides an all-in-one solution for big data analytics and can be extended to support a big variety of data set formats, with its optimization mechanisms working regardless of the nature of the data.

Bibliography

[1] N. Spyratos and T. Sugibuchi, "A high level query language for big data analytics," 2014. [Online]. Available: https://www.lri.fr/~bibli/Rapports-internes/2014/RR1575.pdf

[2] ——, "Hifun - a high level functional query language for big data analytics," 2017. [Online]. Available: http://www.ics.forth.gr/tech-reports/2017/2017.TR467_HiFu_Query_Language_Big_Data_Analytics.pdf

[3] A. E. Nada Elgendy, "Big data analytics: A literature review paper," 2014. [Online]. Available: https://www.researchgate.net/publication/264555968_Big_Data_Analytics_A_Literature_Review_Paper

[4] S. Sonka, "Big data characteristics," 2016. [Online]. Available: http://www.ifama.org/resources/Documents/v19ia/01_Editor.pdf

[5] "Big data." [Online]. Available: https://en.wikipedia.org/wiki/Big_data

[6] R. V. Zicari, "Big data: Challenges and opportunities." [Online]. Available: http://www.odbms.org/wp-content/uploads/2013/07/Big-Data.Zicari.pdf

[7] A. J. Ishwarappa, "A brief introduction on big data 5vs characteristics and hadoop technology." [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915006973

[8] M. C. Stefan Niemeier, Andrea Zocchi, "Reshaping retail: Why technology is transforming the industry and how to win in the new consumer driven world." [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118656660.html

[9] V. Beal, "Big data analytics." [Online]. Available: http://www.webopedia.com/TERM/B/big_data_analytics.html

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters." [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[11] R. Lammel, "Google's mapreduce programming model — revisited." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.5859&rep=rep1&type=pdf

[12] T. Y. Patrick Garrity, "Writing, submitting, and managing map-reduce jobs," 2010. [Online]. Available: http://webmapreduce.sourceforge.net/docs/User_Guide/sect-User_Guide-Introduction-What_is_Map_Reduce.html

[13] A. S. Donald Miner, "Mapreduce design patterns: Building effective algorithms and analytics for hadoop and other systems." [Online]. Available: https://www.amazon.com/MapReduce-Design-Patterns-Effective-Algorithms/dp/1449327176/httpwwwtuto0a-20

[14] A. W. Services, "Amazon elastic mapreduce (amazon emr) developer guide." [Online]. Available: https://www.amazon.com/Amazon-Elastic-MapReduce-Developer-Guide-ebook/dp/B007US6CIO/httpwwwtuto0a-20

[15] E. Miller, "An overview of mapreduce and its impact on distributed data processing." [Online]. Available: https://www.amazon.com/Overview-MapReduce-Impact-Distributed-Processing-ebook/dp/B00CMCGBQQ//httpwwwtuto0a-20

[16] P. D. P. G. Harshawardhan S. Bhosale, "A review paper on big data and hadoop," 2014. [Online]. Available: http://www.ijsrp.org/research-paper-1014/ijsrp-p34125.pdf

[17] "Apache hadoop: The big data refinery," 2012. [Online]. Available: http://hortonworks.com/wp-content/uploads/2012/06/Apache-Hadoop-Big-Data-Refinery-WP.pdf

[18] "Apache hadoop." [Online]. Available: https://en.wikipedia.org/wiki/Apache_Hadoop

[19] D. Borthakur, "Hdfs architecture guide." [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf

[20] K. Chinnakali, "What are the 3 s's of spark and its effect on big data?" 2015. [Online]. Available: http://bigdata-madesimple.com/what-is-3-ss-of-spark-and-its-effect-on-big-data/

[21] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," 2016. [Online]. Available: http://paperity.org/p/78205157/big-data-analytics-on-apache-spark

[22] "Apache spark docs cluster mode overview." [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html

[23] C. Strachey, "Fundamental concepts in programming languages. higher-order and symbolic computation," 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1010000313106

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." [Online]. Available: http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[26] "Apache spark java rdd api." [Online]. Available: https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/api/java/JavaRDD.html

[27] "Kryo serializer." [Online]. Available: https://code.google.com/archive/p/kryo/wikis/V1Benchmarks.wiki

[28] "Apache spark java pair rdd api." [Online]. Available: https://spark.apache.org/docs/2.0.1/api/java/org/apache/spark/api/java/JavaPairRDD.html

[29] "Apache spark docs spark sql." [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html