ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0017

EDITE - ED 130

Doctorat ParisTech
T H È S E
pour obtenir le grade de docteur délivré par

TELECOM ParisTech
Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par
Yongchao TIAN
le 7 Avril 2017

Accelerating Data Preparation for Big Data Analytics

Directeur de thèse: **Marko VUKOLIĆ**
Co-encadrement de la thèse: **Pietro MICHIARDI**

Jury

Mme. Elena BARALIS, Professor, Politecnico di Torino	Rapporteuse
M. Guillaume URVOY-KELLER, Professor, Université Nice Sophia Antipolis	Rapporteur
M. Refik MOLVA, Professor, Eurecom	Examinateur
M. Paolo PAPOTTI, Assistant Professor, Arizona State University	Examinateur

THÈSE

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Abstract

We are living in a big data world, where data is being generated in high volume, high velocity and high variety. Big data brings enormous values and benefits, so that data analytics has become a critically important driver of business success across all sectors. However, if the data is not analyzed fast enough, the benefits of big data will be limited or even lost.

Despite the existence of many modern large-scale data analysis systems, data preparation which is the most time-consuming process in data analytics has not received sufficient attention yet. In this thesis, we study the problem of how to accelerate data preparation for big data analytics. In particular, we focus on two major data preparation steps, data loading and data cleaning.

As the first contribution of this thesis, we design DiNoDB, a SQL-on-Hadoop system which achieves interactive-speed query execution without requiring data loading. Modern applications involve heavy batch processing jobs over large volume of data and at the same time require efficient ad-hoc interactive analytics on temporary data generated in batch processing jobs. Existing solutions largely ignore the synergy between these two aspects, requiring to load the entire temporary dataset to achieve interactive queries. In contrast, DiNoDB avoids the expensive data loading and transformation phase. The key innovation of DiNoDB is to piggyback on the batch processing phase the creation of metadata, that DiNoDB exploits to expedite the interactive queries.

The second contribution is a distributed stream data cleaning system, called Bleach. Existing scalable data cleaning approaches rely on batch processing to improve data quality, which are very time-consuming in nature. We target at stream data cleaning in which data is cleaned incrementally in real-time. Bleach is the first qualitative stream data cleaning system, which achieves both real-time violation detection and data repair on a dirty data stream. It relies on efficient, compact and distributed data structures to maintain the necessary state to clean data, and also supports rule dynamics.

We demonstrate that the two resulting systems, DiNoDB and Bleach, both of which achieve excellent performance compared to state-of-the-art approaches in our experi-

mental evaluations, and can help data scientists significantly reduce their time spent on data preparation.

Résumé

Nous vivons dans un monde de big data, où les données sont générées en grand volume, grande vitesse et grande variété. Le big data apportent des valeurs et des avantages énormes, de sorte que l'analyse des données est devenue un facteur essentiel de succès commercial dans tous les secteurs. Cependant, si les données ne sont pas analysées assez rapidement, les bénéfices de big data seront limités ou même perdus.

Malgré l'existence de nombreux systèmes modernes d'analyse de données à grande échelle, la préparation des données est le processus le plus long de l'analyse des données, n'a pas encore reçu suffisamment d'attention. Dans cette thèse, nous étudions le problème de la façon d'accélérer la préparation des données pour le big data d'analyse. En particulier, nous nous concentrons sur deux grandes étapes de préparation des données, le chargement des données et le nettoyage des données.

Comme première contribution de cette thèse, nous concevons DiNoDB, un système SQL-on-Hadoop qui réalise l'exécution de requêtes à vitesse interactive sans nécessiter de chargement de données. Les applications modernes impliquent de lourds travaux de traitement par lots sur un grand volume de données et nécessitent en même temps des analyses interactives ad hoc efficaces sur les données temporaires générées dans les travaux de traitement par lots. Les solutions existantes ignorent largement la synergie entre ces deux aspects, nécessitant de charger l'ensemble des données temporaires pour obtenir des requêtes interactives. En revanche, DiNoDB évite la phase coûteuse de chargement et de transformation des données. L'innovation importante de DiNoDB est d'intégrer à la phase de traitement par lots la création de métadonnées que DiNoDB exploite pour accélérer les requêtes interactives.

La deuxième contribution est un système de flux distribué de nettoyage de données, appelé Bleach. Les approches de nettoyage de données évolutives existantes s'appuient sur le traitement par lots pour améliorer la qualité des données, qui demandent beaucoup de temps. Nous ciblons le nettoyage des données de flux dans lequel les données sont nettoyées progressivement en temps réel. Bleach est le premier système de nettoyage qualitatif de données de flux, qui réalise à la fois la détection des violations en temps réel et la réparation des données sur un flux de données sale. Il s'appuie sur des structures

de données efficaces, compactes et distribuées pour maintenir l'état nécessaire pour nettoyer les données et prend également en charge la dynamique des règles.

Nous démontrons que les deux systèmes résultants, DiNoDB et Bleach, ont tous deux une excellente performance par rapport aux approches les plus avancées dans nos évaluations expérimentales, et peuvent aider les chercheurs à réduire considérablement leur temps consacré à la préparation des données.

Acknowledgments

Foremost, I would like to express my gratitude to my advisors, Marko Vukolić and Pietro Michiardi, for the invaluable guidance, support and patience in the last three and a half years. Since I did not have any research experience in the beginning, they taught me how to conduct research step by step from the first day of my PhD. When things did not go well with the plan, they encouraged me to insist and helped me to overcome the difficulties. Overall, I learned a lot from them, not only in research but also in daily life. It is a great honor to be their first joint PhD student.

I am also very grateful to other members of my thesis committee: Elena Baralis, Guillaume Urvoy-Keller, Refik Molva and Paolo Papotti. They helped me to improve my manuscript by providing precious comments and suggestions. The discussion with them helped me to have a better understanding of my research work.

I would like to thank my friends and colleagues in Eurecom. Because of their companionship, my life in Eurecom was full of joy. Thank you Shengyun Liu, Paolo Viotti, Fabio Pulvirenti, Xiwen Jiang, Jingjing Zhang, Daniele Venzano, Qianrui Li, Haifan Yin, Xiaohu Wu, Junting Chen, Han Qiu, Alberto Benegiamo, Duc-Trung Nguyen, Francesco Pace, Romain Favraud and Mohammad-Irfan Khan.

Finally, I would like to express my deep gratitude to my parents, Jun Tian and Zhihua Yin, and my fiancée, Shishi Liu, who always supported me unconditionally. I love you with all my heart.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

We live in the era of data deluge, where vast amounts of data are being generated. The data brings enormous values and benefits, not only in scientific domains like astronomy or biology, but also in domains which are closely relevant to our daily lives, such as E-commerce and transportation. Companies can use their collected data to support human decisions, discover customer needs and build new business models. With the technological improvement over the last decades, massive datasets can be stored at low costs. Therefore, more and more companies start to store as much data as they could collect. However, converting the data into valuable insights is still a challenging task.

Relational Database Management Systems (Relational DBMSs) were believed to be the one-size-fits-all tools for data analytics in the last century [1]. In their relational model, all data is represented in terms of tuples, grouped into a set of tables which are related to each other. Each table has a predefined schema which all the tuples in the table must follow. Over the years, Relational DBMSs successfully supported a large number of data-centric applications with quite different features and requirements.

However, as we enter the 21st century, the traditional DBMS becomes a poor fit in many application scenarios. One of the main reasons why the traditional DBMS is out of date is the exponential increase of the data. Google, as a data-driven company, processes more than 3.5 billion requests per day and stores more than 10 exabytes of data. Facebook collects 600 terabytes of data per day, including 4.3 billion pieces of content, 5.75 billion "like" and 350 million photos. Some estimates suggest that overall at least 2.5 quintillion bytes of data is produced every day and 40 zettabytes of data will be in existence by 2020 [3]. Such huge amounts of data are far beyond the capacity of any traditional DBMS. New efficient and powerful data analytics tools are required to cope with the big data challenge.

This need has drawn considerable attention from both academia and industry. Hence, in recent years, modern large-scale data analysis systems have flourished, which bring enormous innovative techniques and optimizations. These systems aim at speeding up the data analysis procedures for large datasets. However, as reported by many data scientists [29], only 20% of their time is spent doing the desired data analysis tasks. Data scientists need to spend 80% of time, sometimes even more, on data preparation which is a slow, difficult and tedious process. Nevertheless, data preparation, the essential step before performing data analysis, has not received sufficient attention despite its importance.

1.2 Data Preparation

Data preparation, which is also called data preprocessing, focuses on determining what the data is, improving data quality, standardizing how data is defined and structured, collecting and consolidating data, and transform data to make it useful, particularly for analysis [30]. In a nutshell, data preparation increases the value of data analysis. It includes the steps of accessing, searching, aggregating, enriching, transforming, cleaning and loading data.

Without proper data preparation, data analytics may generate misleading results if underlying datasets are dirty. One analysis project may fail due to issues with security and privacy, if its dataset is carelessly prepared without hiding sensitive information. Modern data analysis systems all require data preparation as the preliminary step, as they are not capable to retrieve insights from raw data unless data is in proper formats or loaded in the systems. Therefore, data preparation is particularly crucial for the success of data analytics.

In many organizations, data preparation requires manual efforts using processes which are difficult to share or even repeat. More and more data scientists spend too much time on data preparation and are unable to have enough time for solving other challenging data problems. Hence, the biggest problem with data preparation is that it is very time-consuming and costly.

As we are living in a big data world, with the increasing volume and variety of data in recent years, data preparation has grown more demanding and become more time-consuming. In the meanwhile, data also keeps being generated at a higher velocity. Business needs start to demand shorter and shorter intervals between the time when data is collected and the time when the results of analyzing the data are available for manual or algorithmic decision making. Data scientists desire the ability to analyze datasets as soon as possible, e.g, a few seconds after datasets have been collected. When the raw data is generated continuously from a data stream, data scientists may even

want to perform their data analysis incrementally in real-time, rather than wait for all the data to be acquired. Being able to make timely decisions has become increasingly crucial.

Clearly, the slow but indispensable data preparation process becomes the obstacle to achieve timely decision making. To overcome this obstacle, in this thesis, we study the problem of how to accelerate data preparation for big data analytics. In particular, we focus on two kinds of costly data preparation operations, data loading and data cleaning.

1.2.1 Data Loading

Data loading is the process of copying and loading data from a data source to a data warehouse or any other target storage system. It was already a popular concept in the 1970s, as the last step of the well-known ETL (Extract, Transform, Load) process in database usage. Data loading may also include applying the verification of constraints defined in the database schema, such as uniqueness, referential integrity and mandatory fields.

Nowadays, data loading not only is an operation existing in traditional databases but also becomes a primary step in many modern data analysis systems. To load the raw data, these systems often convert the data into some specific data formats, such as a column-based data format, or fully load the data into memory. Apart from changing the data layout, data loading process may also generate additional cumulative informations, such as data statistics and indexes, which can be used to optimize subsequent query executions. Although some systems claim to have the ability to process data in situ without loading, their performance is unsatisfactory. This is because without any useful metadata or optimizing the data layout, these systems can only process the raw data in a brute-force manner, e.g., repeatedly scanning entire files for every query.

As now data scientists want to analyze datasets as soon as possible, the slow data loading becomes a bottleneck in data analytics. In particular, when we are analyzing temporary datasets, which will be simply dropped after executing a few queries, data loading is not suitable. Clearly, we need an approach which avoids the slow data loading but is capable to achieve efficient query execution.

1.2.2 Data Cleaning

As we step into a data-driven world, enforcing and maintaining the data quality become critical tasks. According to an industry survey [31], more than 25% of the critical data in the world's top companies is flawed. Without proper data cleaning, issues with data

quality can lead to misleading analysis outcomes on the "garbage in, garbage out" basis. For example, InsightSquared [32] predicts that dirty data across businesses and the government costs the U.S. economy 3.1 trillion dollars a year.

Data cleaning, also called data cleansing or data scrubbing, is the process of detecting and correcting, or removing, corrupt or inaccurate data records from a dataset. Recently, two major trends in data cleaning have emerged. The first is a quantitative approach, called quantitative data cleaning, which is largely used for outlier detection by employing statistical methods to identify abnormal behaviors and errors. The second is a logical approach, called qualitative data cleaning. Qualitative data cleaning, on the other hand, relies on specifying patterns or rules of a legal data instance. It consists of two phases, violation detection which is to identify the data which violate the defined patterns or rules as errors, and violation repair which is to find a minimal set of changes that fix the detected errors.

Data cleaning is considered as the most time-consuming task in data preparation, mainly because it often involves costly computations, such as enumerating pairs of tuples and handling inequality joins, which are difficult to scale to big datasets. Most of existing data cleaning solutions have focused on batch data cleaning, by processing static data stored in data warehouse, which discourage data scientists from having their timely data analysis. It has become an urgent task to develop new innovative data cleaning solutions which are fast and also effective.

1.3 Contributions

The goal of this thesis is to develop advanced systems to accelerate the time-consuming data preparation process for big data analytics. In particular, we focus on data loading and data cleaning. As contributions, we design and implement two systems, DiNoDB, an interactive-speed query engine for ad-hoc queries on temporary data which avoids data loading by seamlessly integrating with batch processing systems, and Bleach, a novel stream data cleaning system, that aims at efficient and accurate data cleaning under real-time constraints.

1.3.1 DiNoDB: an Interactive-speed Query Engine for Temporary Data

The first contribution consists in the design of DiNoDB, a SQL-on-Hadoop system which achieves interactive-speed query execution without requiring data loading. Modern applications involve heavy batch processing jobs over large volumes of data and at the

same time require efficient ad-hoc interactive analytics on temporary data. Existing solutions, however, typically focus on one of these two aspects, largely ignoring the need for synergy between the two. Consequently, interactive queries require to load the entire dataset that may provide meaningful return on investment only when data is queried over a long period of time.

In contrast, DiNoDB avoids the expensive loading and transformation phase that characterizes both traditional DBMSs and current interactive analytics solutions. It is particularly tailored to modern workflows found in use cases such as machine learning and data exploration, which often involve iterations of cycles of batch and interactive analytics on data that is typically useful for a narrow processing window. The key innovation of DiNoDB is to piggyback on the batch processing phase the creation of metadata that DiNoDB exploits to expedite the interactive queries.

Our detailed experimental analysis both on synthetic and real-life datasets, demonstrates that DiNoDB significantly reduces time-to-insight and achieves very good performance compared to state-of-the-art distributed query engines, such as Hive, Stado, Spark SQL and Impala.

1.3.2 Bleach: a Distributed Stream Data Cleaning System

The second contribution is a distributed stream data cleaning system, called Bleach. Existing scalable data cleaning approaches have focused on batch data cleaning, which are quite time-consuming. As most data sources now come in streaming fashion, such as log files generated in web servers and online purchases, we target at performing data cleaning directly on data streams. Despite the increasing popularity of stream processing systems, no qualitative stream data cleaning techniques have been proposed so far. In this thesis, we bridge this gap by addressing the problem of rule-based stream data cleaning, which sets stringent requirements on latency, rule dynamics and ability to cope with the continuous nature of data streams.

We design Bleach, a stream data cleaning system which achieves real-time violation detection and data repair on a dirty data stream. Bleach relies on efficient, compact and distributed data structures to maintain the necessary state to repair data. Additionally, it supports rule dynamics and uses a “cumulative” sliding window operation to improve cleaning accuracy.

We evaluate Bleach using both synthetic and real data streams and experimentally validate its high throughput, low latency and high cleaning accuracy, which are preserved even with rule dynamics. In the absence of an existing comparable stream-cleaning baseline, we compared Bleach to a baseline system built on the micro-batch paradigm, and experimentally show the superior performance of Bleach.

1.4 Thesis Outline

The thesis is organized as follows. We give the background knowledge of the thesis in Chapter 2, in which we introduce modern large-scale data analysis systems and streaming data processing. Chapter 3 presents our first contribution, DiNoDB, with its detailed design and experimental analysis. In Chapter 4, we describe our stream data cleaning systems, Bleach, introduce the related challenges and also give the experimental evaluation. Finally, in Chapter 5 we conclude the thesis and discuss some future work.

Chapter 2

Background and Preliminaries

In this chapter, we present the necessary background material of the thesis. First, we introduce a few representative modern large-scale analysis systems which will help us introduce and discuss requirements in terms of data preparation. Then, we cover data streams, and give the basis of two popular stream processing frameworks, Apache Storm [70], on top of which we build our stream data cleaning system, Bleach, and Spark Streaming [22].

2.1 Large-scale Data Analysis Systems

Over the last decade, numerous large-scale data analysis systems have emerged to address the big data challenge. Unlike the traditional DBMS which runs in a single machine, these systems run in a cluster with a collection of machines (nodes) in a Shared Nothing Architecture [4] where all machines are connected with a high-speed network and each has its own local disk and local main memory [5], as shown in Figure 2.1. To achieve parallel processing, these systems divide datasets to be analysed into partitions which are distributed on different machines.

Structured Query Language (SQL) is the declarative language most widely used for analyzing data in large-scale data analysis systems. Users can specify an analysis task using an SQL query, and the system will optimize and execute the query. To clarify, we focus on systems for large-scale analysis, namely, the field that is called Online Analytical Processing (OLAP) in which the workloads are read-only, as opposed to Online Transaction Processing (OLTP).

We classify these systems into two categories, Parallel DBMSs and SQL-on-Hadoop Systems, based on their storage layers: Parallel DBMSs store their data inside their database

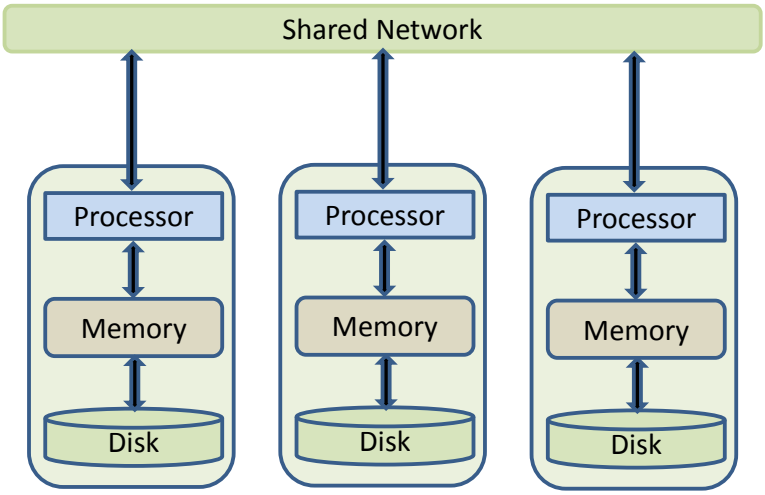


Figure 2.1 – Shared-nothing architecture

instances, while in SQL-on-Hadoop Systems, data is kept in the Hadoop distributed file system.

2.1.1 Parallel DBMSs

Parallel DBMSs are the earliest systems to make parallel data processing available to a wide range of users, in which each node in the cluster is a database instance. Many of these systems are inspired by the research work Gamma [7] and Teradata [8]. They achieve high performance and scalability by distributing the rows of a relational table across the nodes of the cluster. Such a horizontal partitioning scheme enables SQL operators like selection, aggregation, join and projection to be executed in parallel over different partitions of tables located in different nodes. Many commercial implementations are available, including Greenplum [9], Netezza [10], Aster nCluster [12] and DB2 Parallel Edition [27], as well as some open source projects such as MySQL Cluster [11], Postgres-XC [13] and Stado [14].

Some other systems like Amazon RedShift [15], ParAccel [16], Sybase IQ [25] and Vertica [26], vertically partition tables by collocating entire columns together instead of collocating rows with a horizontal partitioning scheme. When executing user queries, such systems can more precisely access the data they need rather than scanning and discarding unwanted data in rows. These column-oriented systems have been shown to use CPU, memory and I/O resources more efficiently compared to row-oriented systems in large-scale data analytics.

For parallel DBMSs, data preparation is always a mandatory and time-consuming step. Data cleaning must be performed in advance to guarantee the quality of data. As parallel DBMSs are built on traditional DBMSs, they all require data to be loaded before executing

any queries. Each datum in the tuples must be parsed and verified so that data conform to a well-defined schema. For large amounts of data, this loading procedure may take a few hours, even days, to finish, even with parallel loading across multiple machines. Moreover, to better benefit from a number of technologies developed over the past 30 years in DBMSs, parallel DBMSs provide optimizations like indexing and compression, which also necessitate the phase of data preprocessing.

2.1.2 SQL-on-Hadoop Systems

A milestore in the big data research is the MapReduce framework [45], which is the most popular framework for processing vast datasets in clusters of machines, mainly because of its simplicity. The open-source Apache Hadoop implementation of MapReduce has contributed to its widespread usage both in industry and academia. Hadoop consists of two main parts: the Hadoop distributed file system (HDFS) and MapReduce for distributed processing. Instead of loading data into the DBMSs, Hadoop users can process data in any arbitray format in situ as long as data is stored in the Hadoop distributed file system.

Hadoop Basics: The MapReduce programming model [34] consists of two functions: $map(k_1, v_1)$ and $reduce(k_2, list(v_2))$. Users implement their processing logic by specifying customized map and reduce functions. The $map(k_1, v_1)$ function accepts input records in the form of *key-value* pair and outputs intermediate one or more key-value pairs of the form $[k_2, v_2]$. The $reduce(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ from the map output, and outputs zero or more key-value pairs of the form $[k_3, v_3]$ as the final results. The transferring data from mappers to reducers is called the *shuffle* phase, in which users can also specify a $partition(k_2)$ function for controlling how the map output key-value pairs are partitioned among the reduce tasks. HDFS is designed be resilient to hardware failures and focuses on high throughput of data access. A HDFS cluster employs a master-slave architecture consisting of a *NameNode* (the master) and multiple *DataNodes* (the slaves). The NameNode manages the file system namespace and regulates client access to files, while the DataNodes serve read and write requests from the clients. In HDFS, a file is split into one or more blocks which are replicated to achieve fault tolerance and stored in a set of DataNodes.

SQL query processing for data analytics over Hadoop has recently gained significant traction, as many enterprise data management tools rely on SQL, and many users prefer writing high level SQL scripts rather than writing complex MapReduce programs. As a consequence, the number of SQL-on-Hadoop systems has increased significantly, which all use HDFS as the underlying storage layer. Next, we present several popular SQL-on-Hadoop systems which are widely used by companies.

Hive [63] is the first native Hadoop system built on top of Hadoop to process SQL-like statements. Queries submitted to Hive are parsed, compiled and optimized to produce a query execution plan. The plan is a Directed Acyclic Graph (DAG) of MapReduce tasks which is either executed through the MapReduce framework or through the Tez framework [17]. Similar to Hadoop, Hive lacks efficient indexing mechanism. Hence, Hive consumes data by performing sequential scan. Hive also supports columnar data organization, typically in the ORC file format, which helps to improve the performance. But transforming data layout as data preparation brings additional cost.

HadoopDB [40] is a hybrid of the parallel DBMS and Hadoop approaches to data analysis, aiming to exploit the best features from both parallel DBMSs and Hadoop. The basic idea is to install a database system on each Hadoop datanode and use Hadoop MapReduce to coordinate execution of these independent database systems. To harvest all the benfits of query optimization provided by the local DBMSs, HadoopDB pushes as much as possible of the query processing work into the local DBMSs. Nevertheless, before performing any query processing, HadoopDB needs to load data from HDFS to its local DBMSs. This is achieved by one of its components, the data loader, which also globally repartitions data based on a specified partiton key.

Impala [66] is an open-source SQL engine architected from the ground up for the Hadoop data processing environment. As MapReduce focuses more on batch processing rather than interactive use by users, Hadoop jobs suffer the overhead incurred from task scheduling. Therefore, to reduce latency, Impala avoids utilizing MapReduce and implements a distributed architecture based on daemon processes that are responsible for all aspects of query execution. These daemon processes run on the same machines as the HDFS infrastructure. Impala also accepts input data in columnar data organization, typically in the Parquet file format [86], that the user needs to create a Parquet table and to load data into the Parquet table. Before executing any queries with Impala, the user should run the COMPUTE `STATS <table>` statement, which instructs Impala to gather statistics on the table. Those statistics will be used later for query optimization.

The MapReduce programming model is too restrict to support some applications that reuse a working set of data across multiple parallel operations, such as iterative machine learning jobs. Therefore, a new framework, called Apache Spark [69], is designed for these applications, which also provides similar scalability and fault tolerance properties as MapReduce. Similar to Hadoop, Spark also runs on top of HDFS infrastructure. Spark provides a more flexible dataflow-based execution model that can express a wide range of data access and communication patterns, rather than only specifying map and reduce functions as in MapReduce. The main abstraction in Spark is that of a resilient distributed dataset (RDD), which is a read-only data structure partitioned across a set of machines. RDDs support a rich set of operators, such as *map*, *filter* and *groupByKey*, and

enable efficient data reuse in a broad range of applications by allowing users to persist intermediate results in memory. RDDs achieve fault tolerance through a notion of lineage: if a RDD partition is lost, Spark can rebuid this RDD partition from the information kept in that RDD about how it was derived from other RDDs.

Spark SQL [71] is the SQL processing module in Spark, which provides a DataFrame API which can perform relational operations on both external data sources and Spark's built-in distributed collections. It introduces a new extensible optimizer, called Catalyst, which is able to add new data sources, optimization rules and user-defined data types. Similar to Impala, Spark SQL also supports Parquet file format. To improve the performance, Spark SQL can also cache entire tables in memory to avoid disk I/O bottleneck, which is similar to the loading procedure in parallel DBMSs.

2.2 Streaming Data Processing

Streaming data is a sequence of data tuples that is unbounded in size and generated continuously from potentially thousands of data sources. Streaming data includes a wide variety of data such as log files generated in web servers, online purchases, player activities in online games, information from social networks and telemetry from sensors, etc [18]. Due to the unbounded nature, streaming data can not be processed after all data is collected as in the batch processing. Streaming data processing needs to be sequential and incremental as the event occurs in real-time or near real-time.

Streaming data processing represents challenges in terms of performance, scalability, robustness and fault-tolerance. Traditionally, custom coding has been used for streaming data processing [126]. But this approach suffers from its inflexibility, high cost of development and maintenance, and slow response time to new feature requests. In recent year, many modern distributed stream processing frameworks have emerged. Upon these frameworks, users can easily build their own stream processing applications. These frameworks fall roughly into two categories. The first category, called real-time streaming data processing systems, includes Apache S4 [19], Apache Storm [70], Apache Samza [20] and Apache Flink [21]. Such systems process the streaming data on a tuple-by-tuple basis in which every tuple is processed as it arrives. In contrast, the systems from the second category, such as Spark Streaming [22], collect data in certain time intervals and process them in batches. These systems are called micro-batch streaming data processing systems. Micro-batch systems tend to have higher throughput than real-time systems, especially when using large batches. However, large batches in micro-batch systems incur high processing latencies, which prohibit real-time processing in streaming data.

Next, we particularly introduce two frameworks, Apache Storm, on top of which our real-time stream data cleaning system, Bleach, is built, and Spark Streaming, on top of which we build a baseline system to evaluate Bleach.

2.2.1 Apache Storm

Apache Storm aims at providing a framework for real-time stream processing, which also achieves scalability and fault-tolerance. Similar to Hadoop, Storm relies on a cluster of heterogeneous machines. In a Storm cluster, there are two kinds of nodes, the master node and the worker nodes. The master node runs a daemon called *Nimbus* that is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. Each worker node runs a daemon called *Supervisor* that listens for work assigned to its machine and controls worker processes based on instructions from Nimbus. All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster, which provides a distributed, open-source coordination service for distributed applications. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless; all state is kept in Zookeeper or on local disk.

A stream in Storm is an unbounded sequence of tuples, which is created and processed in parallel in a distributed fashion. Each stream is defined with a schema, like a table in databases. Users create *topologies* in Storm to process streaming data[1]. A topology is a graph of computation in which each node is a primitive provided by Storm to transform data streams. Each topology includes many worker processes which spread across machines in the cluster. There are two basic primitives in Storm, *spouts* and *bolts*. A spout is a source of streams in a topology, that reads data from external data sources and emits them into the topology. All processing in topologies is done in bolts, which possibly emit new streams and often cooperate with each other to finish complex stream transformations. Both spouts and bolts can emit more than one streams. A Storm topology example can be found in Figure 2.2. In Storm, a task performs the actual data processing. To achieve high throughput, each spout or bolt that users implement in their code executes as many tasks across the cluster. The links between nodes (spouts or bolts) in a topology indicate how data streams are passed around between nodes. For each stream received by a bolt, users need to specify a *stream grouping* which defines how that stream should be partitioned among the bolt's tasks.

Storm guarantees that every spout tuple will be fully processed by the topology with at-least-once semantics [24]. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed through acknowledgements. If Storm fails to detect that a spout tuple has been completed within a timeout, then it fails the tuple and replays it later.

[1] A topology in Storm is similar to a job in Hadoop.

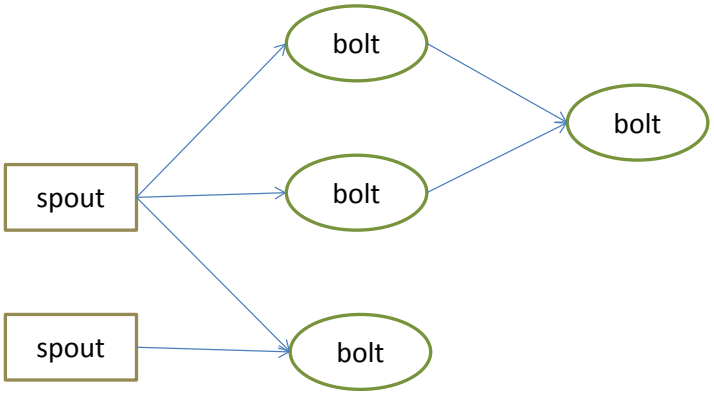


Figure 2.2 – Example of a Storm topology

2.2.2 Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams [23]. Unlike Apache Storm that processes tuples in streams one-at-a-time, Spark Streaming processes tuples in streams in a micro-batch fashion. Namely, as shown in Figure 2.3, Spark Streaming receives input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final result stream which is also in batches. Each batch contains data from a certain time interval.

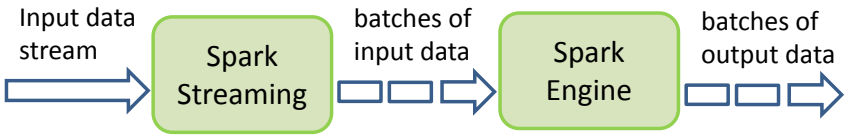


Figure 2.3 – Spark Streaming processing example

Spark Streaming provides a high-level abstraction called discretized stream or DStream [72], which represents a continuous stream of data. DStreams can be created either from input data streams from external sources or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs, and any operation applied on a DStream translates to operations on the underlying RDDs. Spark Streaming provides users a high-level API which hides most of the details of DStream operations for convenience.

To be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.), Spark Streaming provides checkpointing mechanism to keep enough information periodically in a fault-tolerant storage system (e.g., HDFS), such that it can recover from failures.

Chapter 3

DiNoDB: an Interactive-speed Query Engine for Temporary Data

3.1 Introduction

In recent years, modern large-scale data analysis systems have flourished. For example, systems such as Hadoop and Spark [62, 69] focus on issues related to fault-tolerance and expose a simple yet elegant parallel programming model that hides the complexities of synchronization. Moreover, the batch-oriented nature of such systems has been complemented by additional components (e.g., Storm and Spark streaming [70, 72]) that offer (near) real-time analytics on data streams. The communion of these approaches is now commonly known as the "Lambda Architecture" (LA) [73]. In fact, LA is split into three layers, i) the *batch layer* (based on e.g., Hadoop/Spark) for managing and preprocessing append-only raw data, ii) the *speed layer* (e.g., Storm/Spark streaming) tailored to analytics on recent data while achieving low latency using fast and incremental algorithms, and iii) the *serving layer* (e.g., Hive [63], Spark SQL [69], Impala [36]) that exposes the batch views to support ad-hoc queries written in SQL, with low latency.

The problem with such existing large scale analytics systems is twofold. First, combining components (layers) from different stacks, though desirable, raises performance issues and is sometimes not even possible in practice. For example, companies who have expertise in, e.g., Hadoop and traditional SQL-based (distributed) DBMSs, would arguably like to leverage this expertise and use Hadoop as the batch processing layer and DBMSs in the serving layer. However, this approach requires an expensive transform/load phase to, e.g., move data from Hadoop's HDFS and load it into a DBMS [88], which might be impossible to amortize, in particular in scenarios with a narrow processing window, *i.e.,* when working on *temporary data* which may be simply dropped after executing a few queries.

Second, although many SQL-on-Hadoop systems emerged recently, they are not well designed for (short-lived) ad-hoc queries, especially when the data remains in its native, uncompressed, format such as text-based CSV files. To achieve high performance, these systems [35] prefer to convert data into their specific column-based data format, e.g., ORC [87] and Parquet [86]. This works perfectly when both data and analytic queries (that is, the full workload) are in their final production stage. Namely, these self-describing, optimized data formats play an increasing role in modern data analytics, and this especially becomes true once data has been cleaned, queries have been well designed, and analytics algorithms have been tuned. However, when users perform data exploration tasks and algorithm tuning, that is when the data is *temporary*, the original data format typically remains unchanged — in this case, premature data format optimization is typically avoided, and simple text-based formats such as CSV and JSON files are preferred. In this case, current integrated data analytics systems can under-perform. Notably, they often fail to leverage decades old techniques for optimizing the performance of (distributed) DBMSs, *e.g.,* indexing, that is usually not supported.

In summary, contemporary data scientists face a wide variety of competing approaches targeting the *batch* and the *serving* layer. Nevertheless, we believe that these approaches often have overly strict focus, in many cases ignoring one another, thus failing to explore potential benefits from learning from each other.

In this chapter, we propose DiNoDB, an interactive-speed query engine that addresses the above issues. Our approach is based on a seamless integration of batch processing systems (e.g., Hadoop MapReduce and Apache Spark) with a distributed, fault-tolerant and scalable interactive query engine for *in-situ* analytics on *temporary* data. DiNoDB integrates the batch processing with the serving layer, by extending the ubiquitous Hadoop I/O API using *DiNoDB I/O decorators*. This mechanism is used to create, as an additional output of batch processing, a wide range of *metadata*, *i.e.,* auxiliary data structures such as positional maps and vertical indexes, that DiNoDB uses to speed-up the interactive data analysis of *temporary* data files for data exploration and algorithm tuning. Our solution effectively brings together the batch processing and the serving layer for big data workflows, while avoiding any loading and data (re)formatting costs. While, clearly, no data analytics solution can fit all big data use cases, when it comes to ad-hoc interactive queries with a narrow processing window, DiNoDB outperforms state-of-the-art distributed query engines, such as Hive, Stado, Spark SQL and Impala.

In summary, our main contributions in this chapter include:

- The design of DiNoDB, a distributed interactive query engine. DiNoDB leverages modern multi-core architectures and provides efficient, distributed, fault-tolerant and scalable in-situ SQL-based querying capabilities for temporary data. DiNoDB (Distributed NoDB) is the first distributed and scalable instantiation of the NoDB paradigm [41], which was previously instantiated only in centralized systems.

- Proposal and implementation of the DiNoDB I/O decorators approach to interfacing batch processing and interactive query serving engines in a data analytics system. DiNoDB I/O decorators generate, as a result of the batch processing phase, metadata that aims to facilitate and expedite subsequent interactive queries.
- Detailed performance evaluation and comparative analysis of DiNoDB versus state-of-the-art systems including Hive, Stado, Spark SQL and Impala.

The rest of the chapter is organized as follows. In Section 3.2, we further motivate our approach and the need for a system such as DiNoDB. In Section 3.3, we describe the high-level design of DiNoDB. In Section 3.4, we introduce how DiNoDB integrates the batch processing with its query engine. Section 3.5 covers the design of DiNoDB interactive query engine. In Section 3.6 we give our experimental results based on both synthetic and real-life datasets. Section 3.7 overviews related work. Section 3.8 concludes the chapter.

3.2 Applications and use cases

In this section, we overview some of the contemporary uses cases which span both batch processing and interactive analytics in the data analytics flows. These use cases include machine learning (Section 3.2.1) and data exploration (Section 3.2.2). For each of these use cases we discuss: i) how better communication between the batch processing and the serving layer that DiNoDB brings may help, and ii) the applicability of our *temporary data* analytics approach.

3.2.1 Machine learning

In the first use case – which we evaluate in Section 3.6 – we take the perspective of a user (e.g., a data scientist) focusing on a complex data clustering problem. Specifically, we consider the task of learning *topic models* [79], which amounts to automatically and jointly clustering words into "topics", and documents into mixtures of topics. Simply stated, a topic model is a hierarchical Bayesian model that associates with each document a probability distribution over "topics", which are in turn distributions over words. Thus, the output of a topic modeling data analysis can be thought of as a (possibly very large) matrix of probabilities: each row represents a document, each column a topic, and the value of a cell indicates the probability for a document to cover a particular topic.

In such a scenario, depicted in Figure 3.1, the user typically faces the following issues: i) topic modeling algorithms (e.g., Collapsed Variational Bayes (CVB) [78]) require parameter tuning, such as selecting an appropriate number of topics, the number of unique features to consider, distribution smoothing factors, and many more; and ii) computing

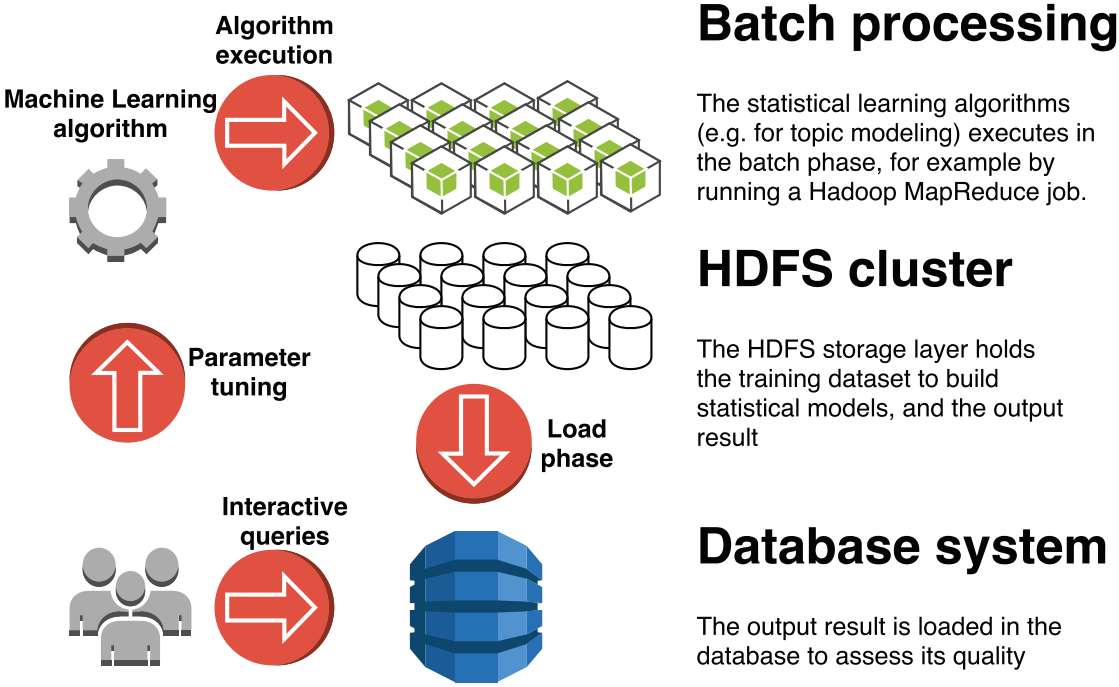


Figure 3.1 – Machine learning use case.

"modeling quality" typically requires a trial-and-error process whereby only domain-knowledge can be used to discern a good clustering from a bad one. In practice, such a scenario illustrates a typical "development" workflow which requires: a *batch processing phase* (e.g., running CVB), an *interactive query phase on temporary data* (i.e., on data interesting in relatively short periods of time), and several iterations of both phases until algorithms are properly tuned and final results meet users' expectations.

DiNoDB explicitly tackles such "development" workflows. Unlike current approaches, which generally require a long and costly data loading phase that considerably increases the data-to-insight time, DiNoDB allows querying temporary data in-situ, and exposes a standard SQL interface to the user. This simplifies query analysis and reveals the main advantage of DiNoDB in this use case, that is the removal of the *temporary data* loading phase, which today represents one of the main operational bottlenecks in data analysis. Indeed, the traditional data loading phase makes sense when the workload (i.e., data and queries) is stable in the long term. However, since data loading may include creating indexes, serialization and parsing overheads, it is reasonable to question its validity when working with temporary data, as in our machine learning use case.

The key design idea behind DiNoDB is that of shifting the part of the burden of a traditional load operation to the batch processing phase of a "development" workflow. While batch data processing takes place, DiNoDB piggybacks the creation of distributed posi-

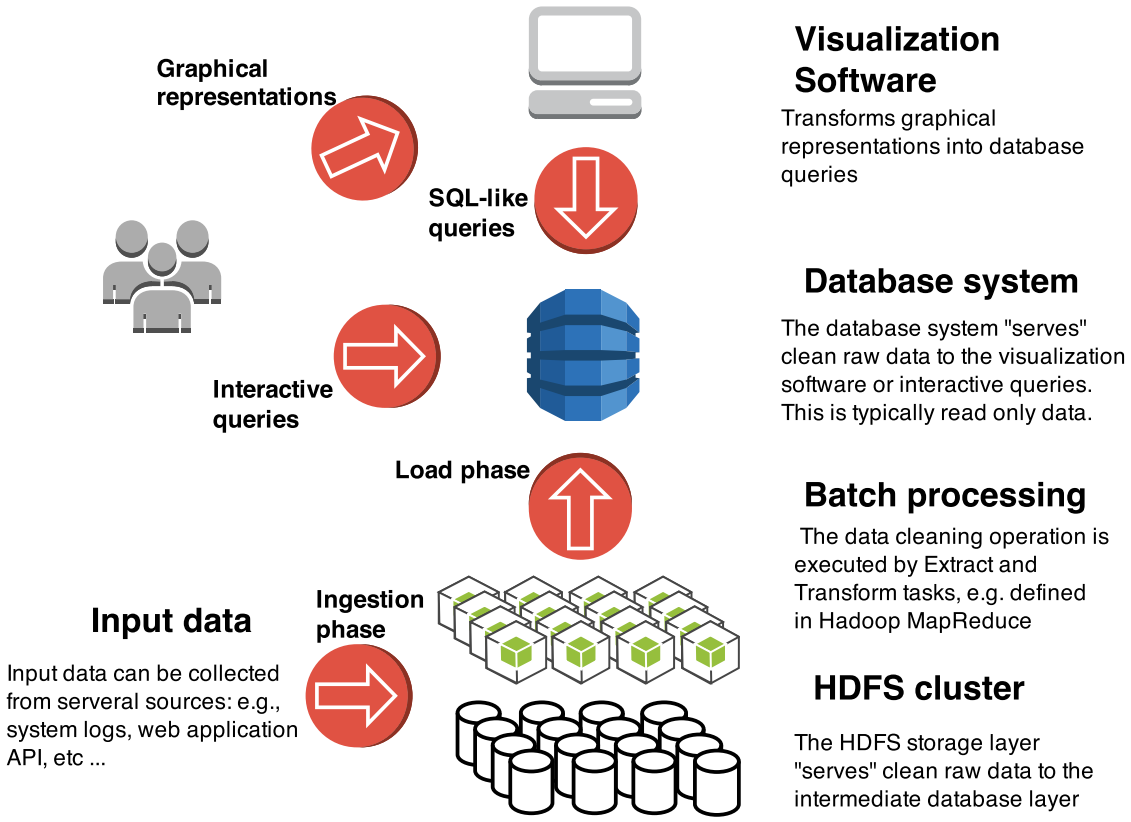


Figure 3.2 – A typical data exploration architecture.

tional maps and vertical indexes (see Section 3.4 for details) to improve the performance of interactive user queries on the temporary data. Interactive queries operate directly on temporary data files produced by the batch processing phase, which are stored on a distributed file system such as HDFS [43].

3.2.2 Data exploration

In this section, we discuss another prominent use case – which we also evaluate in Section 3.6 – and which is another important motivation for our work. Here we consider a user involved in a preliminary, yet often fundamental and time-consuming, *data exploration* task. Typically, the user collects data from different sources (e.g., an operational system, a public API) and stores it on a distributed file system such as HDFS for subsequent processing. However, before any useful processing can happen, data needs to be cleaned and studied in detail.

Data exploration generally requires visualization tools, that assist users in their preliminary investigation by presenting the salient features of raw data. Current state-of-the-art architectures for data exploration can be summarized as in Figure 3.2. A *batch processing phase* ingests dirty data to produce temporary data; such data is then loaded

into a database system that supports an *interactive query phase*, whereby a visualization software (e.g., Tableau Software) translates user-defined graphical representations into a series of queries that the database system executes. Such queries typically "reduce" data into aggregates, by filtering, selecting subsets satisfying predicates and by taking representative samples (e.g., by focusing on top-k elements).

In the scenario depicted above, DiNoDB reduces the data-to-insight time, by allowing visualization software or users to directly interact with the raw representation of temporary data, without paying the cost of the load phase that traditional database systems require, nor data format transformation overheads. In addition, the metadata that DiNoDB generates by piggybacking on the batch processing phase (while data is cleaned), substantially improves query performance, making it a sensible approach for applications where interactivity matters.

3.3 DiNoDB high-level design

In this section, we present the high-level architecture design of DiNoDB. DiNoDB is designed to provide a seamless integration of batch processing systems such as Hadoop MapReduce and Spark, with a distributed solution for in-situ data analytics on large volumes of temporary, raw data files. First, in Section 3.4 we explain how DiNoDB extends the ubiquitous Hadoop I/O API using DiNoDB I/O decorators, a mechanism that generates a wide range of auxiliary metadata structures to speed-up the interactive data analysis using the DiNoDB query engine. Then, in Section 3.5 we describe the DiNoDB query engine, which leverages the metadata generated in the batch processing phase to achieve interactive-speed query performance.

The batch processing phase (e.g., in the machine learning and data exploration use cases outlined previously) typically involves the execution of (sophisticated) analysis algorithms. This phase might include one or more batch processing jobs, whereby output data is written to HDFS.

The key idea behind DiNoDB is to leverage batch processing as a preparation phase for future interactive queries. Namely, DiNoDB enriches the Hadoop I/O API with DiNoDB I/O decorators. Such mechanism piggybacks the generation of metadata by *pipelining* the output tuples produced by the batch engine into a series of specialized *decorators* that store auxiliary metadata along with the original output tuples. We further detail DiNoDB I/O decorators and metadata generation in Section 3.4.

In addition to the metadata generation, DiNoDB capitalizes on data preprocessing by keeping output data in-memory. To be more specific, we configure Hadoop to store output data and metadata in RAM, using the `ramfs` file system as an additional mount

point for HDFS.[1] Our DiNoDB prototype supports both `ramfs` and disk mount points for HDFS, a design choice that allows supporting queries on data that cannot fit in RAM.

Both the output data and metadata are consumed by the DiNoDB interactive query engine. As we detail in Section 3.5, the DiNoDB interactive query engine is a massively parallel processing engine that orchestrates several DiNoDB nodes. Each DiNoDB node is an optimized instance of PostgresRaw [41], a variant of PostgreSQL tailored to querying temporary data files produced in the batch processing phase. To ensure high performance and low query execution times, we co-locate DiNoDB nodes and HDFS DataNodes, where the two share data through HDFS, and in particular through its in-memory, `ramfs` mount.

In the remaining of this chapter we assume that both the raw and the *temporary* data ingested and produced by the batch processing phase, and used in the query serving phase are in a structured textual data format (e.g., comma-separated value files).

3.4 DiNoDB I/O decorators

DiNoDB piggybacks the generation of auxiliary metadata on the batch processing phase using DiNoDB I/O decorators. DiNoDB I/O decorators are designed to be a non-intrusive mechanism, that seamlessly integrates systems supporting the classical Hadoop I/O API, such as Hadoop MapReduce and Apache Spark. DiNoDB I/O decorators operate at the end of the batch processing phase for each final task that produces output tuples, as shown in Figure 3.5. Instead of writing output tuples to HDFS directly, using the standard Hadoop I/O API, the tasks use DiNoDB I/O decorators, which build a metadata generation pipeline, where each decorator iterates over streams of output tuples and compute the various kinds of metadata.

Next, we introduce metadata currently supported in our prototype, and outline the implementation of our decorators to generate *positional maps* [41], *vertical indexes*, *statistics* and *data samples*.

3.4.1 Positional maps

Positional maps are data structures that DiNoDB uses to optimize in-situ querying. Contrary to a traditional database index, a positional map indexes the structure of the file and

[1] This technique has been independently considered for inclusion in a recent patch to HDFS [76] and in a recent in-memory HDFS alternative called Tachyon [82]. Finally it is now added in the latest version of Apache Hadoop [64]

	Attr_x	Attr_y	...	RowLength
Row 0	offset	offset	...	len
Row 1	offset	offset	...	len
Row 2	offset	offset	...	len
...	...	...	...	...
Row n	offset	offset	...	len

Figure 3.3 – DiNoDB positional maps

not the actual data. A positional map (shown in Figure 3.3) contains relative positions of attributes in a data record, as well as the length of each row in the data file. During query processing, the information contained in the positional map can be used to jump to the exact position (or as close as possible) of an attribute, significantly reducing the *cost of tokenizing and parsing* when a data record is accessed.

Algorithm 1 Positional map generation.

```
1:  procedure INITIALIZE
2:      open pmstream
3:  end procedure
4:  procedure PROCESS(tuple)
5:      tuple consists of key attr_0 and value [attr_1, attr_2, ··· , attr_n]
6:      initialize tuplestring, as an empty string
7:      for attr ∈ [attr_0, attr_1, attr_2, ··· , attr_n] do
8:          if attr is sampled then
9:              offset ← getLength(tuplestring)
10:             pmstream.write(offset)
11:         end if
12:         tuplestring ← tuplestring+',' +attr
13:     end for
14:     len ← getLength(tuplestring)
15:     pmstream.write(len)
16:     tuple.setString(tuplestring)
17:     pass tuple to next decorator
18: end procedure
19: procedure CLOSE
20:     close pmstream
21: end procedure
```

To keep the size of the generated positional map relatively small to the size of a data file, the *positional map decorator* implements *uniform sampling*, to store positions only for a subset of the attributes in a file. The *positional map decorator* implements *sampling*, to store positions only for a subset of the attributes in a file. The user can either provide

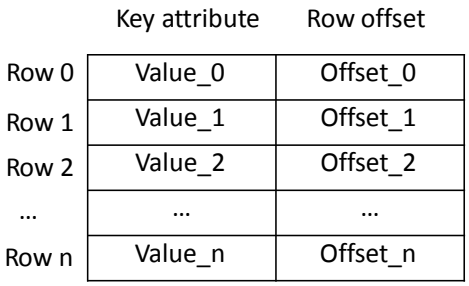


Figure 3.4 – DiNoDB vertical indexes

a sampling rate so that the positional map decorator will perform uniform sampling, or directly indicate which attributes are sampled.

Algorithm 1 demonstrates how the positional map decorator generate positional maps: The positional map decorator is initialized by opening a positional map file stream $pmstream$ (line 1-3); the decorator continuously receives tuples from which lists of attributes can be extracted (line 5); for each tuple, the decorator constructs string $tuplestring$ by iterating through all attributes (line 7-13); during string construction, the offsets of sampled attributes are written to $pmstream$ (line 8-11); when $tuplestring$ is fully formed, its length is also written to $pmstream$ (line 14-15); then the original tuple and its $tuplestring$ is passed to the next decorator (line 16-17); when all tuples are processed, $pmstream$ is closed so that the positional map file is finalized (line 19-21).

An approximate positional map can still provide tangible benefits: indeed, if the requested attribute is not part of the sampled positional map, a nearby attribute position is used to navigate quickly to the requested attribute without significant overhead. In Section 3.6.2.5, we show the effect of different sampling rates in the query execution performance.

3.4.2 Vertical indexes

The positional map can reduce the CPU processing cost associated with parsing and tokenizing data; to provide the performance benefit of an index-based access plan, DiNoDB uses a *vertical index decorator* that accommodates one or more *key attributes* for which vertical indexes are created at the end of the batch processing phase. Such vertical indexes can be used to quickly search and retrieve data without having to perform a full scan on the temporary output file.

Figure 3.4 shows the in-memory data structure of a vertical index. An entry in the vertical index has two fields for each record of the output data: the key attribute value and the record row offset value. As such, every key attribute value is associated with a par-

ticular row offset in the data file, which DiNoDB nodes use to quickly access a specific row of a file. As decorators generate metadata in a single pass, the key attribute values are not required to be unique or sorted. Each time when the vertical index decorator receives a tuple, it generates the index entry for this tuple which is output to a vertical index file.

Algorithm 2 Vertical indexes generation.

```
1:  procedure INITIALIZE
2:      open vistream
3:      offset ← 0
4:  end procedure
5:  procedure PROCESS(tuple)
6:      tuple consists of key attr_0 and value [attr_1, attr_2, ··· , attr_n]
7:      len ← 0
8:      for attr ∈ [attr_0, attr_1, attr_2, ··· , attr_n] do
9:          if attr is key attribute then
10:             vistream.write(attr)
11:         end if
12:         len ← len+ getLength(attr) +1
13:     end for
14:     vistream.write(offset)
15:     len ← len − 1
16:     offset ← offset + len
17: end procedure
18: procedure CLOSE
19:     close vistream
20: end procedure
```

Algorithm 2 gives the details of how the vertical index decorator works. To initialize, the vertical index decorator opens a vertical index file stream $vistream$ and sets a global variable $offset$ to 0 (line 1-4); the decorator continuously receives tuples from which lists of attributes can be extracted (line 6); for each tuple, the decorator writes the value of the key attribute and the current line $offset$ to $vistream$ (line 7-14); the $offset$ is updated by accumulating the string length of each attribute and each field delimiter (line 7-16).

3.4.3 Statistics

Modern database systems rely on statistics to choose efficient query execution plans. Query optimization requires knowledge about the nature of processed data that helps ordering operators such as joins and selections; however, such statistics are available only after loading the data or after a preprocessing phase. Currently DiNoDB I/O deco-

rators can compute the number of records and the number of distinct values for specific attributes from the batch processing phase as the statistics of the output data. To achieve this, our *statistics decorator* uses the near-optimal probabilistic counting algorithm HyperLogLog [83].

The detailed algorithm can be found in Algorithm 3, in which the number of records and the number of distinct values for each attribute is written to a statistics file stream $statsstream$. When multiple statistics decorator instances are used (e.g., each with a reducer in a Hadoop job), each decorator instance generates the statistics for a single data partition (e.g., outputted from a reducer). In order to compute the number of distinct values of attributes in the entire dataset, DiNoDB generates a global variable max_zeros by keeping the max integer for each element from all generated variables max_zeros. With the global max_zeros, DiNoDB can calculate the total number of distinct values for each attribute.

Statistics on attribute cardinality are used by DiNoDB to improve the quality of the query plans for complex queries, e.g., involving join operations. Other kinds of statistics (e.g, skew in the distribution of values per attribute) can be easily supported by DiNoDB I/O decorators as long as there exist a one-pass algorithm to generate them.

3.4.4 Data samples

When the volume of raw data produced by the batch processing phase is very large, it is often a requirement to work, e.g., for visualization purposes, on a concise yet representative subset of the preprocessing output. Therefore, the *data samples decorator* is designed to piggyback the creation of a sampled version of the raw data, that is stored alongside the output of batch processing. By using the *Reservoir Sampling* algorithm [95], the our data samples decorator can randomly choose a data sample of fixed size in one pass without requiring the data to be fit in memory. As such, DiNoDB can query the sampled raw data and expose increased interactivity to visualization software.

3.4.5 Implementation details

DiNoDB I/O decorators are designed to be a non-intrusive mechanism, that seamlessly integrates systems supporting the classical Hadoop I/O API, such as Hadoop MapReduce and Apache Spark. DiNoDB I/O decorators operate at the end of the batch processing phase for each final task that produces output tuples, as shown in Figure 3.5. Instead of writing output tuples to HDFS directly, using the standard Hadoop I/O API, the tasks use DiNoDB I/O decorators, which build a metadata generation pipeline, where each decorator iterates over streams of output tuples and compute the various kinds of metadata described above.

Algorithm 3 Statistics generation.

procedure INITIALIZE
  $row\_count \leftarrow 0$
  $k \leftarrow 8$
  $num\_buckets \leftarrow 2^k$
  initialize a two-dimensional integer array $max\_zeros[num\_attr][num\_buckets]$
end procedure
procedure PROCESS(tuple)
  tuple consists of key $attr_0$ and value $[attr_1, attr_2, \cdots, attr_n]$
  $row\_count \leftarrow row\_count + 1$
  $index\_attr \leftarrow 0$
  for $attr \in [attr_0, attr_1, attr_2, \cdots, attr_n]$ **do**
    $h \leftarrow$ hash($attr$)
    $bucket \leftarrow h \;\&\; (num\_buckets - 1)$
    $bucket\_hash \leftarrow h >> k$
    $max\_zeros[index\_attr][bucket] \leftarrow$ max($max\_zeros[index\_attr][bucket]$, TRAILING_ZEROS($bucket\_hash$))
    $index\_attr \leftarrow index\_attr + 1$
  end for
end procedure
function TRAILING_ZEROS(num)
  if $num = 0$ **then**
    return 32 ▷ Assumes 32-bit integers
  end if
  $p \leftarrow 0$
  while $(num >> p) \;\&\; 1 = 0$ **do**
    $p \leftarrow p + 1$
  end while
  return p
end function
procedure CLOSE
  open $statsstream$
  $statsstream$.write(row_count)
  $statsstream$.write(max_zeros)
  for $i \leftarrow 0$ to $len(max\_zeros) - 1$ **do**
    $num\_distinct\_value \leftarrow 2^{\text{sum}(max\_zeros[i])/num\_buckets} * num\_buckets * 0.79402$
    $statsstream$.write($num\_distinct\_value$)
  end for
  close $statsstream$
end procedure

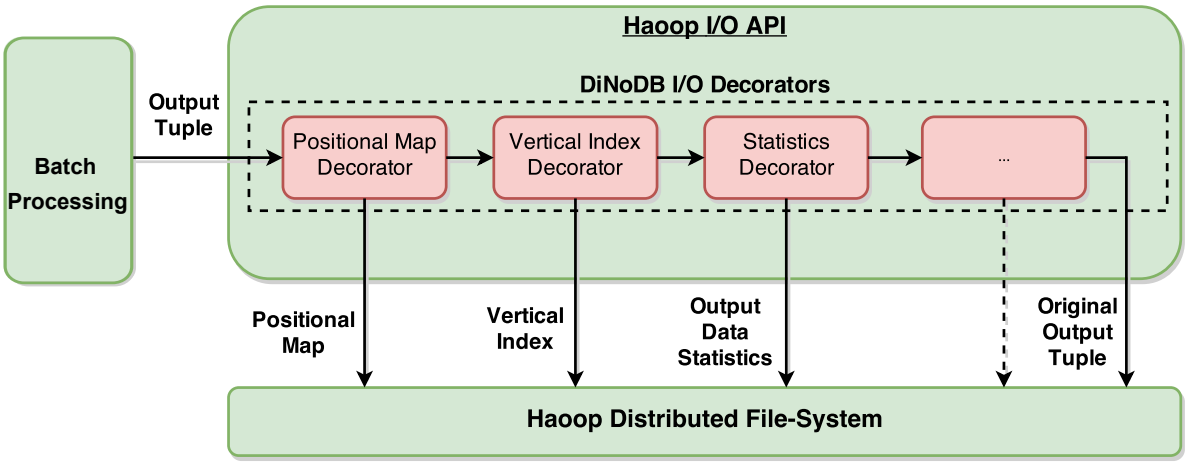


Figure 3.5 – DiNoDB I/O decorator overview.

To use DiNoDB I/O decorators, Hadoop users need to replace the vanilla Hadoop `OutputFormat` class by a new module called `DiNoDBOutputFormat`. Our prototype currently supports the `TextOutputFormat` sub-class, which allows DiNoDB to operate on textual data formats. Specifically, the `DiNoDBTextOutputFormat` module implements a new `DiNoDBArrayWritable` class which is used to generate both the output data and its associated metadata. If users work with Spark, before saving their result RDD to HDFS (by method `saveAsTextFile`) they need to first cast that result RDD to a `DiNoDBRDD`, which internally uses `DiNoDBOutputFormat` as `OutputFormat` class.

DiNoDB I/O decorators are configured by passing a configuration file to each batch processing job in Hadoop or by setting parameters of `DiNoDBRDD` in Spark. Users specify which metadata to generate and indicate parameters, such as the sampling rate to use for the generation of positional maps and the key attributes for the generation of vertical indexes.

Discussion. Although currently DiNoDB focuses on textual data format, the same idea of generating metadata could also be applied to other data formats, like binary files. Depending on different input data format, generated metadata may be different. For example, if the data is in FITS [94] data format, positional map is not needed anymore because each tuple and attribute is usually located in a well-known location. However, vertical indexes and statistics would still help.

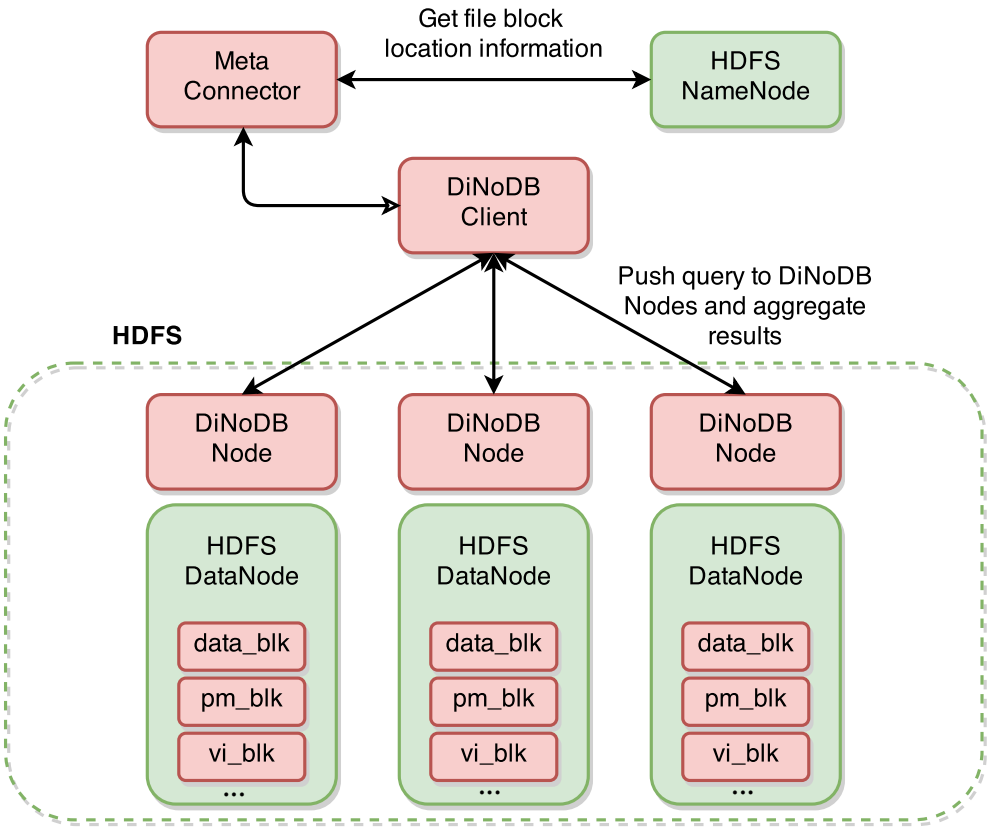


Figure 3.6 – Architecture of the DiNoDB interactive query engine.

3.5 The DiNoDB interactive query engine

At a high level (see Figure 3.6), the DiNoDB interactive query engine consists of a set of DiNoDB nodes, orchestrated using a massively parallel processing (MPP) framework. In our prototype implementation, we use the Stado MPP framework [14], which nicely integrates PostgreSQL-based database engines. DiNoDB ensures data locality by co-locating DiNoDB nodes with HDFS DataNodes.

In the following sections, we first describe the DiNoDB client (Section 3.5.1) and the DiNoDB nodes (Section 3.5.2). Then, we describe how DiNoDB achieves fault-tolerance (Section 3.5.3).

3.5.1 DiNoDB clients

A DiNoDB client serves as entry point for DiNoDB interactive queries. It provides a standard shell command interface, hiding the network layout and the distributed system architecture from the users. As such, applications can use DiNoDB just like a traditional DBMS.

DiNoDB clients accept application requests (queries), and communicate with DiNoDB nodes. When a DiNoDB client receives a query, it fetches the metadata for the "tables" (output files of the batch phase) indicated in the query, using the *MetaConnector* module. The MetaConnector (see Figure 3.6) operates as a proxy between DiNoDB and the HDFS NameNode, and is responsible for retrieving HDFS metadata information like partitions and block locations of raw data files. Using HDFS metadata, the MetaConnector guides the DiNoDB clients to query the DiNoDB nodes that hold raw data files relevant to the user queries. Additionally, the MetaConnector remotely configures DiNoDB nodes so that they can build the mapping between "tables" and the related HDFS blocks, including all data file blocks and metadata blocks, e.g., positional map blocks and vertical index blocks. In summary, the anatomy of a query execution is as follows: i) using the MetaConnector, a DiNoDB client learns the location of every raw file block and pushes the query to the respective DiNoDB nodes; ii) the DiNoDB nodes process the query in parallel; and finally, iii) the DiNoDB client aggregates the result.

Note that, since the DiNoDB nodes are co-located with the HDFS DataNodes, DiNoDB inherits fault-tolerance from HDFS replication. If a DiNoDB client detects a failure of a DiNoDB node, or upon the expiration of a timeout on DiNoDB node's responses, the DiNoDB client will issue the same query to another DiNoDB node holding a replica of the target HDFS blocks. We discuss DiNoDB fault-tolerance in more details in Section 3.5.3.

3.5.2 DiNoDB nodes

The DiNoDB nodes are based on PostgresRaw [41], a query engine optimized for in-situ querying. In the following, we first briefly recall how PostgresRaw differs from native PostgreSQL and then explain all the details behind DiNoDB nodes, and differences with respect to PostgresRaw.

PostgresRaw. PostgresRaw is a centralized instantiation of the NoDB paradigm [41], and is a variant of PostgreSQL that avoids the data loading phase and executes queries directly on data files. PostgresRaw adopts in-situ querying instead of loading and preparing the data for queries. To accelerate query execution, PostgresRaw tokenizes only necessary attributes and parses only qualified tuples. Moreover, PostgresRaw *incrementally* builds a positional map, which contains relative positions of attributes in a line, and updates it during the query execution: as such, the more queries are executed, the more complete and useful (for performance) the positional map will be.

From PostgresRaw to DiNoDB nodes. DiNoDB nodes instantiate customized PostgresRaw databases which execute user queries, and are co-located with HDFS DataNodes. In the vanilla PostgresRaw [41] implementation, a "table" maps to a single data file. Since the HDFS files are instead split into multiple blocks, DiNoDB nodes use a new

file reader mechanism that can access data on HDFS and maps a "table" to a list of data file blocks. In addition, the vanilla PostgresRaw implementation is a multiple-process server, which forks a new process for each new client session, with individual metadata and data cache per process. Instead, DiNoDB nodes place metadata and data in *shared memory*, such that user queries – which are sent through the DiNoDB client – can benefit from them across multiple sessions.

DiNoDB nodes can take advantage of the fact that data is naturally partitioned into HDFS blocks to leverage modern multi-core processors. Hence, data and the associated metadata can be easily accessed by multiple instances of PostgresRaw, to allow node level parallelism. DiNoDB users can selectively indicate whether raw data files are placed on disk or in memory. Hence, DiNoDB nodes can seamlessly benefit from a memory-backed file system to dramatically improve query execution times.

Exploiting metadata. DiNoDB nodes leverage positional map files generated in the preprocessing phase when executing queries, instead of only populating them incrementally as in PostgresRaw. In the case of the approximate positional maps, DiNoDB nodes use the sampled attributes as anchor points, to retrieve nearby attributes within the same row, required to satisfy a query. When vertical indexes are available, DiNoDB nodes use them to speed up queries with low selectivity, by employing an index-based access plan as a replacement for a full sequential scan. Both the positional map and the vertical index files are loaded by a DiNoDB node when the first query requires them. As our performance evaluation shows (see Section 3.6), the metadata loading time is almost negligible, when compared to the execution time of the query.

Data update. If new data is injected to an existing table without using DiNoDB I/O decorators (e.g., manually upload), there is no associated metadata pre-generated. In this case, DiNoDB nodes can still exploit the available metadata and do not require new metadata for the newly added data to process queries. Partially-available metadata can still accelerate query execution. HDFS is an immutable filesystem, which means that both data and metadata can not be modified after being written. If part of data is deleted, the associated metadata needs to be deleted as well.

3.5.3 Fault tolerance

In a nutshell, the key idea behind DiNoDB fault tolerance is to exploit HDFS n-way replication, in which every HDFS block on a given node is replicated to n-1 other nodes. As DiNoDB nodes co-locate with HDFS DataNodes, user queries can be directed to multiple nodes: in case one node is not available, DiNoDB clients automatically forward queries to other nodes with replicas of the data. By virtue of pro-active request redirection, DiNoDB can address the issues related to latency-tail tolerance [37].

Table 3.1 – Comparison of Systems

Systems	DiNoDB	ImpalaT	ImpalaP	SparkSQL	SparkSQLc	Hive on Tez	Stado
Loading requirement	no	no	yes	no	yes	no	yes
Indexing support	yes	no	column-oriented	no	no	no	yes

However, the default HDFS replication mechanism is not suited for DiNoDB: indeed, HDFS does not guarantee data and metadata generated by DiNoDB I/O decorators to be replicated on the same nodes. For example, some blocks assigned to DataNode D_1 may be replicated across DataNodes D_2 and D_3, whereas other blocks assigned to D_1 might be replicated differently, e.g., on DataNodes D_4 and D_5.

We thus proceed with the design of a new replication mechanism for HDFS that is tailored to DiNoDB, which achieves two objectives: (i) it co-locates data blocks with the corresponding metadata blocks and (ii) it allows selecting different "storage levels" for replicas, to save on cluster resources. To address data/metadata co-location, our DiNoDB prototype implements a *per-node* n-way replication. Every block assigned to a given DataNode D_i is systematically replicated across the same DataNodes D_j and D_k. We are aware that, on the long run, such simple approach may lead the HDFS subsystem to be poorly balanced. An alternative approach that we are currently considering is to create a new Hadoop Output Format that, similarly to Apache Parquet [86], supports "containers" that include data and metadata. Finally, DiNoDB supports different storage levels across replicas: as such, a "primary" replica can be tagged to be stored in the HDFS `ramfs` mount point, while "secondary" replicas are instead stored in an HDFS disk-based mount point [84].

3.6 Experimental evaluation

We now present a detailed experimental evaluation of DiNoDB using both real and synthetic datasets. We compare DiNoDB against state-of-the-art data analytics systems, including Impala, Spark SQL, Hive on Tez and Stado. Stado, an MPP system based on PostgreSQL, needs to load the data before executing queries, while Hive on Tez executes queries directly on data files stored in HDFS. Impala and Spark SQL can execute queries either directly on data files or after creating a copy of data in a new file format such as Parquet. Additionally, Spark SQL provides functionality to cache data files in memory as Resilient Distributed Dataset (RDD) [89].

Our goal here is to perform a comparative analysis across all systems showing the *aggregate query execution time*, which accounts for the total time to execute a certain number of queries on temporary output data of the batch processing phase. Additionally, we

highlight the merits and applicability of metadata generated by the DiNoDB I/O decorators.

3.6.1 Experimental Setup

All the experiments are conducted in a cluster of 9 machines, with 4 cores, 16 GB RAM and 1 Gbps network interface each. The underlying distributed file system is HDFS. Eight machines are configured as worker nodes (DataNode), while the remaining one is acting as a coordinator (NameNode) in HDFS. To avoid disk bottlenecks, all the datasets are stored in HDFS with a `ramfs` mount point on each DataNode.

We compare DiNoDB with Impala (version 1.4), Spark SQL (version 1.1.0), Hive (version 0.13.1) on Tez (version 0.4.1) and PostgreSQL-based Stado. For DiNoDB, we assign 3 out of 4 cores to a DiNoDB node and we balance the data of the underlying HDFS file system across the 3 cores. The fourth core acts as a server and coordinates the three client-sessions of PostgresRaw. All the other systems also fully utilize CPU resources, either by multithreading (Impala and Spark SQL) or by multiprocessing (Hive on Tez and PostgreSQL-based Stado). In our experiments, we evaluate Spark SQL in two ways: i) by caching the entire table in memory as RDD before query execution (we label this variant of Spark SQL as SparkSQLc); and ii) the "normal" execution without caching (labeled as SparkSQL). Similarly, we evaluate Impala in two ways: i) by converting the dataset from text format to Parquet format before query execution (labeled as ImpalaP); and ii) the "normal" execution without converting (labeled as ImpalaT). A brief comparison among these systems can be found in Table 3.1.

3.6.2 Experiments with synthetic data

In this section, we use a synthetic dataset of 70 GB containing $5 * 10^7$ tuples. Each tuple has 150 attributes with integers uniformly distributed in the range [0-10^9). The DiNoDB I/O decorators already produce a 3.5 GB positional map file (with 1/10 sampling rate) and a 1.1 GB vertical index file, where we choose the first attribute as the key attribute.

We use as input a sequence of SELECT PROJECT SQL queries that effectively simulate the kind of workload in a data exploration use case, which involves mainly filtering and selecting subsets of data. We discuss the exact queries in more detail in each experiment.

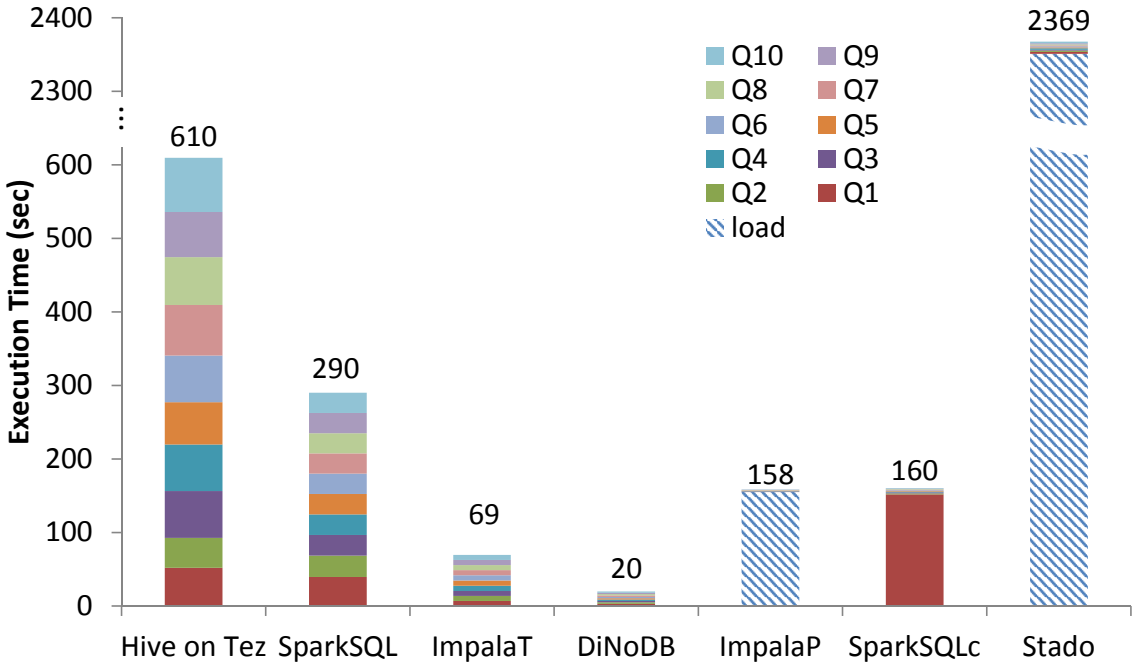


Figure 3.7 – DiNoDB vs. other distributed systems: Positional map reduces the cost of accessing data files.

3.6.2.1 Random queries (stressing PM)

In this experiment, we show the benefit of using positional maps in DiNoDB. We use as input a sequence of 10 SELECT PROJECT SQL queries of the following template: `select ax from table where ay < 100000`. Attributes *ax* and *ay* are randomly selected and the selectivity is 0.1‰. We examine two categories of systems that evaluate queries i) directly on data files and ii) on loaded data. Specifically, Hive on Tez, SparkSQL, ImpalaT and DiNoDB execute queries directly, without any data loading process. ImpalaP and Stado require to load the data before querying, while SparkSQLc uses the first query to load the data in a memory-backed RDD (lazy loading).

Figure 3.7 plots the query execution time of the 10 queries. For queries over loaded data we also report the required loading time. Considering the aggregate query execution time for the 10 queries, DiNoDB is more than three times faster than the second fastest system, ImpalaT. Additionally, when it comes to individual query times DiNoDB consistently outperforms systems executing queries on data files. DiNoDB achieves that by exploiting the positional map that is generated by the DiNoDB I/O decorators to reduce the CPU cost of accessing data files (parsing and tokenizing). On the other hand, SparkSQLc, ImpalaP, and Stado achieve shorter query execution times only after spending 150, 155 and 2352 seconds, respectively, for data loading. In Section 3.6.2.3, we further investigate the trade-off between initially investing time to prepare the data for querying versus quickly accessing the data on files.

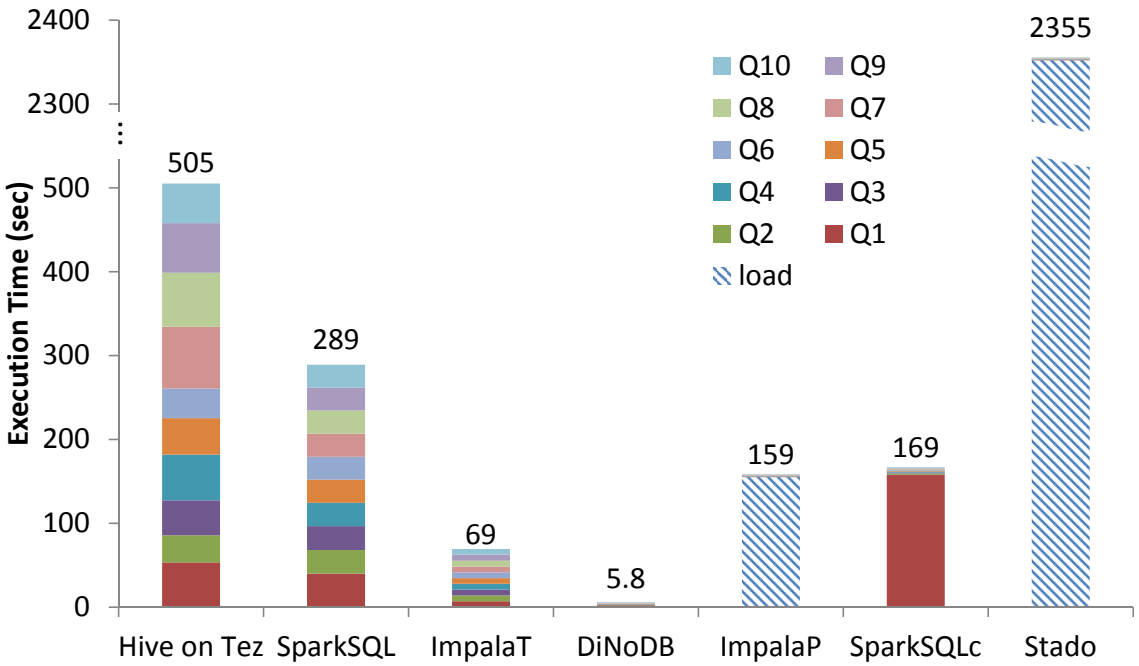


Figure 3.8 – DiNoDB vs. other distributed systems: Vertical indexes significantly improve DiNoDB's performance.

3.6.2.2 Key attribute based queries (stressing VI)

In this experiment, we demonstrate the impact of exploiting the vertical indexes in DiNoDB. We use as input a sequence of 10 SELECT PROJECT SQL queries following the template: `select ax from table where akey < 100000`. The attribute in the WHERE clause is no longer a random attribute, but the attribute which we set as the *key attribute* by an appropriate configuration of the DiNoDB I/O decorators. The selectivity is again 0.1‰.

Figure 3.8 shows the query execution time of the query sequence. DiNoDB significantly benefits from exploiting the generated VI file to perform index scan access to the data (saving CPU and I/O cost). Overall, the average query cost of DiNoDB is less than 1 second. On the other hand, Hive on Tez, SparkSQL and ImpalaT that do not have an indexing mechanism, have similar performance as in the random query experiment above. Since Stado uses PostgreSQL as the local database system in our experiments, it supports index scan on data. However, it needs an additional 9 seconds delay for the index building phase, so in total (including the data loading phase), Stado needs about 2350 seconds before it is able to execute any queries.

Vertical indexes dramatically accelerate query execution in DiNoDB if queries hit the key attribute. In this experiment, DiNoDB's query execution time is competitive compared to Stado, ImpalaP and SparkSQLc (besides the first query which is very slow for SparkSQLc). However, unlike those systems, DiNoDB does not require any data loading

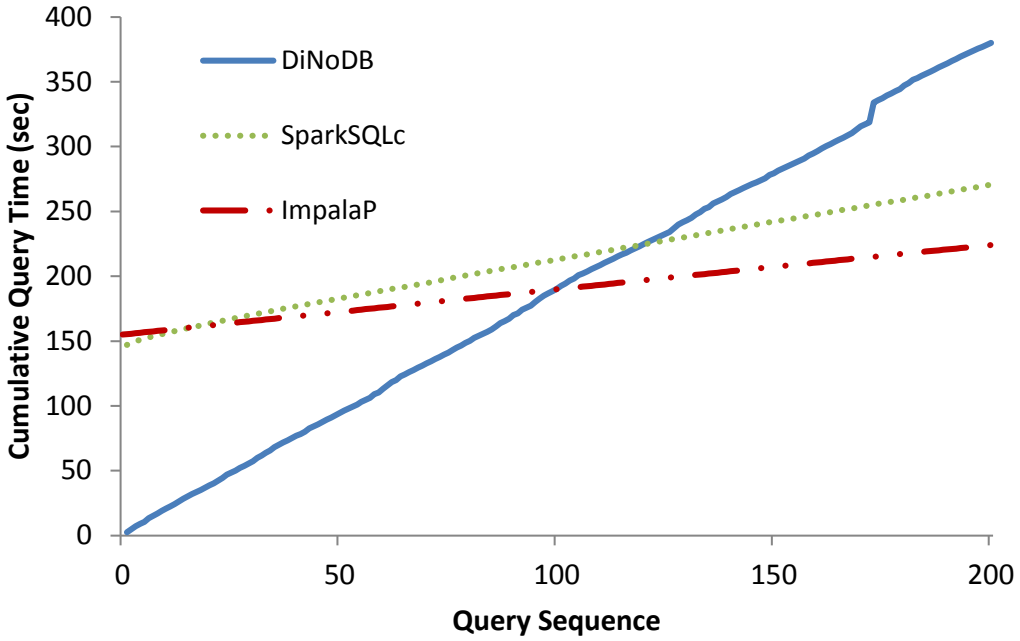


Figure 3.9 – DiNoDB is a sensible alternative for data exploration scenarios even for a long sequence of queries.

phase. Again, considering the aggregate query execution time for 10 queries, DiNoDB is more than 10 times faster than the second fastest system we study, ImpalaT.

3.6.2.3 Break-even point

In the previous experiments we show that for a relatively low number of queries, DiNoDB outperforms other distributed systems. In this experiment, we are interested in finding the break-even point, that is the number of queries DiNoDB can execute before its performance becomes equal or worse than the alternative systems. To this end, we use the same dataset and the same query patterns as in Section 3.6.2.1, for a sequence of 200 queries. We compare DiNoDB with SparkSQLc and ImpalaP.

The results, shown in Figure 3.9, indicate that for this workload, 100 queries represent the break-even point. If users execute less than 100 queries on temporary data, DiNoDB outperforms alternative approaches. For more than 100 queries, ImpalaP and SparkSQLc perform better than DiNoDB. In this case, the cost of converting the initial dataset to Parquet and RDD, respectively, is amortized. Clearly, a large number of queries on temporary output data as part of the illustrative exploratory use cases we consider in this work, is unlikely, making DiNoDB a sensible choice.

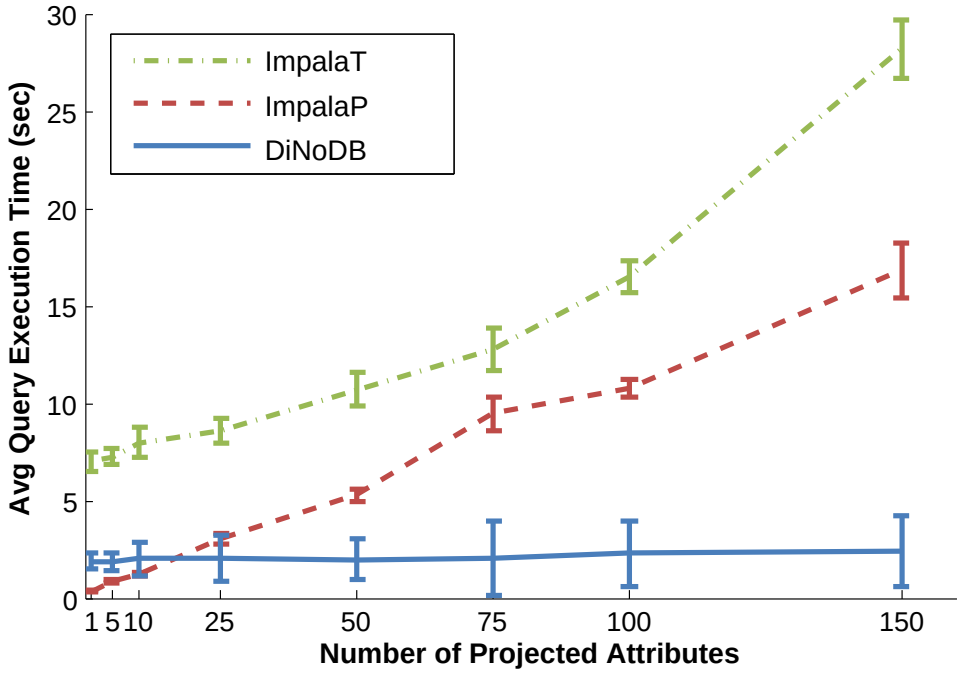


Figure 3.10 – Latency in function of the number of projected attributes.

3.6.2.4 Impact of data format

In the previous experiments we compared different data analytics systems assuming native, uncompressed, data format such as text-based CSV (or JSON). For Impala, in particular, we considered the transformation from text-based to the columnar Parquet format (ImpalaP) as the "loading" phase of ImpalaP. Although we expect our assumption of raw, text-based input to be reasonable for *temporary data* of various applications (with such temporary data being the focus of this chapter and DiNoDB), one may argue that the comparison to ImpalaP has not been entirely "fair" so far — as a user may write the output of a Hadoop job directly in Parquet format making it ready for ImpalaP and hence avoiding the need for loading/conversion phase.

Here, we compare the raw performance of ImpalaP (with data already in Parquet format) and DiNoDB (with data in text-based CSV format), with ImpalaT (using also text format) as a reference. We run a sequence of SELECT PROJECT queries, varying the number of projected attributes from 1 to 150 (i.e., all attributes, where the query boils down to a "`select * from table where ax < 100000`" query), while keeping the 0.1‰ selectivity and running 50 queries per experiment.

Figure 3.10 shows the average query latency with standard deviation against the number of projected attributes. This experiment shows that ImpalaP and DiNoDB consistently outperform ImpalaT, but that the comparison between ImpalaP and DiNoDB is not conclusive. ImpalaP outperforms DiNoDB for a relatively small number of projected

Figure 3.11 – Different sampling rate of positional map.

attributes (i.e., less than 30 attributes), whereas DiNoDB has the edge when the number of projected attributes is larger. We can also observe that the performance of DiNoDB is relatively constant with respect to the number of projected attributes, which makes it suitable for a wide range of ad-hoc queries. Besides pre-generated metadata, another advantage of DiNoDB comes from the feature inherited from PostgresRaw [41] called *selective parsing*: DiNoDB delays the binary transformation of projected attributes until it knows that the selectivity condition is satisfied for the given tuple. Since the selectivity of queries is very low, DiNoDB significantly reduces the CPU processing costs compared with ImpalaT and ImpalaP.

3.6.2.5 Impact of approximate positional maps

As we described in Section 3.4, we use a *uniform sampling* technique to keep PM metadata small. We investigate how different sampling rates of PM influence query execution cost. We use the same dataset and query sequence as in Section 3.6.2.1. We vary the sampling rate of PM from 1/10, 1/25, 1/50, 1/75, 0 (only containing the length of row) to no PM at all.

The results, shown in Figure 3.11, illustrate the trade-off between query execution time and space constraints. With more positions kept in the PM, DiNoDB achieves lower query execution times, but the metadata file of PM is larger. If a user is more sensitive to query execution time, she can choose a more aggressive sampling rate of PM. Otherwise, if cluster resources are limited, users can choose lower sampling rates, which result in

smaller PM files. Besides pre-generated PM metadata, DiNoDB also incrementally generates PM during query execution. Therefore, as shown in Figure 3.11, the difference in query execution time is more significant in the first few queries when PM is not detailed enough.

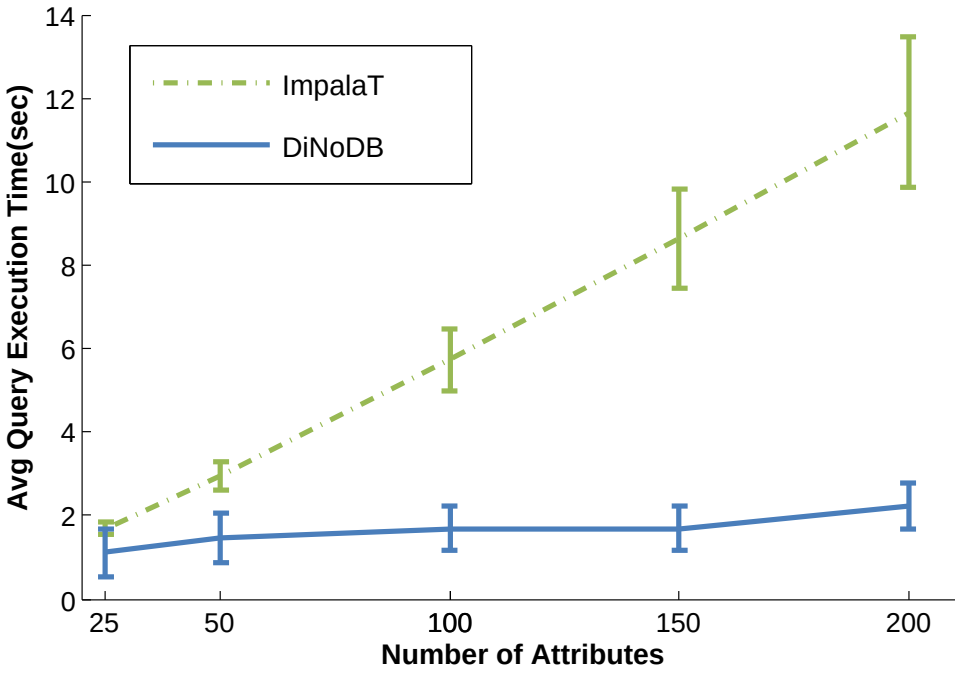
3.6.2.6 Scalability

In this section, we study the scalability of DiNoDB, compared to ImpalaT. Both systems perform a sequential scan operation (i.e., in case DiNoDB uses the positional map but not the vertical index).

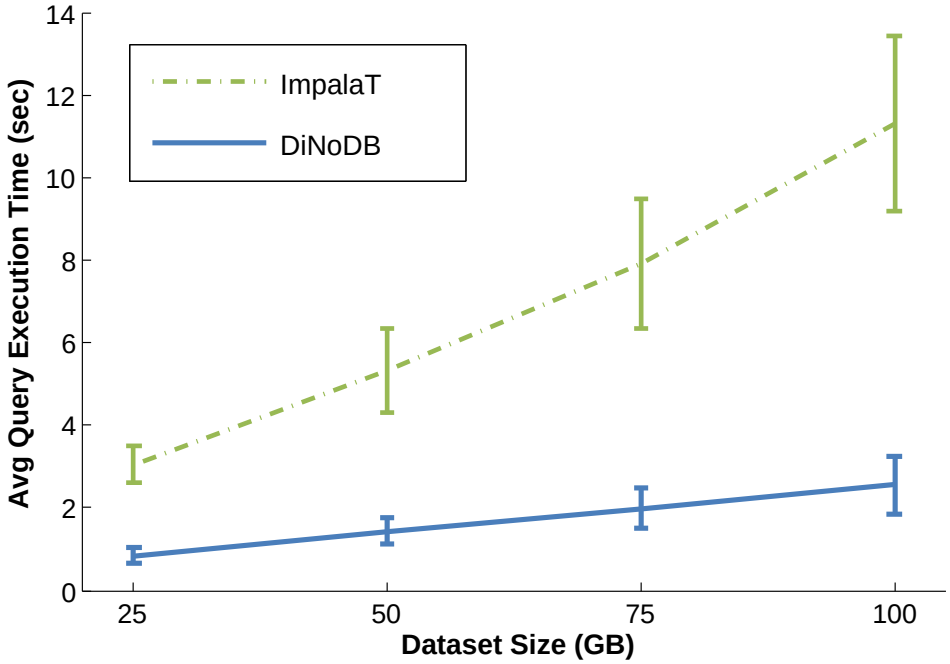
Scaling the number of attributes. In this experiment, we fix the number of records (rows) of the synthetic dataset, but we vary the number of attributes in the range between 25 and 200 attributes (dataset size from 12 GB to 96 GB). For each of these datasets, we execute a sequence of 50 SELECT PROJECT SQL queries in the same template as in Section 3.6.2.1. The average query execution time with standard deviation of DiNoDB and ImpalaT are shown in Figure 3.12(a). DiNoDB average query execution time is almost constant when the number of attributes grows thanks to PM metadata; clearly, PM metadata size grows with the number of attributes in the data. Instead, ImpalaT scales linearly with the number of attributes since it needs to parse every attribute (or byte) in a row: hence, it needs more CPU cycles when there are more attributes. ImpalaT has roughly the same performance as DiNoDB when there are 25 attributes. However, if the number of attributes exceeds 50, which is often the case the data analytics scenarios of our use cases, ImpalaT needs much more time to execute a query than DiNoDB.

Scaling the dataset size. In this experiment, we keep constant the number of attributes to 100, and vary the number of records (rows) in the synthetic dataset. We compare DiNoDB and ImpalaT when dataset size ranges between 25 and 100 GB, by executing 50 SELECT PROJECT SQL queries on each dataset. The average query execution time with standard deviation is shown in Figure 3.12(b). We observe that both ImpalaT and DiNoDB average query latency scale linearly with the dataset size. However, the slope for DiNoDB is less steep than that of ImpalaT.

Discussion. The DiNoDB I/O decorators that generate PMs are crucial for the performance of DiNoDB, especially for datasets with many attributes. Note, that our prototype implementation of DiNoDB does not enjoy many of the important optimizations that are available for Impala, such as efficient data type handlers, just-in-time compilation and so on.



(a) scaling the number of attributes



(b) scaling the dataset size

Figure 3.12 – DiNoDB vs. ImpalaT: Scalability.

3.6.3 Experiments with real life data

In this section, we give two examples of use cases we describe in Section 3.2 using real life data. One is a topic modeling use case and the other is a data exploration use case. In both examples, we show the query performance in different systems as well as the overhead of DiNoDB I/O decorators in the batch processing phase.

3.6.3.1 Experiment on machine learning

Here, we focus on a topic modeling use case, described in Section 3.2.1. We use a 40 GB dataset collected by Symantec[2], consisting of roughly 55 million emails (in JSON format) tagged as spam by their internal filtering mechanism. To better understand the features of these spam emails, data scientists in Symantec often try to discover topics that occur in these collections of emails. In this experiment, we first play the role of a data scientist involved in topic modeling, and use Apache Mahout (version 0.11), a scalable machine learning library for Hadoop MapReduce. The CVB algorithm is iterative, and thus consists of multiple Hadoop MapReduce jobs with many intermediate data outputs. Since users are usually only interested in the final output, we instruct our DiNoDB I/O decorators to generate metadata solely in the last stage, upon algorithm termination, in which the distribution of documents and topics is finalized.

DiNoDB I/O decorators overhead. First, we compare the Mahout topic modeling job to the one assembled with our DiNoDB I/O decorators, to study the overhead that it may impose on the preprocessing phase. In general, machine learning algorithms executed on large datasets take quite a long time to complete. Topic modeling is not an exception; in our experiments, we set the two most important parameters of CVB to 20 topics and 5 iterations[3]. We run both the Mahout topic modeling job with DiNoDB I/O decorators and without DiNoDB I/O decorators[4]. The resulting output file is a text file of roughly 23 GB, containing the probability matrix with 55 million rows and 21 columns. The metadata which is generated by our DiNoDB I/O decorators mechanism is about 880 MB (with 1/5 sampling rate for positional map).

The overall runtime of Mahout topic modeling job is approximately 5 hours and 40 minutes. In particular, we run the last stage both without DiNoDB I/O decorators and with DiNoDB I/O decorators 10 times to measure the overhead of DiNoDB I/O decorators.

[2]Symantec: www.symantec.com

[3]These parameters have been discussed with domain experts, who require a reasonable number of topics for manual inspection.

[4]Native Mahout outputs results of topic modeling in binary through *SequenceFileOutputFormat* class and needs command *vectordump* to transform data to text format. In contrast, in our experiment we let Mahout output results in text format directly by replacing SequenceFileOutputFormat class with *TextOutputFormat* class (without DiNoDB I/O decorators) or *DiNoDBTextOutputFormat* class (with DiNoDB I/O decorators)

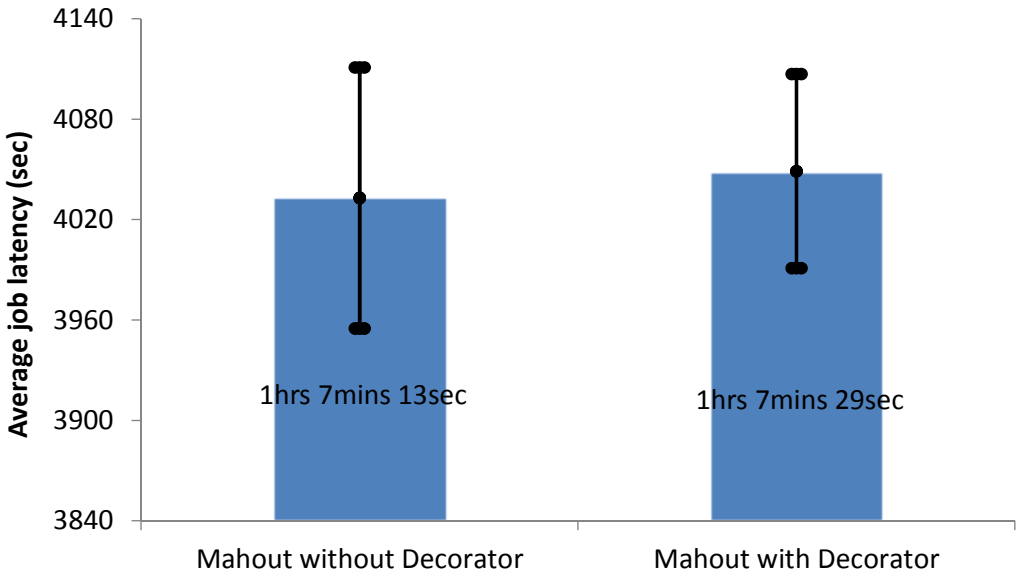


Figure 3.13 – The processing time of the last stage of topic modeling with Mahout.

The average processing time with standard deviation of the last stage is shown in Figure 3.13, which shows that the overhead of DiNoDB I/O decorators is about 16 seconds. This overhead, relative to the processing time of the last stage and the overall runtime of Mahout topic modeling job, demonstrates that the overhead introduced by our DiNoDB I/O decorators is essentially negligible.

Query performance. In this experiment, we query the output of the topic-modeling phase: the output data file is a "doc-topic" table, in which each row consists in a document identifier and the probability that such document belongs to each of the 20 topics. Hence, the output data file has one `INT` attribute (`docid`, which also serves as the key attribute) and 20 `FLOAT` attributes (probabilities). In order to understand if spam emails are well assigned to different topics, we would like to know which subset of emails have the highest probability to be in each topic. Therefore, the queries we execute with DiNoDB choose the top-10 spam emails per topic, sorted by probability measure: `select docid, p_topic_x from table order by p_topic_x desc limit 10`, where `docid` is document (email) id and `p_topic_x` is the probability of that document belonging to topic_x. Note that, since the queries are not based on the selectivity of the key attribute (`docid`), DiNoDB does not use VI, and solely relies on the PM file to improve performance.

The result of a 10-query sequence for each system is shown in Figure 3.14. For a fair comparison, we penalize DiNoDB by adding the 16 seconds overhead of DiNoDB I/O decorators (as seen in Figure 3.13) to the query execution time of DiNoDB, although we believe that users are much more sensitive to the latency in interactive analytics than the latency in batch processing. With the help of positional map file generated by the DiNoDB I/O decorators, DiNoDB still achieves one of the shortest execution times for

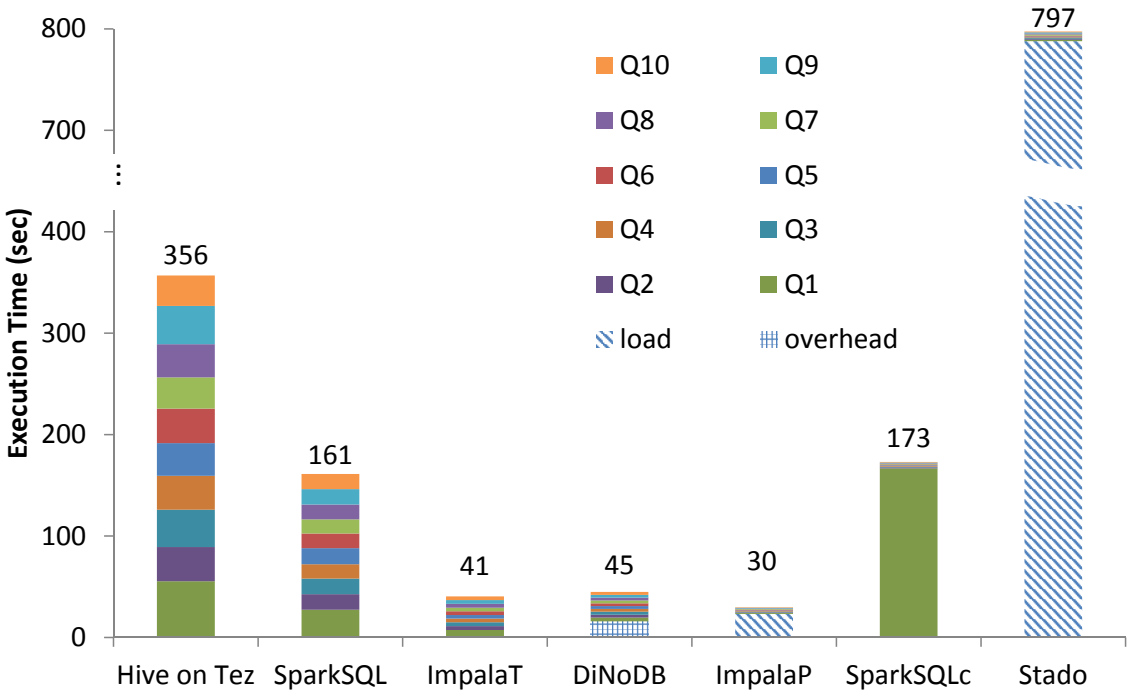


Figure 3.14 – Query execution time in machine learning use case (Symantec dataset).

this sequence of queries, being only slightly slower only from ImpalaP and ImpalaT even though the overhead from batch processing was taken into account. Notice that if the overhead of DiNoDB I/O decorators in the batch processing phase is not considered in the query execution time, DiNoDB achieves the shortest execution time. Like in Section 3.6.2 SparkSQLc, ImpalaP and Stado have very short single query execution cost, which is less than 1 second, but they all need to first load data. Note that, if we compare the loading time of SparkSQLc and ImpalaP with their loading time in section 3.6.2, we find that Impala has a better support for data type FLOAT than Spark SQL.

3.6.3.2 Experiment on data exploration

In this section, we focus on the data exploration use case, described in Section 3.2.2. We use a trace file which is the result of merging all the log files of Ubuntu One[5] servers for 30 days (773 GB of CSV text). Now, let's assume that a user is interested in knowing the features of files stored in Ubuntu One. To better analyze this trace, this user filters out unnecessary information like server's RPC logs and creates a new data file called "FileObject" in which original trace is reorganized based on files. So, this user chooses to write Hadoop programs to preprocess the trace file.

[5] http://en.wikipedia.org/wiki/Ubuntu_One

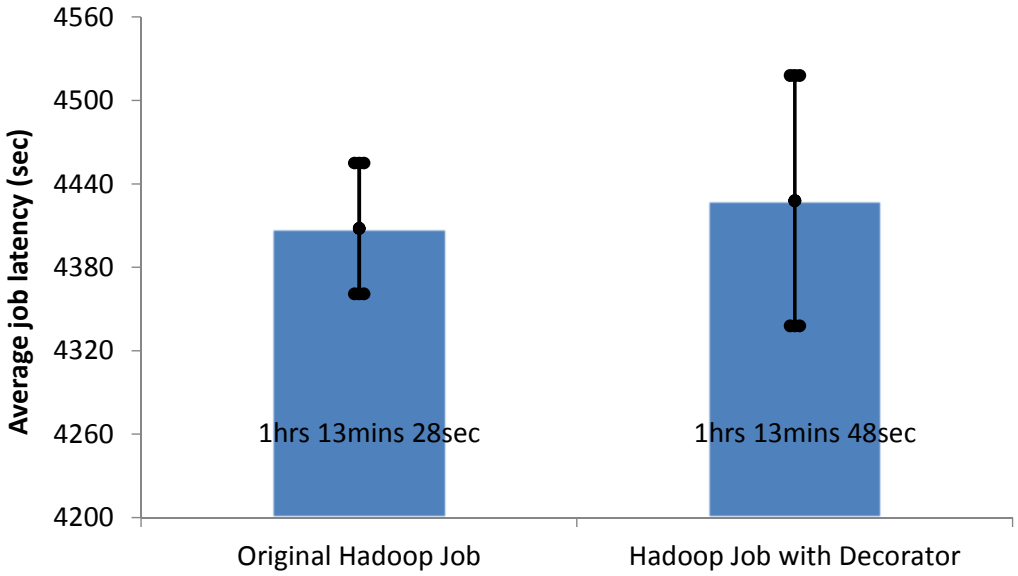


Figure 3.15 – The batch processing time in data exploration use case (Ubuntu One dataset)

DiNoDB I/O decorators overhead. Here, we compare the normal Hadoop pre-processing job to the one assembled with DiNoDB I/O decorators, to study the overhead brought by DiNoDB I/O decorators. In "FileObject" file, each row/record represents a file stored in Ubuntu One server. Each row has 26 attributes, giving detailed information, e.g., mime type, creation time, file size, etc. To produce this "FileObject" file, we run both the original Hadoop job and the Hadoop job with DiNoDB I/O decorators 10 times. The resulting output is about 40 GB in size, containing information of 137 million unique files. A 2.2 GB PM metadata file is generated by DiNoDB I/O decorators mechanism (with 1/10 sampling rate). As shown in Figure 3.15, the average processing time of original Hadoop job is 1 hour and 13 minutes and 28 seconds. With DiNoDB I/O decorators, the Hadoop job needs about 20 seconds extra time, which is 0.45% of the total batch processing cost. This overhead of DiNoDB I/O decorators is also negligible as in the machine learning use case.

Query performance. In this experiment, we compare the performance of the aforementioned systems when quering the result data file, "FileObject". The queries[6] we use in this experiment compute, for example, the number of distinct file extensions that users of the Ubuntu One service store, how many times the most popular file is downloaded, etc. The result of a 10-query sequence is shown in Figure 3.16. For a fair comparison the 20 seconds overhead of DiNoDB I/O decorators (as seen in Figure 3.15) is added to the query execution time of DiNoDB. We find that query execution time of SparkSQLc, ImpalaP and Stado is much longer than in the previous experiments. That's because the

[6] www.eurecom.fr/~tian/dinodb/ubuntuone.html provides details on the dataset schema and the queries we used in our experiments.

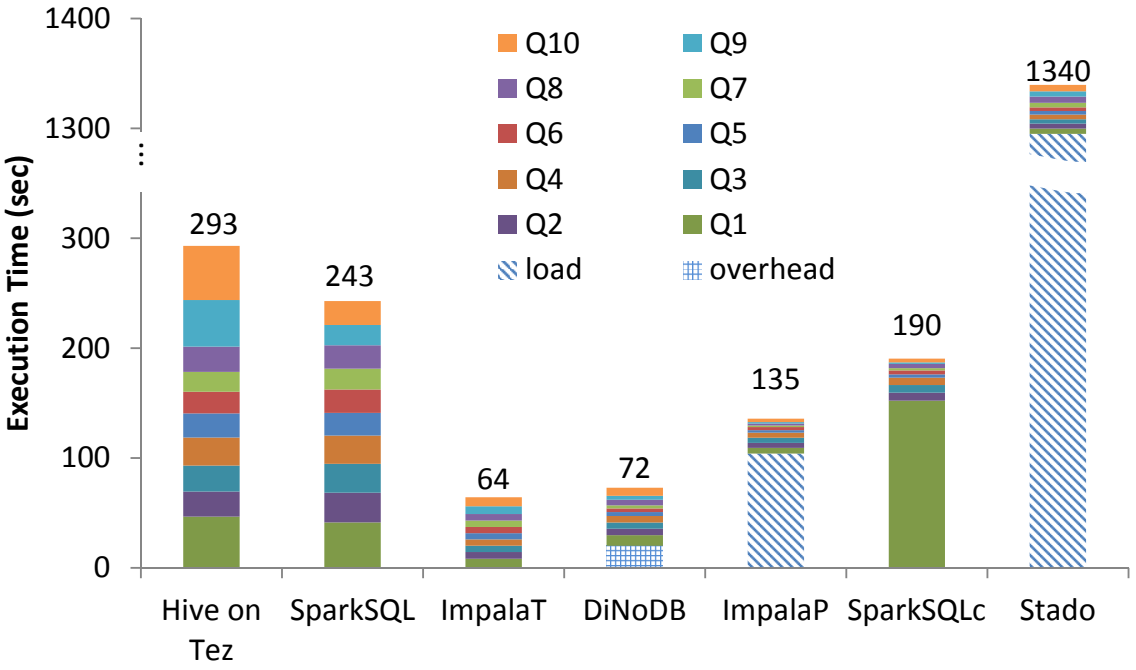


Figure 3.16 – Query execution time in data exploration use case (Ubuntu One dataset).

queries executed in this experiment are more complex, by selecting more attributes and using operators like group by and aggregation. DiNoDB outperforms alternative systems except ImpalaT when considering total query execution time including DiNoDB I/O decorators overhead.

We note that, in this experiment as well as the respective experiment in Section 3.6.3.1, ImpalaT query performance is on par with that of DiNoDB (without DiNoDB I/O decorators overhead). This is because when there are not many attributes (e.g., 20), the advantage of DiNoDB is not that obvious, as we demonstrated in Section 3.6.2.6.

3.6.4 Impala with DiNoDB I/O decorators

Metadata generated by DiNoDB I/O decorators in the batch processing phase can also be beneficial to other systems: they are not confined to be used in conjunction with DiNoDB. In this section, we present how Impala performs when we implant the DiNoDB I/O decorators in the workflow.

In this experiment, an Impala user operates on the output data generated by a batch processing phase on the Ubuntu One server logs described previously. Let's assume the user wants to compute the number of times that files of a particular type, which are stored on the Ubuntu One service, are downloaded during a given period of time. In this case, in addition to the "FileObject" output file, a "DownloadRecord" output file is also produced in the batch processing phase. In "DownloadRecord" file, each record

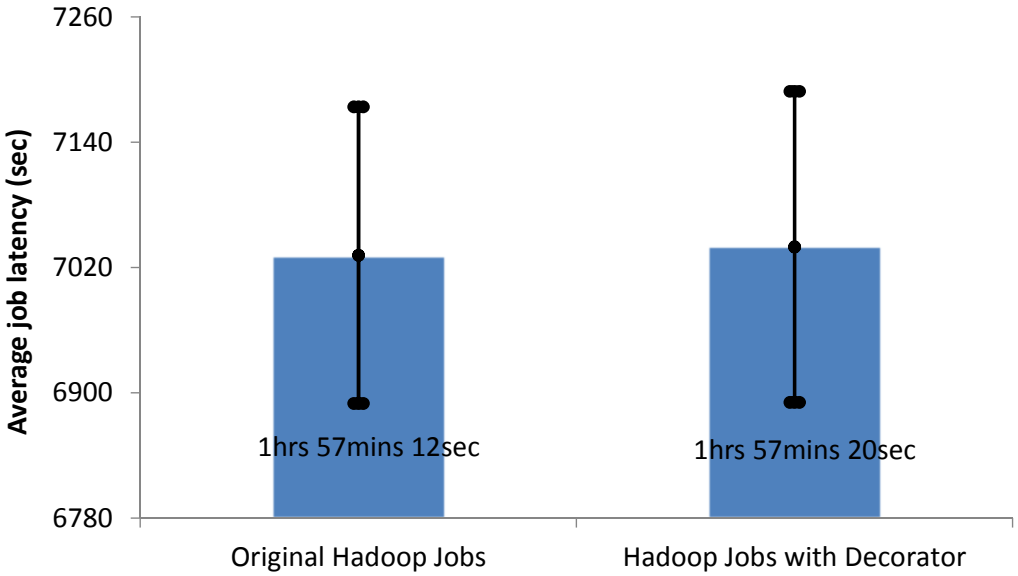


Figure 3.17 – DiNoDB I/O decorators overhead to generate statistics.

represents a download operation from Ubuntu One servers. Thus, our user instructs Impala to join two output data files. We run Hadoop jobs with and without DiNoDB I/O decorators 10 times to measure the overhead of DiNoDB I/O decorators when only statistics decorator is used. The average batch processing time with standard deviation is shown in Figure 3.17 which shows that, on average, statistics decorator only brings an extra 8 seconds overhead.

Next, we compare Impala in three situations: i) executing queries without statistics, ii) executing queries after generating statistics using the built-in command "Compute Statistics" and iii) executing queries with statistics generated by DiNoDB I/O decorators during the batch processing phase[7]. We execute 4 queries in each case and we show the results in Figure 3.18 where the overhead of statistics decorator is also included. In the first case, without statistics, Impala cannot choose an optimal query plan for the join operator, so the query execution time is the longest. In the second case, the query latency is quite short, but Impala needs to spend more than 1 minute to generate statistics before it can execute the queries. With DiNoDB I/O decorators, Impala achieves the lowest execution time even with the overhead from batch processing: this demonstrates the flexibility of our approach, and validates the design choice of piggybacking costly operations in the batch processing phase.

[7]Statistics metadata can be passed to Impala by injecting them into the Impala metastore

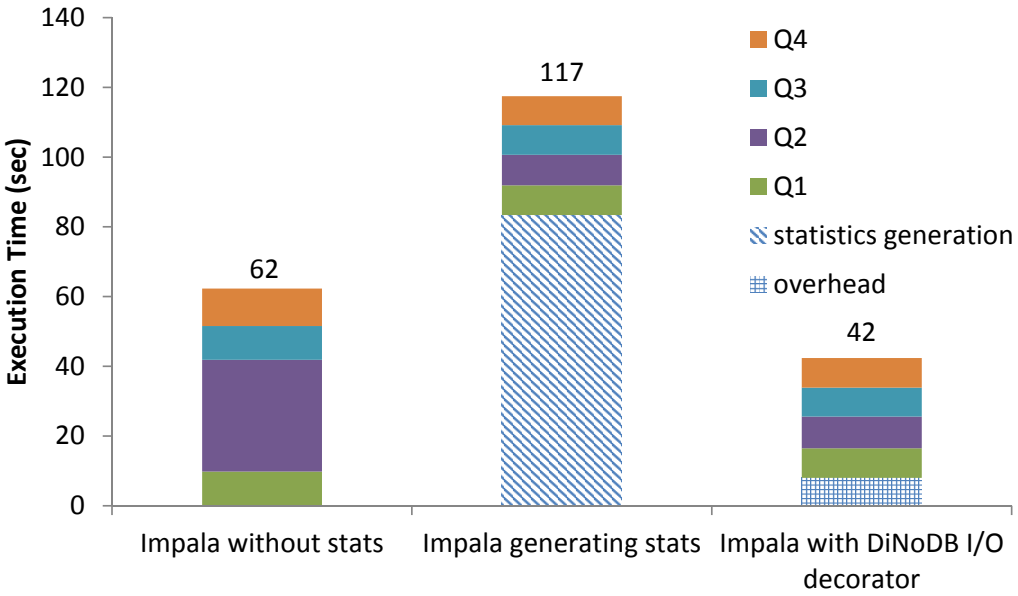


Figure 3.18 – DiNoDB I/O decorators can be beneficial to other systems (e.g., Impala).

3.7 Related work

Several research works and commercial products complement the batch processing nature of Hadoop/MapReduce [45,62] with systems to query large-scale data at interactive speed using a SQL-like interface. Examples of such systems include HadoopDB [40] and Vertica [61]. These systems require data to be loaded before queries can be executed: in workloads for which data-to-query time matters, for example due to the ephemeral nature of the data at hand, the overheads due to the load phase, crucially impact query performance. In [77] the authors propose the concept of "invisible loading" for HadoopDB as a technique to reduce the data-to-query time; with invisible loading, the loading to the underlying DBMS happens progressively and on demand during the first time we need to access the data. In contrast to such systems, DiNoDB avoids data loading and is tailored for querying raw data files leveraging metadata. Such files are built in DiNoDB with a lightweight piggybacking mechanism for workloads involving a preliminary data processing phase such as machine learning and data exploration use cases.

Shark [56] presents an alternative design: it relies on a novel distributed shared memory abstraction called Resilient Distributed Datasets (RDDs) [89] to perform most computations in memory while offering fine-grained fault tolerance. Shark builds on Hive [63] to translate SQL-like queries to execution plans running on the Spark system [69], hence marrying batch and interactive data analysis. Recently, Spark SQL [71] was announced as the Shark replacement in the Spark stack, as the new SQL engine for Spark designed from ground-up. The main characteristic of Spark SQL is that, just like Shark, it works with Spark's RDDs. Hence, both Spark SQL and Shark require a variant of data loading,

to transform the raw HDFS data file and bring it into the RDD representation, in order to benefit fully from the Spark's in-memory low-latency processing. In contrast, DiNoDB achieves low latency while working on raw files, entirely avoiding data loading.

Impala [36] is a state-of-the-art massively parallel processing (MPP) SQL query engine that runs in Hadoop. As such, Impala is probably the closest system to DiNoDB. However, while co-located with Hadoop, Impala does not leverage the possible synergy between batch processing of Hadoop and analytics power of a MPP SQL query engine. In this chapter, we exactly propose such a synergy, and build DiNoDB to validate our approach. Namely, in DiNoDB batch processing generates metadata (e.g., positional maps and vertical indexes) that helps expedite SQL analytical queries. In this work, we demonstrate that the synergy between batch processing and query engines is beneficial for the analytical phase.

SCANRAW [90, 91] is proposed as a novel database physical operator, which loads data speculatively using available I/O bandwidth. PostgresRaw [41] is a centralized DBMS that avoids data loading and transformation prior to queries. DiNoDB leverages PostgresRaw as a building block to obtain DiNoDB nodes, which are to be seen as enhanced version of PostgresRaw (see Section 3.5.2 for detailed comparison between DiNoDB nodes and PostgresRaw). A critical difference between DiNoDB and these works is that, DiNoDB is a distributed, massively parallel system for large-scale data analytics integrated with Hadoop batch processing framework, whereas PostgresRaw and SCANRAW are centralized database solutions. Dremel [93] and GLADE [92] adopt multi-level aggregation to overcome single node bottleneck. This approach could be applied to DiNoDB to improve its scalability in future work.

Finally, DiNoDB shares some similarities with several research work that focuses on improving Hadoop performance. For example, Hadoop++ [46] modifies the data format to include a Trojan Index so that it can avoid full file sequential scan. Furthermore, CoHadoop [48] co-locates related data files in the same set of nodes so that a future join task can be done locally without transferring data in network. However, these systems require users to write specific Hadoop program, which could be complex and hard. Therefore, neither Hadoop++ nor CoHadoop are suitable for interactive raw data analytics, like DiNoDB is.

To conclude, we compare our current version of DiNoDB with a preliminary version, presented in [38]. The preliminary version of DiNoDB used HadoopDB to orchestrate DiNoDB nodes, which suffered from the inherent overhead of the HadoopDB framework when it came to interactive queries. Therefore, we replaced HadoopDB with a massively parallel architecture, but had to address fault tolerance of such an architecture. Moreover, our current version of DiNoDB introduces DiNoDB I/O decorators, which are easy to implement and help couple batch layer and serving layer more seamlessly. As a result,

our current version of DiNoDB outperforms state-of-the-art analytics systems such as Spark SQL and Impala, for temporary data.

3.8 Conclusion

Parallel data processing systems such as Hadoop MapReduce have received increasing attention, for their promise to analyze any amount or kind of data. However, with such systems, users had to move and load the data produced in the batch processing phase into a fast relational database, to achieve interactive-speed data manipulation. The specter of "leaving something important behind" related to data movement and adaptation has led academics and the industry to designing new systems that would expose a standard, interactive way to manipulate data, while being fully integrated in a growing ecosystem of data processing tools.

In this work, we presented the architecture of DiNoDB, a distributed system tuned for interactive-speed queries on temporary data files generated by large-scale batch-processing frameworks. As shown by our extensive experimental evaluation, for the use-cases DiNoDB targets – ad hoc queries on a narrow processing window, our system outperforms current SQL-on-Hadoop solutions. DiNoDB uses a decorator mechanism that enhances the standard Hadoop I/O API and piggybacks the creation of auxiliary metadata required for interactive-speed query performance. In addition, DiNoDB I/O decorators seamlessly integrate with existing frameworks and distributed storage systems.

Our experimental evaluation, that we do on both synthetic and real-world datasets, highlights the key benefits of DiNoDB in a number of prominent use cases, making it suitable for a wide range of ad-hoc analytical workloads.

Chapter 4

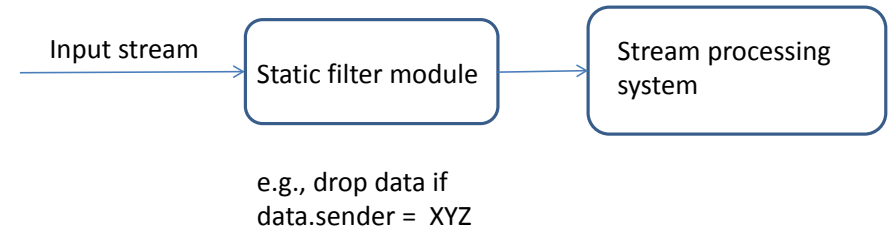
Bleach: a Distributed Stream Data Cleaning System

4.1 Introduction

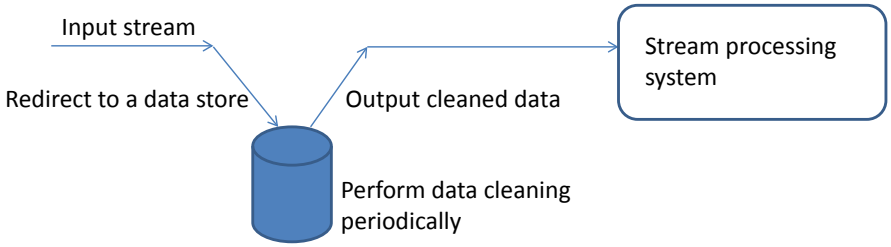
Today, we live in a world where decisions are often based on analytics applications that process continuous streams of data. Typically, data streams are combined and summarized to obtain a succinct representation thereof: analytics applications rely on such representations to make predictions, and to create reports, dashboards and visualizations [119, 122, 123]. All these applications expect the data, and their representation, to meet certain quality criteria. Data quality issues interfere with these representations and distort the data, leading to misleading analysis outcomes and potentially bad decisions.

As such, a range of data cleaning techniques were proposed recently [99, 127, 128]. However, most of them focus on "batch" data cleaning, by processing static data stored in data warehouses, which are quite time-consuming. They neglect the important class of streaming data. In this chapter, we address this gap and focus on *stream data cleaning*. The challenge in stream cleaning is that it requires both *real-time* guarantees as well as high *accuracy*, requirements that are often at odds.

A naïve approach to stream data cleaning, as shown in Figure 4.1(a), is to include simple static filters to process dirty records, but its cleaning ability is quite limited. Another naïve approach, as shown in Figure 4.1(b), could simply extend existing batch techniques, by buffering data records in a temporary data store and cleaning it periodically before feeding it into downstream components. Although likely to achieve high accuracy, such a method clearly violates real-time requirements of streaming applications. The problem is exacerbated by the volume of data cleaning systems need to process, which prohibits



(a) Use simple data cleaning way



(b) Use batch data cleaning way

Figure 4.1 – Existing stream data cleaning

centralized solutions. Therefore, our goal is to design a *distributed stream data cleaning system*, which achieves efficient and accurate cleaning in real-time.

In this chapter, we focus on qualitative data cleaning, whereby a set of domain-specific rules define how data should be cleaned: in particular, we consider functional dependencies (FDs) and conditional functional dependencies (CFDs). Our system, called Bleach, proceeds in two phases: *violation detection*, to find rule violations, and *violation repair*, to repair data based on such violations. Bleach relies on efficient, compact and distributed data structures to maintain the necessary state (e.g., summaries of past data) to repair data, using an incremental equivalence class algorithm.

We further address the complications due to the long-term and dynamic nature of data streams: the definition of dirty data could change to follow such dynamics. Bleach supports dynamic rules, which can be added and deleted without requiring idle time. Additionally, Bleach implements a sliding window operation that trades modest additional storage requirements to temporarily store cumulative statistics, for increasing cleaning accuracy.

Our experimental performance evaluation of Bleach is two-fold. First, we study the performance, in terms of throughput, latency and accuracy, of our prototype and focus on the impact of its parameters. Then we compare Bleach to an alternative baseline system, which we implement using a micro-batch streaming architecture. Our results indicate the benefits of a system like Bleach, which hold even with rule dynamics. Despite extensive work on rule-based data cleaning [96, 99, 101, 102, 111, 113, 114, 118], we are not aware of any other stream data cleaning system.

4.2 Preliminaries

Next, we introduce basic notation we use throughout the chapter, then we define the problem statement we consider.

4.2.1 Background and Definitions

In this thesis, we assume that a stream data cleaning system ingests a data stream and outputs a cleaned data stream. We consider an input data stream instance D_{in} with schema $S(A_1, A_2, ..., A_m)$ where A_j is an attribute in schema S. We assume the existence of unique tuple identifiers for every tuple in D_{in}: thus given a tuple t_i, $id(t_i)$ is the identifier of t_i. In general we define a function $id(e)$ which returns the identifier (ID) of e where e can be any element. A list of IDs $[id(e_1), id(e_2), ..., id(e_n)]$ is expressed as $id(e_1, e_2, ..., e_n)$ for brevity. The output data stream instance D_{out} complies with schema S and has the same tuple identifiers as in D_{in}, i.e., with no tuple loss or duplication. The basic unit, a *cell* $c_{i,j}$, is the concatenation of a tuple id, an attribute and the projection of the tuple on the attribute: $c_{i,j} = (id(t_i), A_j, t_i(A_j))$. Note that $t_i(A_j)$ is the value of $c_{i,j}$, which can also be expressed as $v(c_{i,j})$. Sometimes, we may simply express $c_{i,j}$ as c_i when the cell attribute is not relevant to the discussion. In our work, when we point at a specific tuple t_i, we also refer to this tuple as the *current* tuple. Tuples appearing earlier than t_i in the data stream are referred to as *earlier* tuples and those appearing after t_i are referred to as *later* tuples. To perform data cleaning, a set of rules $\Sigma = [r_1, ..., r_n]$ are defined. Each rule has a unique rule identifier $id(r_k)$.

A FD rule, r_k, is represented as $(X \rightarrow A)$, in which $X \subseteq S$ and $A \in S$. X and A are respectively referred to as a set of left-hand side (LHS) attributes and right-hand side (RHS) attribute: $LHS(r_k) = X$, $RHS(r_k) = A$. [1] A data stream instance D satisfies r_k , denoted by $D \models r_k$, when for every pair of tuples t_1 and t_2, if $t_1(B) = t_2(B)$ for all $B \in X$, then $t_1(A) = t_2(A)$. In other words, if there exist any two tuples, t_1 and t_2, that have the same values for LHS attributes X, but different values for RHS attribute A, then there must be some errors in t_1 or t_2. Cells of LHS (RHS) attributes are also referred to as LHS (RHS) cells.

As an extension of FD rules, a CFD rule, r_m, is represented by $(X \rightarrow A, cond(Y))$, in which $cond(Y)$ is a boolean function on a set of attributes Y where $Y \subseteq S$. Y is referred to as a set of conditional attributes. For a data stream instance D, if there exists a pair of tuples t_1 and t_2 satisfying condition $cond(t_1(Y)) = cond(t_2(Y)) = true$ where $t_1(B) = t_2(B)$ for all $B \in X$ but $t_1(A) \neq t_2(A)$, then we say t_1 and t_2 violate against r_m. If there is no such pairs of tuples violating against r_m, then the data stream instance

[1] When the rule is clear in the context, we omit r_k so that $LHS = X$, $RHS = A$.

D satisfies r_m, expressed as $D \models r_m$. Note that a FD rule can be seen as a special case of CFD rule where $cond(Y)$ is always true and Y is $\emptyset$.

If D satisfies a set of rules Σ, denoted $D \models \Sigma$, then $D \models r_k$ for $\forall r_k \in \Sigma$. If D does not satisfy Σ, D is a dirty data stream instance. We refer to an attribute as an *intersecting* attribute if it is involved in multiple rules.

4.2.2 Challenges and Goals

An ideal stream data cleaning system should accept a dirty input stream D_{in} and output a clean stream D_{out}, in which all *rule violations* in D_{in} are repaired ($D_{out} \models \Sigma$). However, this is not possible in reality due to:

- **Real-time constraint**: As the data cleaning is incremental, the cleaning decision for a tuple (repair or not repair) can only be made based on itself and earlier tuples in the data stream, which is different from data cleaning in data warehouses where the entire dataset is available. In other words, if a dirty tuple only has violations with later tuples in the data stream, it can not be cleaned. A late update for a tuple in the output data stream cannot be accepted.
- **Dynamic rules**: In a stream data cleaning system, the rule set is not static. A new rule may be added or an obsolete rule may be deleted at any time. A processed data tuple can not be cleaned again with an updated rule set. Reprocessing the whole data stream whenever the rule set is updated is not realistic.
- **Unbounded data**: A data stream produces an unbounded amount of data, that cannot be stored completely. Thus, stream data cleaning can not afford to perform cleaning on the full data history. Namely, if a dirty tuple only has violations with tuples that appear much earlier in the data stream, it is likely that such a tuple will not be cleaned.

Consider the example in Figure 4.2, which is a data stream of on-line shopping transactions. Each tuple represents a purchase record, which contains a purchased item ($item$), the category of that item ($category$), a client identifier ($clientid$), the city of the client ($city$) and the zip code of that city ($zipcode$). In the example, we show an extract of five data tuples of the data stream, from t_1 to t_5.

Now, assume we are given two FD rules and one CFD rule stating how a clean data stream should look like: (r_1) the same items can only belong to the same category; (r_2) two records with the same $clientid$ must have the same city; (r_3) two records with the same non-null zip code must have the same city:

(r_1) $item \rightarrow category$

(r_2) $clientid \rightarrow city$

time

	item	category	clientid	city	zipcode
	...	...	...	...	...
t_1	MacBook	computer	11111	France	75001
t_2	bike	sports	33333	Lyon	null
t_3	Interstellar	movies	22222	Paris	75001
t_4	bike	toys	44444	Nice	06000
t_5	Titanic	movies	11111	Paris	null
	...	...	...	...	...

Figure 4.2 – Illustrative example of a data stream consisting of on-line transactions.

(r_3) $zipcode \rightarrow city, zipcode \neq null$

In our example, there are three violations of rules r_1, r_2 and/or r_3: ($v1$) t_1 and t_3 have the same non-null zip code ($t_1(zipcode) = t_2(zipcode) \neq null$) but different city names ($t_1(city) \neq t_2(city)$); ($v2$) t_2 claims bikes belong to category sports while t_4 classifies bikes as toys ($t_2(item) = t_4(item), t_2(category) \neq t_4(category)$); and ($v3$) t_1 and t_5 have the same $clientid$ but different city names ($t_1(clientid) = t_5(clientid), t_1(city) \neq t_5(city)$).

Note that when a stream data cleaning system receives tuple t_1, no violation can be detected as in our example t_1 only has violations with later tuples t_3 and t_5. Thus, no modification can be made to t_1. Furthermore, delaying the cleaning process for t_1 is not a feasible option, not only because of real-time constraints, but also because it is difficult to predict for how long this tuple should be buffered for it to be cleaned. Therefore, stream data cleaning must be incremental: whenever a new piece of data arrives, the data cleaning process starts immediately. Although performing incremental violation detection seems straightforward, incremental violation repair is much more complex to achieve. Coming back to the example in Figure 4.2, assume that the stream cleaning system receives tuple t_5 and successfully detects the violation v_3 between t_5 and t_1. Such detection is not sufficient to make the correct repair decision, as the tuple t_1 also conflicts with another tuple, t_3. An incremental repair in stream data cleaning system should also take the violations among earlier tuples into account.

To account for the intricacies of the violation repair process, we use the concept of *violation graph* [99]. A violation graph is a data structure containing the detected violations, in which each node represents a cell. If some violations share a common cell, they will be grouped into a single *subgraph*. Therefore, the violation graph is partitioned into smaller independent subgraphs. A single cell can only be in one subgraph. If two subgraphs share a common cell, they need to merge. The repair decision of a tuple is only

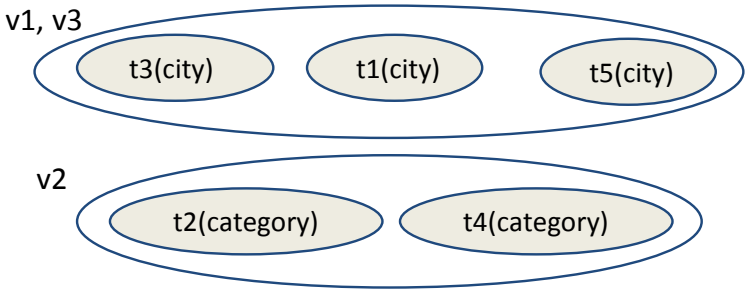


Figure 4.3 – An example of a violation graph, derived from our running example.

relevant to the subgraphs in which its cells are involved. A violation graph for our example can be seen in Figure 4.3. Given this violation graph, to make the repair decision for tuple t_5, the cleaning system can only rely on the upper subgraph which consists of violation v_1 and v_3 with the common cell $t_1(city)$. We now give our problem statement as following.

Problem statement: Given an unbounded data stream with an associated schema[2] and a *dynamic* set of rules, how can we design an *incremental* and *real-time* data cleaning system, including violation detection and violation repair mechanisms, using bounded computing and storage resources, to output a cleaned data stream?

In the following sections, we overview the Bleach architecture and provide details about its components. As shown in Figure 4.15, the input data stream first enters the **detect module** (Sections 4.3), which reveals violations against defined rules. The intermediate data stream is enriched with violation information, which the **repair module** (Section 4.4) uses to make repair decisions. Finally, the system outputs a cleaned data stream. The rule controller module, is discussed in Section 4.5. We discuss the windowing operation in Section 4.6. In Section 4.7, we discuss the dependency relations between rules. Section 4.8 presents our experimental results. Section 4.9 overviews related work and Section 4.10 concludes.

4.3 Violation Detection

The violation detection module aims at finding input tuples that violate rules. To do so, it stores the tuples in-memory, in an efficient and compact data structure that we call the *data history*. Input tuples are thus compared to those in the data history to detect violations. Figure 4.5 illustrates the internals of the detect module: it consists of an ingress router, an egress router and multiple detect workers (DW). Bleach maps

[2]Note that although we restrict the data stream to have a fixed schema in this work, it is easy to extend our work to support a dynamic schema.

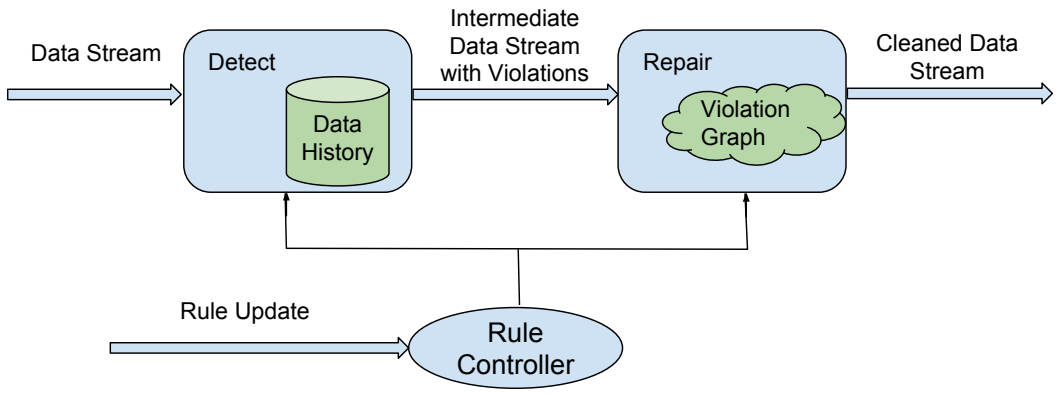


Figure 4.4 – Stream data cleaning Overview

violation rules to such DWs: each worker is in charge of finding violations for a specific rule.

4.3.1 The Ingress Router

The role of the ingress router is to partition and distribute incoming tuples to DWs. As discussed in Section 4.2, only a subset of the attributes of an input tuple are relevant when verifying data validity against a given rule. For example, a FD rule only requires its LHS and RHS attributes to be verified, ignoring the rest of the input tuple attributes.

Therefore, when the ingress router receives an input tuple, it partitions the tuple based on the current rule set, and only sends the relevant information to each DW in charge of each specific rule. As such, an input tuple is broken into multiple sub-tuples, which all share the same identifier of the corresponding input tuple. Note that some attributes of an input tuple might be required by multiple rules: in this case, sub-tuples will contain redundant information, allowing each DW to work independently. An example of tuple partitioning can be found in Figure 4.5, where we reuse the input data schema and the rules from Section 4.2.

4.3.2 The Detect Worker

Each DW is assigned a rule, and receives the relevant sub-tuples stemming from the input stream. For each sub-tuple, a DW performs a lookup operation in the data history, and emits a message to downstream components when a rule violation is detected.

To achieve efficiency and performance, lookup operations need to be fast, and the intermediate data stream should avoid redundant information. Next, we describe how the data history is represented and materialized in memory; then, we describe the output messages a DW generates, and finally outline the DW algorithm.

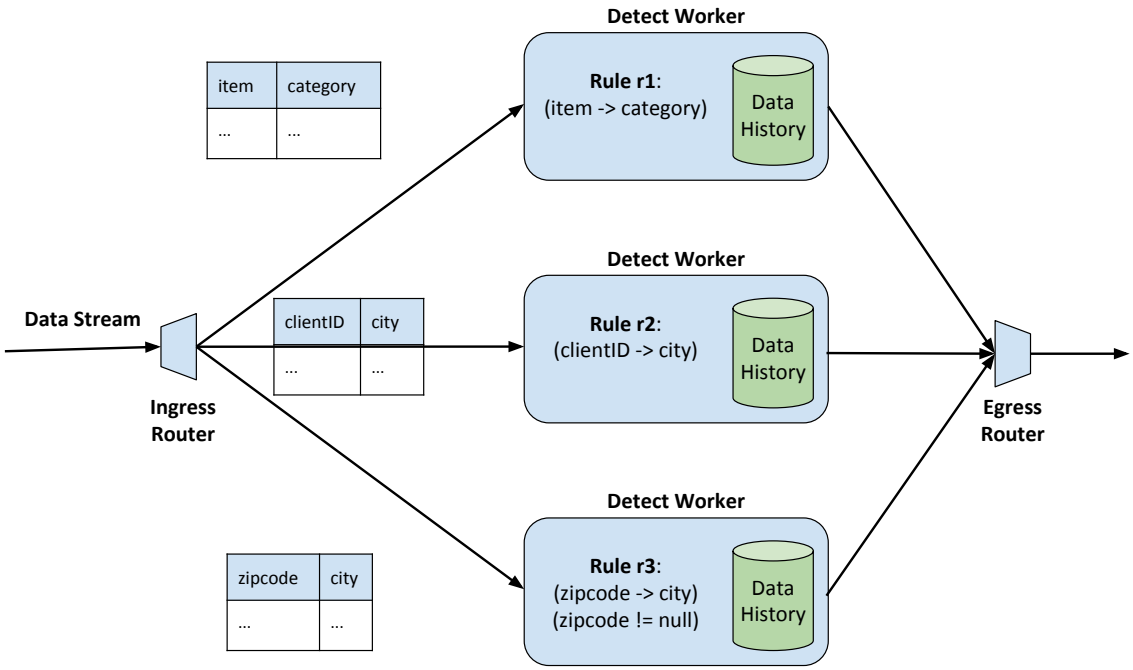


Figure 4.5 – The detect module

Data history representation. A DW accumulates relevant input sub-tuples in a compact data structure that enables an efficient lookup process, which makes it similar to a traditional indexing mechanism.

The structure[3] of the data history is illustrated in Figure 4.6. First, to speed-up the lookup process, sub-tuples are grouped by the value of the LHS attribute used by a given rule: we call such group a *cell group* (CG). Thus, a CG stores all RHS cells whose sub-tuples share the same LHS value. The identifier of a cell group cg_l is the combination of the rule assigned to the DW, and the value of LHS attributes, expressed as $id(cg_l) = (id(r_k), t(LHS))$ where r_k is the rule assigned to the DW.

Next, to achieve a compact data representation, all cells in a CG sharing the same RHS value are grouped into a *super cell* (SC): $sc_m = [c_{1,j}, c_{2,j}, ..., c_{n,j}]$. From Section 4.2, recall that a cell is made of a tuple ID, an attribute and a value: $(id(t_i), A_j, t_i(A_j))$. Therefore, a super cell can be *compressed* as a list of tuple IDs, an attribute and their common value: $sc_m = (id(t_1, t_2, ..., t_n), A_j, t(A_j))$ where $t(A_j) = t_1(A_j) = ... = t_n(A_j)$. Hence, within an individual DW, sub-tuples whose cells are compressed in the same *sc* are equivalent, as they have the same LHS attributes value (the identity of the cell group) and the same RHS attribute value (the value of super cell). A cell group cg_l now can be expressed as: $cg_l = ((id(r_k), t(LHS)), [sc_1, sc_2, ...])$ including an identifier and a list of super cells.

[3]The techniques we use are similar to the notion of *partitions* and *compression* introduced in Nadeef [101].

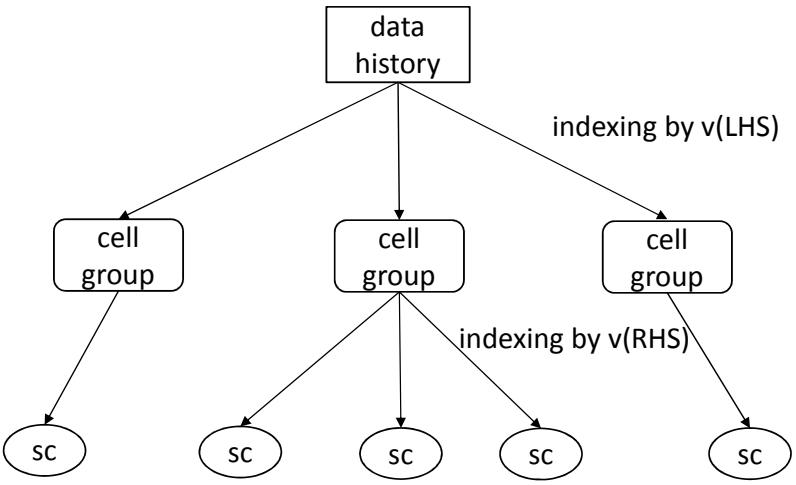


Figure 4.6 – The structure of the data history in a detect worker

In summary, the lookup process for a given input sub-tuple is as follows. Cell groups are stored in a hash-map using their identifiers as keys: therefore the DW first finds the CG corresponding to the current sub-tuple. Cells in the corresponding CG are the only cells that might be in conflict with the current cell. Overall, the complexity of the lookup process for a sub-tuple is $O(1)$.

Violation messages. DWs generate an intermediate data stream of *violation messages*, which help downstream components to eventually repair input tuples. The goal of the DW is to generate as few messages as possible, while allowing effective data repair.

When the lookup process reveals the current tuple does not violate a rule, DWs emit a non-violation message (msg_{nvio}). Instead, when a violation is detected, a DW constructs a message with all the necessary information to repair it, including: the ID of the cell group corresponding to the current tuple and the RHS cells of the current and earlier tuples in data history: $msg_{vio} = (id(cg_l), c_{cur}, c_{old})$.

Now, to reduce the number of violation messages, the DW can use a super cell in place of a single cell (c_{old}) in conflict with the current tuple. In addition, recall that a single CG can contain multiple super cells, thus possibly requiring multiple messages for each group. However, we observe that two cells in the same CG must also conflict with each other, as long as their values are different. Since the data repair module in Bleach is stateful, it is safe to omit some violation messages.

Algorithm details. Next, we present the DW violation algorithm details, as illustrated in Algorithm 4. The algorithm starts by treating FD rules as a special case of CFD rules (line 1). Then, when a DW receives a sub-tuple t_i satisfying the rule condition (line 2), it performs a lookup in the data history to check if the corresponding cell group cg_l exists (line 3). If yes, it determines the number of SC contained in the cg_l (line 4). If there is only one SC sc_{old}, violation detection works as follows. If the RHS cell of the current sub-tuple,

Algorithm 4 Violation Detection

```
1:  given rule r = (X → A_j, cond(Y))
2:  procedure RECEIVE(sub-tuple t_i)                    ▷ cond(t_i(Y)) = true
3:      if ∃id(cg_l) = (id(r), t_i(X)) then
4:          if |cg_l| = 1 then                          ▷ cg_l contains sc_old
5:              if v(sc_old) = t_i(A_j) then
6:                  Emit msg_nvio
7:              else
8:                  Emit msg_vio (id(cg_l), c_cur, sc_old)
9:              end if
10:         else
11:             Emit msg_vio (id(cg_l), c_cur, null)
12:         end if
13:     else
14:         Create cg_l                                 ▷ Create a new cell group
15:         Emit msg_nvio
16:     end if
17:     Add c_cur to cg_l
18: end procedure
```

c_{cur}, has the same value as sc_{old}, it emits a non-violation message (line 5-6). Otherwise, a violation has been detected: the DW emits a *complete* violation message, containing both the current cell and the old cell (line 8). If the CG contains more than one SC, the DW emits a single *append-only* violation message, which only contains the cell of the current sub-tuple (line 11). Such compact messages omit the SC from the data history, since they must be contained in earlier violation messages. Finally, if the lookup procedure (line 3) fails, the DW creates a new cell group and emits a non-violation message (line 14-15). At this point, the current cell c_{cur} is added to the corresponding group cg_l (line 17), either in an existing SC, or as a new distinct cell. It is worth noticing that, following Algorithm 4, a DW emits a single message for each input sub-tuple, no matter how many tuples in the data history it conflicts with.

4.3.3 The Egress Router

The egress router gathers (violation or non-violation) messages for a given data tuple, as received from all DWs, and sends them downstream to the repair module.

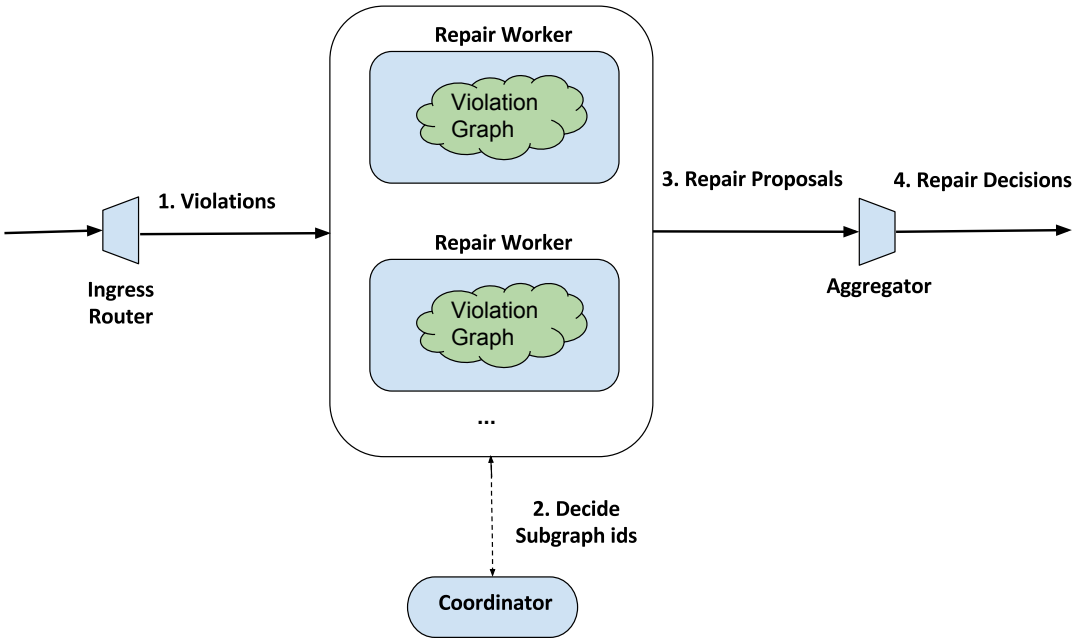


Figure 4.7 – The repair module

4.4 Violation Repair

The goal of this module is to take the repair decisions for dirty data tuples, based on an intermediate stream of violation messages generated by the detect module. To achieve this, Bleach uses a data structure called *violation graph*. Violation messages contribute to the creation and dynamics of the violation graph, which essentially groups those cells that, together, are used to perform data repair.

Figure 4.7 sketches the internals of the repair module: it consists of an ingress router, the repair workers (RW), and an aggregation component that emits clean data. An additional component, called the coordinator, steers violation graph management, with the contribution of RWs.

4.4.1 The Ingress Router

The ingress router broadcasts all incoming violation messages to all RWs. As opposed to its counterpart in the detect module, it does not perform data partitioning. Although each RW receives all violation messages, a cell in a violation message will only be stored in one RW with the goal of creating and maintaining the violation graph.

4.4.2 The Repair Worker

Next, we describe the operation of a RW. First, we focus on the violation graph and the data repair algorithm. Then, we move to the key challenge that RWs address, that is how to maintain a *distributed* violation graph. As such, we focus on graph partitioning and maintenance. Due to violation graph dynamics, coordination issues might arise in a distributed setting: such problems are addressed by the coordinator component.

The repair algorithm. Current data repair algorithms use a violation graph to repair dirty data based on user-defined rules. A violation graph is a succinct representation of cells (both current and historical) that are in conflict according to some rules. A violation graph is composed of subgraphs. As incoming data streams in, the violation graph evolves: specifically, its subgraphs might merge or split, depending on the contents of violation messages.

Using the violation graph, several algorithms can perform data cleaning, such as the equivalence class algorithm [125] or the holistic data cleaning algorithm [102]. Currently, Bleach uses an *incremental* version of the equivalence class algorithm, that supports streaming input data, although alternative approaches can be easily plugged in our system. The idea of the equivalence class algorithm is to group all elements which should be equivalent to each other, and then to decide an unique value for elements in the same group. Thus, a subgraph in the violation graph can be interpreted as an equivalence class, in which all cells are supposed to have the same value.

The Bleach violation graph is built using violation messages output by the detect module. We say that a subgraph sg intersects with a violation message msg_{vio}, denoted by $msg_{vio} \cap sg \neq \emptyset$, either when any of the current or old cells encapsulated in msg_{vio} are already contained in sg or when sg has cells which are in the same cell group as any of the cells in msg_{vio}. When there is only one RW, upon receiving a violation message msg_{vio}, the RW checks if there is a subgraph intersecting with msg_{vio}. If such sg exists, the RW adds msg_{vio} to sg, denoted by $msg_{vio} \xrightarrow{\text{add}} sg$, by adding both cells in msg_{vio}. If none of the subgraphs intersects with msg_{vio}, a new subgraph will be created with the two cells in msg_{vio}, denoted by $msg_{vio} \xrightarrow{\text{add}} null$. If more than one such subgraphs exist, Bleach merges these subgraphs to a single subgraph, and then adds msg_{vio} to it: $msg_{vio} \xrightarrow{\text{add}} (sg_1, sg_2, ...)_{merged}$.

We define a *subgraph identifier* $id(sg_k)$ to be the list of cell group IDs comprised in msg_{vio}: $id(cg_1, cg_2, ...)$. A subgraph can be expressed as $sg_k = (id(cg_1, cg_2, ...), [sc_1, sc_2, ...])$: it consists of a group of SC, stored in compressed format, as shown in Section 4.3.2. Note that when two subgraphs merge, their identifiers are also merged by concatenating both CG ID lists. To make the subgraph ID clear, sg_k can be presented as $sg_{id(cg_1,cg_2,...)}$.

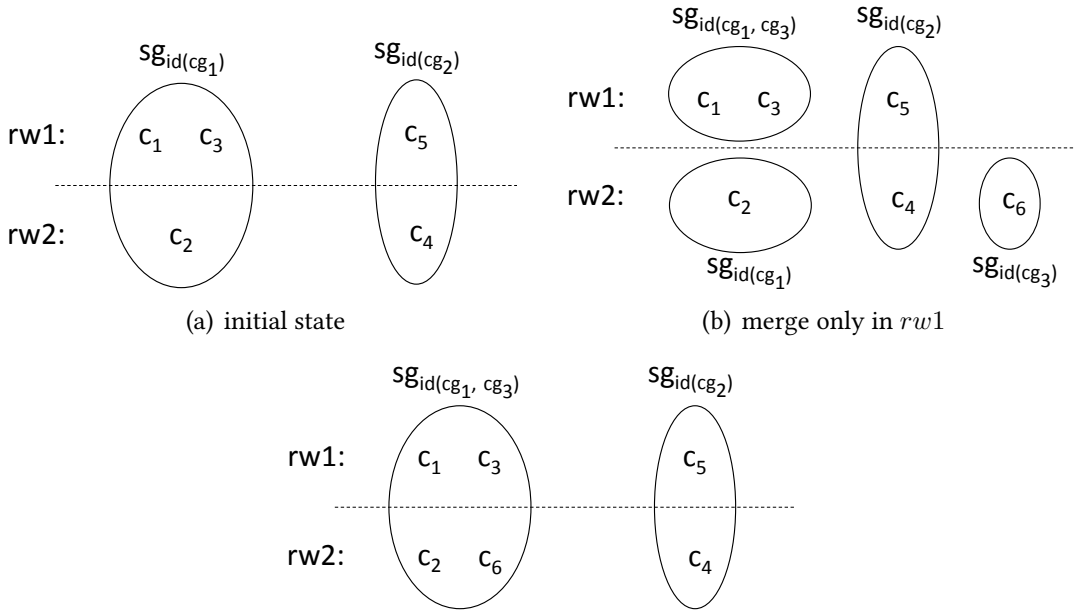


Figure 4.8 – Violation graph build example

Distributed violation graph. Due to the unbounded nature of streaming data, it is reasonable to expect the violation graph to grow to sizes exceeding the capacity of a single RW. As such, in Bleach, the violation graph is a distributed data structure, partitioned across all RWs.

However, unlike for DWs, the partitioning scheme can not be simply rule based, because a cell may violate multiple rules, creating issues related to coordination and load balancing. More generally, no partitioning scheme can guarantee that cells from a single violation message or a single subgraph to be placed in a single RW.

Therefore, Bleach partitions the violation graph based on cells using cells tuple IDs (e.g., hash partitioning). Since violation messages are broadcasted to all RWs, a violation message msg_{vio} is partially added to a subgraph sg in each RW, denoted by $msg_{vio} \xrightarrow{\text{p\_add}} sg$, such that only cells matching the partitioning scheme are added in sg. Hence, a subgraph spans several RWs, each storing a fraction of the cells comprised in the subgraph. We use the subgraph identifier to recognize partitions from the same subgraph.

An illustrative example is in order. Let's assume there are two RWs, $rw1$ and $rw2$, and the current violation graph consists in two subgraphs $sg_{id(cg_1)}$, containing cells c_1, c_2, c_3, and $sg_{id(cg_2)}$, containing cells c_4, c_5. In our example, the violation graph is partitioned as in Figure 4.8(a): both RWs have a portion of cells of every subgraph.

4.4.3 The Coordinator

The problem we address now stems from violation graph dynamics, which evolves as new violation messages stream into the repair module. As each subgraph is partitioned among all RWs, subgraph partitions must be identified by the same ID.

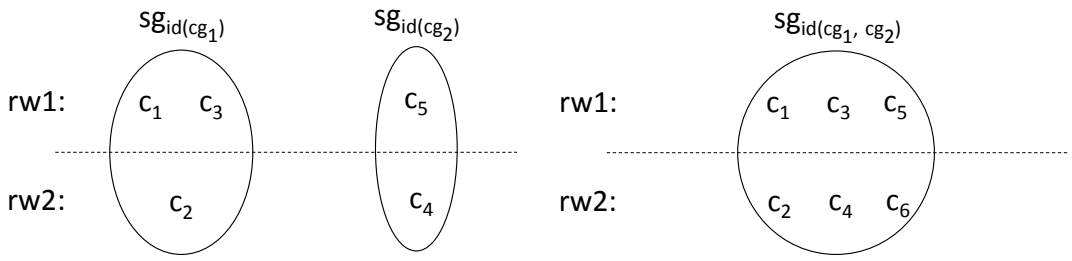
Continuing with the example from Figure 4.8(a), suppose a new violation message $\{id(cg_3), c_6, c_1\}$ is received by both RWs. Now, in $rw1$, the new violation is added to subgraph $sg_{id(cg_1)}$ since both the message and the subgraph share the same cell c_1: as such, the new subgraph becomes $sg_{id(cg_1,cg_3)}$. Instead, in $rw2$, the new violation triggers the creation of a new subgraph $sg_{id(cg_3)}$, since no common cells are shared between the message and existing subgraphs in $rw2$. The violation graph becomes *inconsistent*, as shown in Figure 4.8(b): this is a consequence of the independent operation of RWs. Instead, the repair algorithm requires the violation graph to be in a consistent state, as shown in Figure 4.8(c), where both RWs use the same subgraph identifier for the same equivalence class.

To guarantee the consistency of the violation graph among independent RWs, Bleach uses a stateless coordinator component that helps RWs agree on subgraph identifiers. In what follows we present three variants of the simple protocol RWs use to communicate with the coordinator.

Algorithm 5 RW-basic

procedure RECEIVE($[msg_{vio1}, msg_{vio2}, ...]$)
  Initialize a merge proposal mp
  for msg_{vioi} in $[msg_{vio1}, msg_{vio2}, ...]$ **do**
    Find $[sg_{i_1}, sg_{i_2}, ...]$ where $msg_{vioi} \cap sg_{i_j} \neq \emptyset$
    $msg_{vioi} \xrightarrow{\text{p\_add}} (sg_{i_1}, sg_{i_2}, ...)_{merged}$
    Add $(Attr(msg_{vioi}), id((sg_{i_1}, sg_{i_2}, ...)_{merged}))$ to mp
  end for
  Send mp to the coordinator
end procedure
procedure RECEIVE(md) ▷ merge decision
  for $(A_i, id(sg_i))$ in md **do**
    Find $[sg_{i_1}, sg_{i_2}, ...]$ where $id(sg_{i_j}) \subseteq id(sg_i)$
    Merge subgraphs $[sg_{i_1}, sg_{i_2}, ...]$ such that $id((sg_{i_1}, sg_{i_2}, ...)_{merged}) = id(sg_i)$
  end for
  Send a repair proposal to the aggregator
end procedure

RW-basic. Algorithm 5 demonstrates how RWs work with the coordinator in the RW-basic approach. When a RW receives violation messages for a tuple, it adds the cells in the messages to the violation graph, according to its local state and the partitioning



(a) initial state (b) after independent processing

Figure 4.9 – Example of violation graph built without coordination

scheme. Note that in Algorithm 5 (line 4-6) $(sg_{i_1}, sg_{i_2}, ...)_{merged}$ is a general case including when there is none or only one intersecting subgraph. Then, the RW creates a merge proposal containing the subgraph IDs for each conflicting attribute, and sends it to the coordinator. Once the coordinator receives merge proposals from all RWs, it merges subgraph IDs for each attribute from the various merge proposals and produces a merge decision which is sent back to all RWs. With the merge decision, RWs merge their local subgraphs and converge to a globally consistent state. Then, RWs are ready to generate repair proposals (more details in Section 4.4.4).

Clearly, such a simple approach to coordination harms Bleach performance. Indeed, the RW-basic scheme requires one round-trip message for every incoming data tuple, from all RWs.

However, we note that it is not necessarily true that the coordination is always needed for every tuple. For example, when every cell violates at most one rule, every subgraph would only have a single CG ID. Thus, coordination is not necessary. More generally, given violation messages for a tuple, coordination is only necessary when there is a complete violation message containing an old cell which already exists in the violation graph because of a different violation rule.

Figure 4.9 gives an example, where the initial state (Figure 4.9(a)) is the same as in Figure 4.8(a). Then, two violation messages, $\{id(cg_1), c_6, null\}$ and $\{id(cg_2), c_6, null\}$, are received. Cell c_6 is a current cell contained in the current tuple. Obviously $sg_{id(cg_1)}$ and $sg_{id(cg_2)}$ should merge into $sg_{id(cg_1,cg_2)}$. This can be accomplished without coordination by both repair workers, as shown in Figure 4.9(b). Indeed, each RW is aware that c_6 is involved in two subgraphs, although c_6 is only stored in $rw2$ because of the partitioning scheme.

Next, we use the above observations and propose two variants of the coordination mechanism that aim at bypassing the coordinator component to improve performance.

RW-dr. In RW-dr, the coordination is only conducted if it is necessary, and the repair worker sends a merge proposal to the coordinator and waits for the merge decision.

Algorithm 6 Update Top-k values

```
1:  State in a subgraph partition:
2:      values — a sorted array of top-k values
3:      val_cnt — a map keeping the cell count of every candidate value
4:  procedure UPDATE(Cell c)
5:      val = v(c)
6:      for i ← 0 to k − 1 do
7:          if values[i] = null then
8:              values[i] ← val return
9:          else if values[i] = val then
10:             return
11:         else if val_cnt(val) > val_cnt(values[i]) then
12:             r_val ← values[i]
13:             values[i] ← val
14:             break
15:         end if
16:     end for
17:     for j ← i + 1 to k − 1 do
18:         if values[j] = null or values[j] = val  then
19:             values[j] ← r_val
20:             return
21:         else
22:             tmp_val ← values[j]
23:             values[j] ← r_val
24:             r_value ← tmp_val
25:         end if
26:     end for
27: end procedure
```

However, this approach is not exempt from drawbacks: it may cause some data tuples in the stream to be delivered out of order. This is because the repair worker wait for the merge decision in a non-blocking way. The violation messages of a tuple which do not require coordination may be processed in the coordination gap of an earlier tuple.

RW-ir. With this variant, no matter if the violation messages of a tuple require coordination or not, a RW immediately updates its local subgraphs, executes the repair algorithm and emits a repair proposal downstream to the aggregator component. Then, if necessary, the RW lazily executes the coordination protocol. Clearly, this approach caters to system performance and avoids tuples to be delivered out of order, but might harm cleaning accuracy. Indeed, individual data repair proposals from a RW are based on a local view prior to finishing all necessary merge operations on subgraphs, which has a direct impact on equivalence classes.

4.4.4 The Aggregator

With the consistent distributed violation graph, each RW emits a data repair proposal, which includes the candidate values and their frequency computed in a local subgraph partition. In case there are too many candidate values, we only send the top-k values, where $k = 5$. To avoid sorting all candidate values whenever a new cell is added in a subgraph, RWs uses Algorithm 6 to update the top-k candidate values efficiently. Note that the frequencies of candidate values are already updated in Algorithm 6. The complexity of updating top-k values is $O(n)$ instead of $O(n^2)$ which is the complexity of resorting all the candidate values.

The aggregator component collects all repair proposals and selects the candidate value to repair a given cell as the one having the highest aggregate frequency. Finally, the aggregator modifies the current data tuple and outputs a clean data stream.

Note that the aggregator only modifies current tuples in the output stream. Instead, cells stored in the violation graph are not modified regardless of the repair decision: this allows to update frequency counts as new data streams into the system, thus steering the aggregator to make different repair decisions as the violation graph evolves. To avoid potential bottlenecks, Bleach can have multiple coordinators and aggregators, so that their workload can be distributed based on current tuple IDs.

4.5 Dynamic rule management

In stream data cleaning the rule set is usually not immutable but dynamic. Therefore, we now introduce a new component, the rule controller, shown in Figure 4.15, which allows Bleach to adapt to rule dynamics. The rule controller accepts rule updates as input and guides the detect and the repair module to adapt to rule dynamics without stopping the cleaning process and without loosing state. Rule updates can be of two types: one for adding a new rule and one for deleting an existing rule.

Detect. In the detect module, the addition of a rule triggers the instantiation of a new DW, as input tuples are partitioned by rule. The new DW starts with no state, which is built upon receiving new input tuples. As such, violation detection using past tuples cannot be achieved, which is consistent with the Bleach design goals. Instead, the deletion of an existing rule simply triggers the removal of a DW, with its own local data history.

Repair. In the repair module, the addition of a new rule is not problematic with respect to violation graph maintenance operations. Instead, the removal of a rule implies violation graph dynamics (subgraphs might shrink or split) which are more challenging to address.

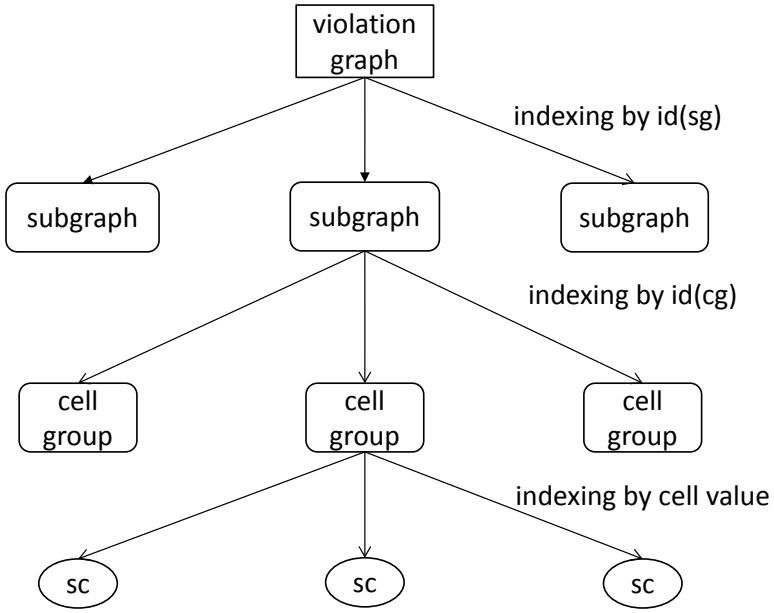


Figure 4.10 – The structure of violation graph

Thus, in a subgraph, we further group cells by cell groups. A subgraph now can also be expressed as: $sg_k = (id(cg_1, cg_2, ...), [cg_1, cg_2, ...])$, where each cell group gathers super cells. To facilitate the access of subgraphs and their cell groups, the hash indexing mechanism is also used in the violation graph. Figure 4.10 shows the structure of the violation graph stored in a repair worker. Note that a subgraph with multiple cell groups is indexed by every cell group ID in its subgraph ID.

Some cells might span multiple groups, as they may violate multiple rules. We label such peculiar cells as *hinge cells*. For each hinge cell, the subgraph keeps the IDs of its connecting cell groups: $c_i^* = (c_i, id(cg_{i_1}, cg_{i_2}, ...))$. Although a hinge cell exists in multiple cell groups, it is counted only once for its value. Thus, we can compute the cell frequency of a value v in a subgraph with m cell groups and n hinge cells as following:

$$val\_cnt(v) = \sum_{k=0}^{m} val\_cnt_{cg_k}(v) - \sum_{i=0}^{n}(|c_i^*| - 1).$$

where $val\_cnt(v)$ stands for the cell frequency of value v, $val\_cnt_{cg_k}(v)$ is the count from cell group cg_k and $|c_i^*|$ is the number of cell groups which hinge cell c_i^* with value v belongs to. The frequency is accumulated from all cell groups without duplicated counts. Hinge cells with the same value and the same connecting cell groups can also be compressed into super cells.

With the new organization of cells in subgraphs, the violation graph updates as following upon the removal of a rule. If a subgraph contains a single cell group related to the deleted rule, RWs are simply instructed to remove it. If a subgraph contains multiple

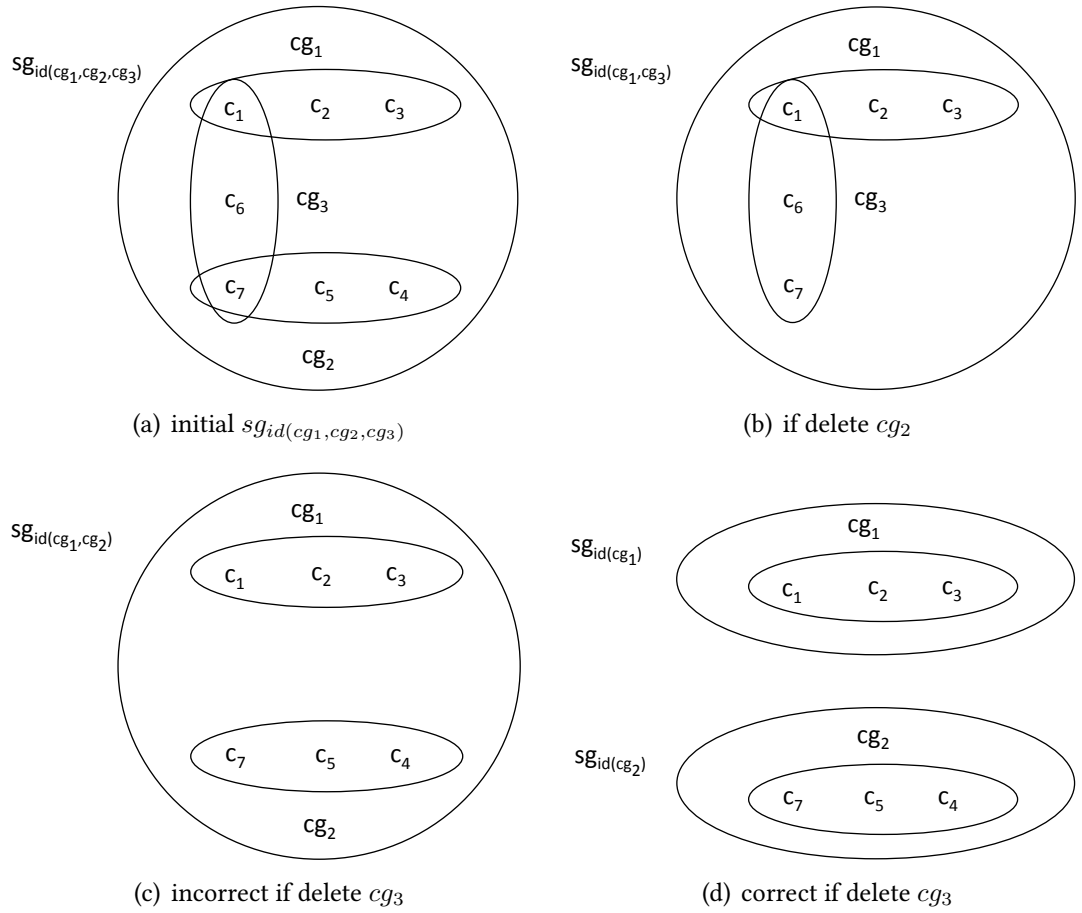
Algorithm 7 Subgraph Split: delete a rule

```
1:  subgraph sg state:
2:      [cg_1, cg_2, ...] — cell groups in sg
3:      [c_1^*, c_2^*, ...] — hinge cells in sg
4:  procedure DeleteRule(r)                                  ▷ deleted rule
5:      remove cell groups in [cg_1, cg_2, ...] relevant to rule r
6:      if no cell group is removed then
7:          return
8:      end if
9:      remove hinge cells in [c_1^*, c_2^*, ...] which do not connect at least two cell groups
10:     initialize a list of subgraph IDs, sid_list ← []
11:     for c_i^* in [c_1^*, c_2^*, ...] do
12:         n_sid ← id(cg_{i_1}, cg_{i_2}, ...)
13:         for sid in sid_list do
14:             if sid not disjoint n_sid then
15:                 n_sid ← n_sid + sid
16:                 remove sid from sid_list
17:             end if
18:         end for
19:         add n_sid to sid_list
20:     end for
21:     for cg_i in [cg_1, cg_2, ...] do
22:         if id(cg_i) is not contained in any element in sid_list then
23:             add id(cg_i) to sid_list
24:         end if
25:     end for
26:     if sid_list has more than one element then
27:         split sg by sid_list
28:     else if sid_list has exact one element then
29:         return
30:     else
31:         delete sg
32:     end if
33: end procedure
```

cell groups, RWs remove the cell groups related to the deleted rule and update the hinge cells. With the remaining hinge cells, RWs check the connectivity of the remaining cell groups in the subgraph and decide to split the subgraph or not. A detailed algorithm can is shown in Algorithm 7.

For each subgraph in the violation graph, we are given the cell groups $[cg_1, cg_2, ...]$ and the hinge cells $[c_1^*, c_2^*, ...]$ (line 1-3). When a rule is deleted, the relevant cell groups are removed (line 5). If no CG is removed, this subgraph remains the same, the procedure ends (line 6-8). Then, the hinge cells are updated that hinge cells which connect only

one single CG after removing cell groups are also removed (line 9). Next, the RW starts to construct a subgraph ID list sid_list for sg from the hinge cells and the cell groups (line 10-25). All subgraph IDs in sid_list should not share any CG ID with each other.



(a) initial $sg_{id(cg_1,cg_2,cg_3)}$ (b) if delete cg_2

(c) incorrect if delete cg_3 (d) correct if delete cg_3

Figure 4.11 – Subgraph split example

If there are more than one subgraph ID in sid_list, the subgraph is split into subgraphs whose IDs are from sid_list (line 26-27). If there is only one subgraph ID in sid_list, the subgraph does not need to be split and the only subgraph ID becomes its new subgraph ID (line 28-29). If sid_list is empty, the subgraph is simply deleted (line 31).

An example of a split operation can be seen in Figure 4.11. The initial state of a subgraph is shown in Figure 4.11(a): the subgraph is $sg_{id(cg_1,cg_2,cg_3)}$, and its contents are three cell groups. Cell c_1 and c_7 are *hinge* cells, which work as bridges, connecting different cell groups together. Now, as a simple case, assume we want to remove the rule pertaining to cg_2: the subgraph should become $sg_{id(cg_1,cg_3)}$, as shown in Figure 4.11(b). Note that cell c_7 looses its status of hinge cell. A more involved case arise when we delete the rule pertaining to cg_3 instead of the rule pertaining to cg_2. In this case, the subgraph should not become $sg_{id(cg_1,cg_2)}$ as shown in Figure 4.11(c). Indeed, removing cg_2 eliminates all

existing hinge cells connecting the remaining cell groups. Thus, the subgraph must split in two separate subgraphs $sg_{id(cg_1)}$ and $sg_{id(cg_2)}$ as shown in Figure 4.11(d).

4.6 Windowing

Bleach provides windowed computations, which allow expressing data cleaning over a sliding window of data. Despite being a common operation in most streaming systems, window-based data cleaning addresses the challenge of the unbounded nature of streaming data: without windowing, the data structures Bleach uses to detect and repair a dirty stream would grow indefinitely, which is unpractical.

In this section, we discuss two windowing strategies: a basic, tuple-based windowing strategy and an advanced strategy that aim at improving cleaning accuracy.

4.6.1 Basic Windowing

The underlying idea of the basic windowing strategy is to only use tuples within the sliding window to populate the data structures used by Bleach to achieve its tasks. Next, we outline the basic windowing strategy for both DWs and RWs operation.

Windowed Detection. We now focus on how DWs maintain their local data history. Clearly, the data history only contains cells that fall within the current window. When the window slides forward, DWs update the data history as follows: *i)* if a cell group ends up having no cells in the new window, DWs simply delete it; *ii)* for the remaining cell groups, DWs drop all cells that fall outside the new window, and update accordingly the remaining super cells.

Note that, if implemented naively, the first operation above can be costly as it involves a linear scan of all cell groups. To improve the efficiency of data history updates, Bleach uses the following approach. It creates a FIFO queue of k lists, which store cell groups. In case the sliding step is half the window size, $k = 2$; more generally, we set k to be the window size divided by the sliding step. Any new cell group from the current window enters the queue in the k-th list. Any cell group updates, e.g. due to a new cell added to the cell group, "promotes" it from list j to list k. As the window slides forward, the queue drops the list (and its cell groups) in the first position and acquires a new empty list in position $k + 1$.

Windowed Repair. Now we focus on how to maintain the violation graph in RWs. Again, the violation graph only stores cells within the current window. When the window slides forward, RWs update the violation graph as follows:

- If a subgraph has no cells in the new window, RWs delete the subgraph;

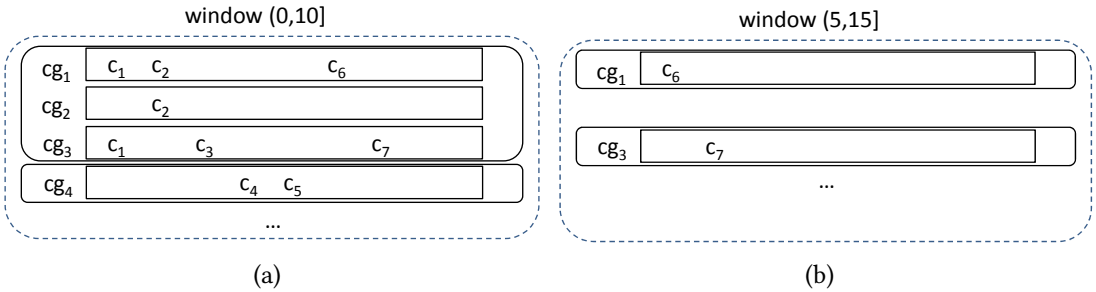


Figure 4.12 – tuple-based windowing example

- For the remaining subgraphs, if a cell group has no cells in the new window, RWs delete the cell group;
- RWs also delete hinge cells that are outside of the new window. This could require subgraphs to split, as they could miss a "bridge" cell to connect its cell groups;
- For the remaining cell groups, RWs drop all cells outside of the new window, and update the remaining super cells accordingly.

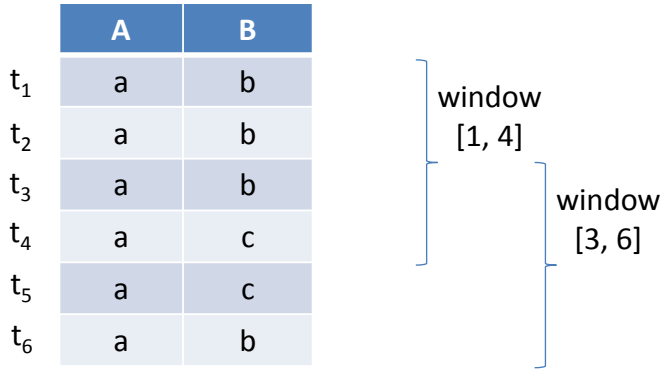
For efficiency reasons, Bleach uses the same k-list approach described for DWs to manage violation graph updates due to a sliding window.

Figure 4.12 gives an example about how the violation graph is updated when a tuple-based window slides. For simplicity, we assume in this example that the tuple id of cell c_i is i. When the tuple window is $(0, 10]$, there are two subgraphs, $sg_{id(cg_1,cg_2,cg_3)}$ and $sg_{id(cg_4)}$. But when the window slides to $(5, 15]$, $sg_{id(cg_4)}$ is dropped because it does not have any cells in the new window. Likewise, the cell group cg_2 is deleted in $sg_{id(cg_1,cg_2,cg_3)}$. But instead of shrinking to $sg_{id(cg_1,cg_3)}$, $sg_{id(cg_1,cg_2,cg_3)}$ is split into two subgraphs, $sg_{id(cg_1)}$ and $sg_{id(cg_3)}$, since the two remaining cell groups do not have any common cells as hinge cells within the new window.

4.6.2 Bleach Windowing

The basic windowing strategy only relies on the data within the current window to perform data cleaning, which may limit the cleaning accuracy. We begin with a motivating example, then describe the Bleach windowing strategy, that aims at improving cleaning accuracy. Note that here we only focus on the repair module and its violation graph, since Bleach windowing does not modify the operation of the detect module.

Figure 4.13(a) illustrates a data stream of two-attribute tuples. Assume we use a single FD rule $(A \rightarrow B)$, a window size of 4 tuples, a sliding step of 2 tuples, and the *basic* windowing strategy. When t_4 arrives, the window covers tuples $[1, 4]$. According to the repair algorithm, Bleach repairs $t_4(B)$ and sets it to the value b. Note that as we



(a) input data

	A	B
t_1	a	b
t_2	a	b
t_3	a	b
t_4	a	**b**
t_5	a	c
t_6	a	b

(b) output data with basic windowing

	A	B
t_1	a	b
t_2	a	b
t_3	a	b
t_4	a	**b**
t_5	a	**b**
t_6	a	b

(c) output data with Bleach windowing

Figure 4.13 – Motivating example: Basic vs. Bleach windowing.

described in Section 4.4, $t_4(B)$ remains unchanged in the violation graph. Now, when tuple t_5 arrives, the window moves to cover tuples $[3, 6]$, even though t_6 has yet to arrive. With only three tuples in the current window, the algorithm determines $t_5(B)$ is correct, because now the majority of tuples have value c. The output stream produced using basic windowing is shown in Figure 4.13(b). Clearly, cleaning accuracy is sacrificed, since it is easy to see that $t_5(B)$ should have been repaired to value b, which is the most frequent value overall. Hence, we need a different windowing strategy to overcome such problems.

Bleach windowing relies on an extension of a super cell, which we call a *cumulative super cell*. The idea is for the violation graph to accumulate past state, to complement the view Bleach builds using tuples from the current window. Hence, a cumulative super cell is represented as a super cell, with an additional field that stores the number of occurrences of cells with the same RHS value, including those that have been dropped because they fall outside the sliding window boundaries.

When using Bleach windowing, RWs maintain the violation graph by storing cumulative super cells instead of super cells. When the window slides forward, RWs update

the violation graph as follows. The first two steps are equivalent to those for the basic strategy. The last two steps are modified as follows:

- For the remaining subgraphs, RWs delete hinge cells that do not bridge cell groups anymore because of the update. Also, RWs split subgraphs according to the remaining hinge cells;
- For the remaining cell groups and hinge cells, RWs update cumulative super cells, "flushing" cells which fall outside the new window while keeping their count.

Now, going back to the example in Figure 4.13(a), when tuple t_5 arrives, Bleach stores two cumulative super cells: $csc_1(id(t) = [3], value = \text{`}b\text{'}, count = 3)$ and $csc_2(id(t) = [4, 5], value = \text{`}c\text{'}, count = 2)$. Although t_1 and t_2 have been deleted because they are outside the sliding window, they still contribute to the count field in csc_1. Therefore, tuple $t_5(B)$ is correctly repaired to value b, as shown in Figure 4.13(c).

4.6.3 Discussion

When using cumulative super cells, Bleach keeps tracks of candidate values to be used in the repair algorithm as long as cell groups remain. By using cumulative super cells for hinge cells, subgraphs only split if some cell groups are removed when the window moves forward. Note that the introduction of cumulative super cells does not interfere with dynamic rule management: in particular, when deleting a rule, subgraphs update correctly when hinge cells use the cumulative format. Overall, to compute the count of a candidate value in a subgraph, cumulative super cells accumulate the counts of relevant cumulative super cells from all cell groups, taking into account any duplicate contributions from hinge cells.

Obviously, Bleach windowing requires more storage than basic windowing, as cumulative super cells store additional information, and because of the "flush" operation described earlier, which keeps a super cell structure, even when it has an empty cell list. Section 4.8 demonstrates that such additional overhead is well balanced by superior cleaning accuracy, making Bleach windowing truly desirable.

4.7 Rule Dependency

Until now, we discussed how Bleach performs data cleaning, especially when rules may share the same RHS attribute. However, we did not discuss the case in which the RHS attribute of a rule can be at the same time the LHS attribute of another rule. For clarity, we give the notion of *rule dependency* that rule r_2 is dependent on rule r_1, $r_1 \xleftarrow{\text{dep}} r_2$, if the RHS attribute of r_1 is one of the LHS attributes of r_2. With the existence of rule

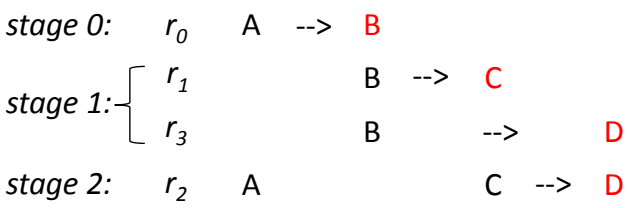


Figure 4.14 – The three rule subsets

dependency among rules, the cleaning accuracy of Bleach may be unsatisfactory. This is because Bleach may introduce new violations (for rule r_2) when repairing the violations (of rule r_1).

4.7.1 Multi-Stage Bleach

We design a variant of Bleach, *Multi-Stage Bleach*, to cope with the rule dependency. In Multi-Stage Bleach, the cleaning process consists of multiple stages. Each stage includes both violation detection and violation repair, but only corresponding to a subset of violation rules. The rules are divided into multiple subsets, $[ruleset_0, ruleset_1, ...]$, according to rule dependency: The rules in the first rule set $ruleset_0$ are not dependent on any other rules; Other rules in rule set $ruleset_i$ are dependent on at least one of the rules in $ruleset_{i-1}$. If a rule r is not dependent on any rule in a rule set $ruleset_i$, then $ruleset_i \xleftarrow{\text{dep}} r$.

We give an example in which a dirty data stream with schema $S(A, B, C, D)$ is to be cleaned and four FD rules are defined: $r_0 = (A \rightarrow B), r_1 = (B \rightarrow C), r_2 = (A, C \rightarrow D)$ and $r_3 = (B \rightarrow D)$. These four rules are divided into three rule sets as shown in Figure 4.14, because $r_0 \xleftarrow{\text{dep}} r_1 \xleftarrow{\text{dep}} r_2$ and $r_0 \xleftarrow{\text{dep}} r_3$. Each rule set is binded to a cleaning stage in Multi-Stage Bleach, where only violations of rules in this rule set are detected and repaired.

Figure 4.15 gives the overview of Multi-Stage Bleach, where the initial input stream is the input stream of the first stage and the stage $i + 1$ accepts the cleaned output stream of stage i as the input data stream. The last stage outputs the final output stream. Each stage is an independent Bleach instance, although they share the same computation resources (the detect module and the repair module). In each stage, the LHS attributes are immutable as in the example of Figure 4.14, the cells of attribute B and C can only be modified in stage 0 and stage 1 respectively. Thus, any modification of tuples in a stage will not cause any new violation in the preceding stages.

Consider a special example where there is a dirty data stream with schema $S(A, B, C, D)$ and three defined FD rules, $r_4 = (A \rightarrow B), r_5 = (B \rightarrow C)$ and $r_6 = (C \rightarrow A)$. Clearly, r_5 is dependent on r_4, r_6 is dependent on r_5 and r_4 is dependent on r_6, as shown in

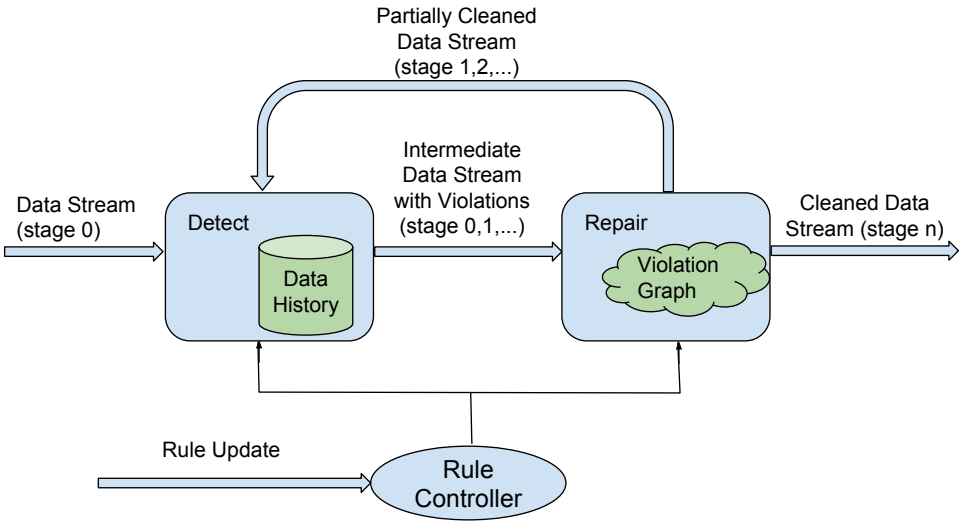


Figure 4.15 – The Overview of Multi-Stage Bleach

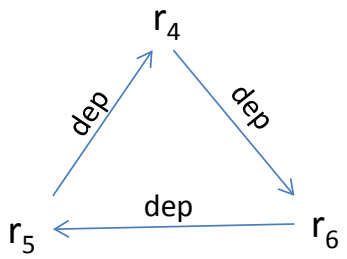


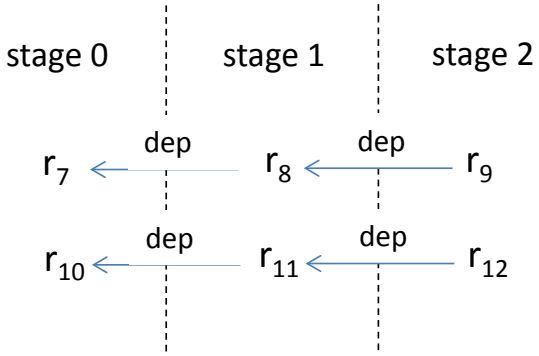
Figure 4.16 – circular dependency among rules

Figure 4.16. There is a *circular dependency* among these three rules that no rule that is not dependent on any other rule. In this case, Multi-Stage Bleach groups these rules into the same stage. A better solution is in the plan of our future work, although this situation is quite rare.

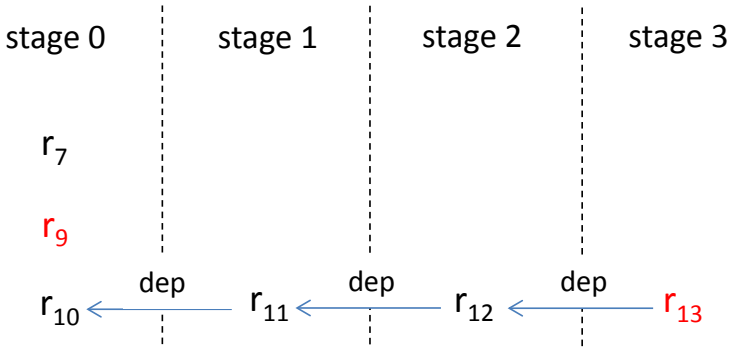
4.7.2 Dynamic Rule Management

Multi-Stage Bleach also supports dynamic rule management. The particular task is to update the binding between rules and stages. When a rule is added to a stage or deleted, other rules may also need to be reorganized, migrating from one stage to another.

Figure 4.17 gives an example how violation rules in Multi-Stage Bleach are reorganized when updating rules. The initial six rules from r_7 to r_{12} are divided into three stages, as shown in Figure 4.17(a) with their dependency relationship. Assume that rule r_8 is deleted and a new rule, r_{13}, is added. The rules will be reorganized as shown in Figure 4.17(b). Rule r_9 is moved to stage 0, since it is not depend on any other rules now. The new rule r_{13} is added in a new stage, stage 3, as it is depend on rule r_{12} in stage 2.



(a) before updating rules



(b) delete r_8 and add r_{13}

Figure 4.17 – Motivating example: Basic vs. Bleach windowing.

Therefore, in Multi-Stage Bleach, not only rules but also stages are dynamic. Since the detect and repair modules are shared by all stages, the dynamic resource management is not mandatory in Multi-Stage Bleach.

In Bleach, a rule is associated with states including cell groups and subgraphs, which provides Bleach the ability to perform accurate incremental data cleaning. Unfortunately, states in different stages are not compatible with each other, because data streams in different stages are not identical:

- Data streams in different stages are **unsynchronized** that preceding stages process more tuples than succeeding stages;
- Tuples in different stages are **inconsistent** that tuples with the same tuple ID may have different attribute values.

As a consequence, Multi-Stage Bleach processes migrating rules as newly added rules that their states in the old stages are just deleted. To summarize, when a rule is updated, Multi-Stage Bleach reassigns all rules to stages, only keeping the states of the rules which belong to the same stage before and after the reassignment.

Table 4.1 – Example rule sets used in our experiments.

$r_0 : ss\_item\_sk \rightarrow i\_brand, (ss\_item\_sk \neq null)$
$r_1 : ss\_item\_sk \rightarrow i\_category, (ss\_item\_sk \neq null)$
$r_2 : ca\_state, ca\_city \rightarrow ca\_zip, (ca\_state, ca\_city \neq null)$
$r_3 : ss\_promo\_sk \rightarrow p\_promo\_name, (ss\_promo\_sk \neq null)$
$r_4 : ss\_store\_sk \rightarrow s\_store\_name, (ss\_store\_sk \neq null)$
$r_5 : ss\_ticket\_num \rightarrow s\_store\_name, (ss\_ticket\_num \neq null)$
$r_6 : file\_extension \rightarrow mime\_type, (file\_extension \neq null)$

4.8 Evaluation

We built Bleach prototype implementation using Apache Storm [70].[4] Input streams, including both the data stream and rule updates, are fed into Bleach using Apache Kafka [120]. We conducted all experiments in a cluster of 18 machines, with 4 cores, 8 GB RAM and 1 Gbps network interface each.

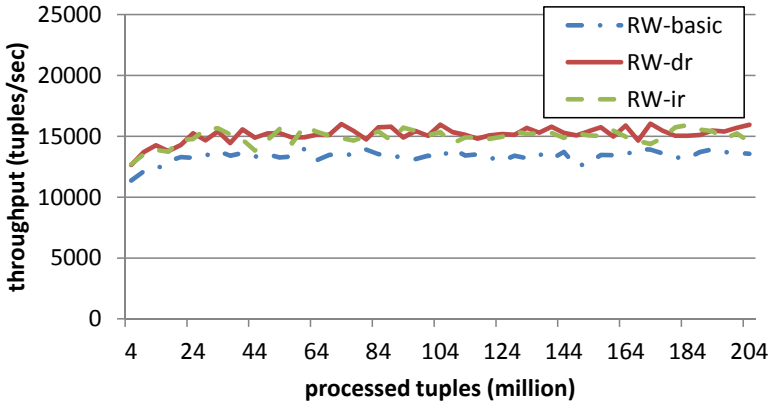
We evaluate Bleach using both synthetic and real-life datasets. The synthetic dataset is generated from TPC-DS (with scale factor 100 GB) where we join a fact table *store_sales* with its dimension tables to build a single table (288 million tuples). We manually design six CFD rules, from r_0 to r_5, as shown in Table 4.1. Among these rules, r_4 and r_5 have the same RHS attribute s_store_name. We generate a dirty data stream as follows: we modify the values of RHS attributes with probability 10% and replace the values of LHS attributes with NULL with probability 10%.[5] The real-life dataset we use is the result of merging all the log files of Ubuntu One servers [98] for 30 days (773 GB of CSV text). Instead of modifying any values, the dataset already contains dirty records. We design a CFD rule, r_6, as shown in Table 4.1. With rule r_6, the dirty data percentage of the dataset is roughly $7 * 10^{-3}$%. By exporting the datasets to Kafka, we simulate "unbounded" data streams. In all the experiments, we use Bleach windowing as the default windowing strategy and set the window size to 2M tuples and the sliding step to 1M tuples, unless otherwise specified. The synthetic dataset is used in all experiments except in our last battery of experiments, where the real-life dataset is used.

Our goal is to demonstrate that Bleach achieves efficient stream data cleaning under real-time constraints. Our evaluation uses *throughput, latency* and *dirty ratio* as performance metrics. We express the dirty ratio as the fraction of dirty data remaining in the data stream: the smaller the dirty ratio, the higher the cleaning accuracy. The processing latency is measured from uniformly sampled tuples (1 per 100).

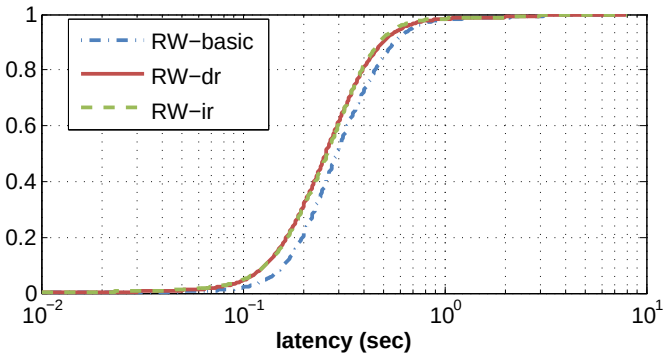
[4] Nothing prevents Bleach to be built using alternative systems such as Apache Flink, for example.

[5] In our experiments we also used BART [100], which is a well accepted dirty data generator. However, BART fails to scale to hundreds of millions of tuples due to memory reasons. Thus, we present results obtained using our custom process, which mimics that of BART but scales to large data streams.

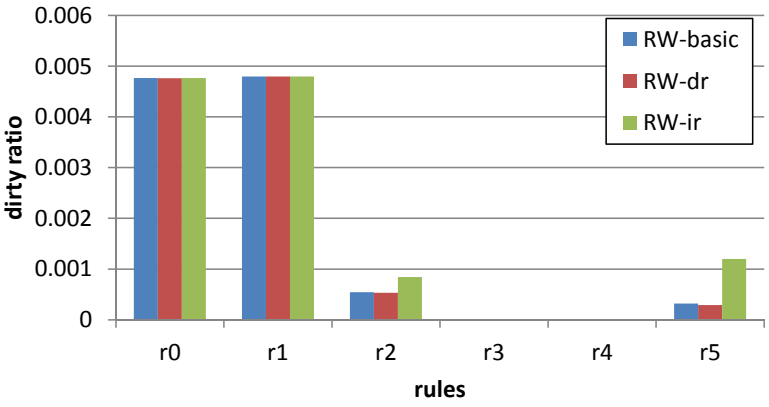
4.8.1 Comparing Coordination Approaches



(a) Throughput



(b) Latency



(c) Dirty Ratio

Figure 4.18 – Comparison of coordination mechanisms: RW-basic, RW-dr and RW-ir.

In this experiment we compare the three RW approaches discussed in Section 4.4, according to our performance metrics, as shown in Figure 4.18: *RW-basic* requires coordination among repair workers for each tuple; *RW-dr* omits coordination for tuples when possi-

ble; *RW-ir* is similar to *RW-dr*, but allows repair decisions to be made before finishing coordination.

Figure 4.18(a) shows how Bleach throughput evolves with processed tuples. The throughput with both *RW-dr* and *RW-ir* is around 15K tuples/second, whereas *RW-basic* achieves roughly 13K tuples/second. The inferior performance of RW-basic is due to the large number of coordination messages required to converge to global subgraph identifiers, while *RW-dr* and *RW-ir* only require 7% coordination messages in *RW-basic*. Figure 4.18(b) shows the CDF of the tuple processing latency for the three RW approaches. *RW-basic* has the highest processing latency, on average 364 ms. The processing latency of *RW-ir* is on average 316 ms. *RW-dr* average latency is slightly higher, about 323 ms. This difference is due again to the additional round-trip-messages required by coordination: with *RW-ir*, RWs make their repair proposals without waiting for coordination to complete, therefore the small processing latency. Figure 4.18(c) illustrates the cleaning accuracy. All three approaches lower the ratio of dirty data significantly to at most 0.5% (even 0% for rule r_3 and r_4). For the first five rules, the three approaches achieve similar cleaning accuracy. Instead, for rule r_5 the *RW-ir* method suffers and the dirty ratio is larger. Indeed, for rule r_5 whose cleaning accuracy is heavily linked to rule r_4, *RW-ir* fails to correctly update some of its subgraphs because it eagerly emits repair proposals without waiting for coordination to complete.

In the following experiments, we use the *RW-dr* as the default mechanism.

4.8.2 Comparing Windowing Strategies

In this experiment, we compare the performance of the basic and Bleach windowing strategies. Additionally, for stress testing, we increase the input dirty data ratio by modifying the values of RHS attributes with probability 50% for data in the interval from 40M to 42M tuples.

As shown in Figure 4.20 and Figure 4.21, the two windowing strategies are essentially equivalent in terms of throughput and latency: this is good news, as it implies the requirement for *cumulative super cells* is a negligible toll on performance. Next, we focus on a detailed view of the cleaning accuracy, which is shown in Figure 4.19. Bleach windowing achieves superior cleaning accuracy: in general, the dirty ratio is one order of magnitude smaller than that of basic windowing. This advantage is kept also in presence of a dirty ratio spike in the input data. In particular, for rules r_3 and r_4, Bleach windowing achieves 0% dirty ratio, irrespectively of the dirty ratio spike.

Overall, Bleach windowing reveals that keeping state from past windows can indeed dramatically improve cleaning accuracy, with little to no performance overhead.

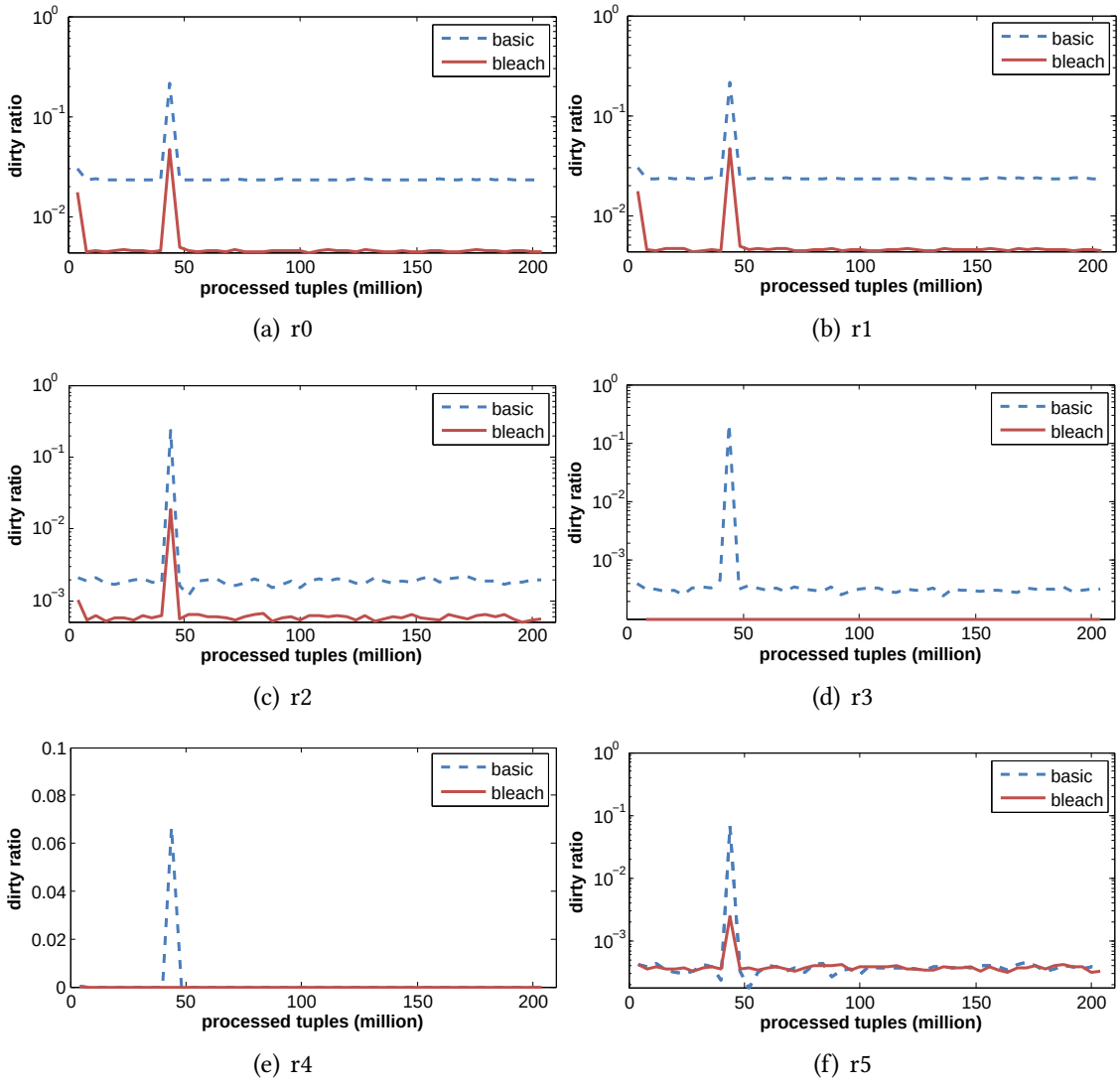


Figure 4.19 – Cleaning accuracy of two windowing strategies

4.8.3 Comparing Different Window sizes

In this experiment, we evaluate Bleach with different window sizes. We set the window size as 200K, 500K, 1M and 2M respectively (the sliding step is half of the window size), and the experiment result is as shown in Figure 4.22 and Figure 4.23. We see that Bleach has a higher chance to clean the data stream with more tuples in the window.

Figure 4.22 shows that the throughput decreases as the size of window increases. With a larger window, there are more tuples to be detected for violations in the data history. Hence, more violations are detected and sent to the repair module. The violation graph in the RWs will be larger. As a consequence, any subgraph operations including merging and split will take more time to finish. With our implementation the throughput drops 23% when the window size increases 10 times. In contrast, Figure 4.23 demonstrates that

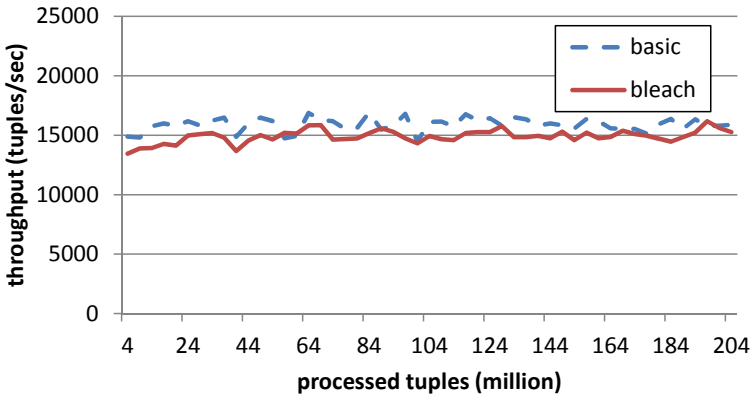


Figure 4.20 – Throughput of two windowing strategies

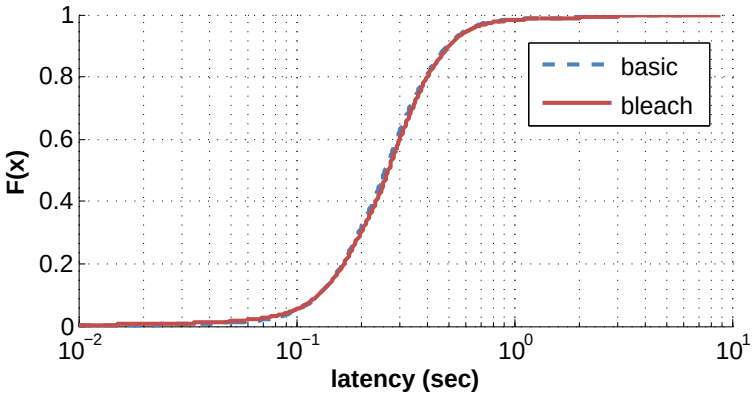


Figure 4.21 – Processing latency CDF of two windowing strategies

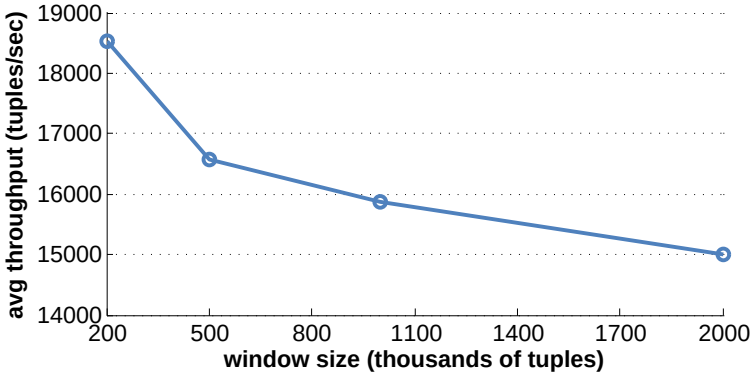


Figure 4.22 – Throughput of different window sizes

the cleaning accuracy may increase more than 10 times when the window size increases 10 times[6].

[6]The cleaning accuracy of rule r_3 and r_4 is not shown in Figure 4.23, as their cleaning accuracy is all 100% with four different window sizes.

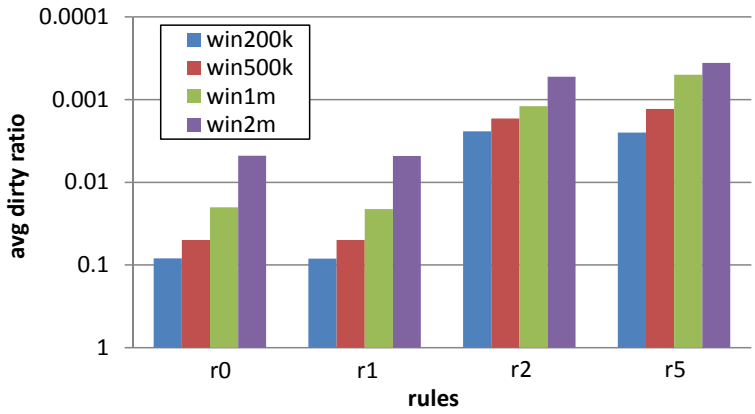


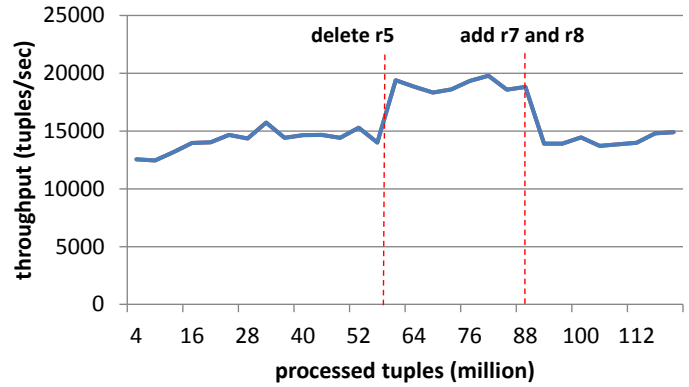
Figure 4.23 – Cleaning accuracy of different window sizes

4.8.4 Dynamic Rule Management

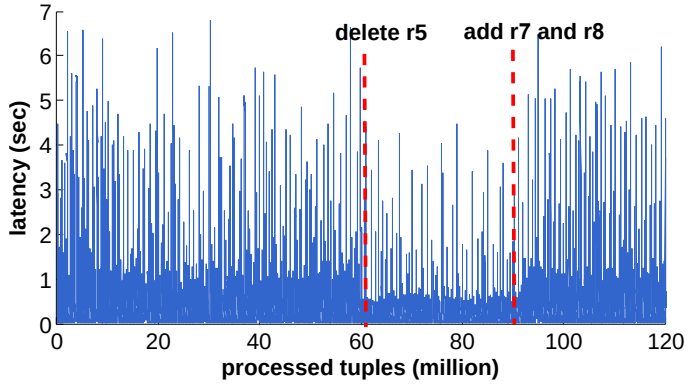
Next, we study the performance of Bleach in presence of rule dynamics, as shown in Figure 4.24. To do this, we initially use the same input data stream and rule set as in Section 4.8.1. However, while Bleach is cleaning the input stream, we delete rule r_5 and add two new rules r_7 $(ss\_ticket\_num \rightarrow c\_email\_addr, (ss\_ticket\_num \neq null))$ and r_8 $(ss\_customer\_sk \rightarrow c\_email\_addr, (ss\_customer\_sk \neq null))$, as indicated in the Figure.

Figure 4.24(a) and Figure 4.24(b) show the evolution in time of throughput and latency, whereas Figure 4.24(c) gives the CDF of the processing latency. Figure 4.24(a) shows that rule dynamics can result in an increase in throughput. Indeed, removing r_5 (at the 60M tuple) implies that Bleach needs to manage fewer rules; in addition, r_4 becomes simpler to manage, as there are no more intersections with r_5. Similarly, Figure 4.24(b) shows that also latency decreases upon r_5 removal. When rules r_7 and r_8 are added (at the 90M tuple), the throughput drops and the latency grows, as Bleach has more rules to manage and because the new rules have intersecting attributes, requiring more work from RWs. Figure 4.24(c), shows the latency distribution computed from output tuple samples. While the average latency is roughly 320 ms, we notice a tail in the distribution, indicating that some (few) tuples experience latencies up to seconds. This has been observed across all our experiments, and is due to the sliding window mechanism, which imposes computationally demanding operations when updating the violation graph, resulting also in rather low-level garbage collection problems.

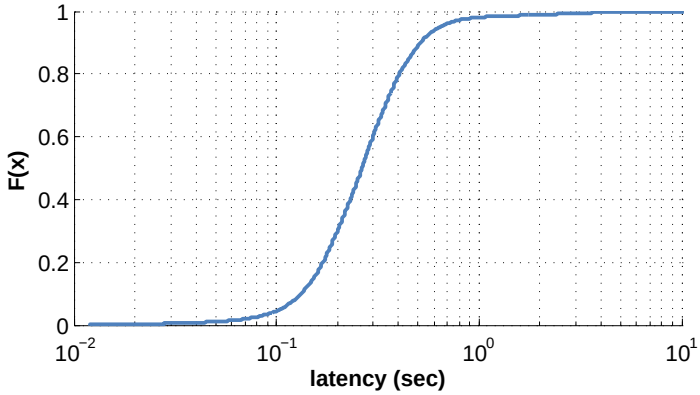
Overall, we conclude that Bleach supports dynamic rule management seamlessly, with essentially no impact on performance, and no system restart required.



(a) Throughput



(b) Latency in time



(c) Latency CDF

Figure 4.24 – Bleach performance with dynamic rule management.

4.8.5 Comparing Bleach to a Baseline Approach

We conclude our evaluation with a comparative analysis of Bleach and a baseline approach, which is based on the micro-batch streaming paradigm (that we refer to as micro-batch cleaning). Essentially, micro-batch cleaning buffers input data records and performs batch data cleaning periodically, as determined by a sliding window. Our im-

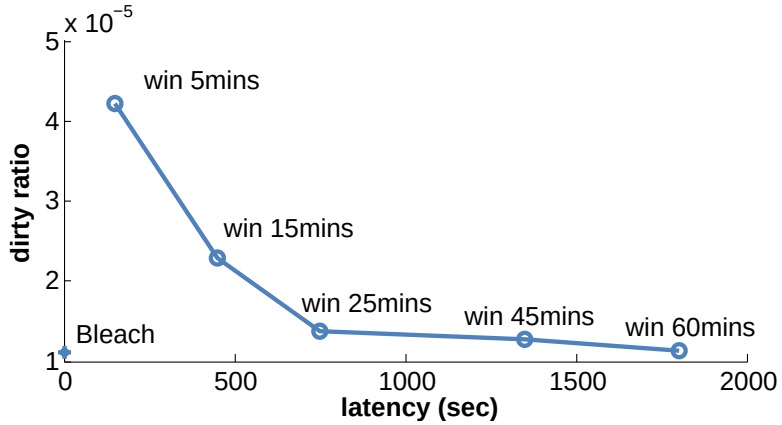


Figure 4.25 – Comparison of Bleach vs. micro-batch cleaning: latency/accuracy tradeoff.

plementation uses Spark Streaming in which window processing is time-based and not tuple-based.[7] To demonstrate the performance of micro-batch cleaning and compare it to Bleach, we perform a series of experiments whereby we increase the sliding window size. We use the stream data input from the Ubuntu One trace file and its violation rule r_6. We feed the input stream at a constant throughput of 1317 tuples/second, which is the average speed of the generation of the trace file. Thus, we focus on performance analysis expressed only in terms of latency and dirty ratio. Instead of consuming all the tuples in the input stream which will take 30 days, each experiment lasts 24 hours.

Figure 4.25 illustrates the performance of both systems. As expected, for the micro-batch cleaning, the average latency is proportional to the window size: larger sliding windows entail higher latencies. Indeed, since the data in the input stream is uniformly distributed, the average latency equals the sum of half of the window size and the average execution time for cleaning data in each window. As for the cleaning accuracy, intuitively, the larger the sliding window, the more accurate micro-batch cleaning gets, hence a smaller output stream dirty ratio. Although the tuple-based window in Bleach contains roughly the same amount of tuples as the time-based window of 25 minutes in micro-batch cleaning, Bleach achieves better cleaning accuracy because of its cumulative statistics. In particular, we notice that to achieve the same cleaning accuracy as Bleach, micro-batch cleaning requires the sliding window to be larger than 45 minutes, which incurs in an average latency larger than 22 minutes. Instead, in Bleach, the average latency is less than 200 ms.

[7] For our experiment, the performance difference between the two is negligible.

4.9 Related work

Many approaches to data cleaning [96,99,101,102,111,113,114,118] tackle the problem of detecting and repairing dirty data in a database based on predefined data quality rules. Next, we discuss a number of representative approaches.

Kolahi et al. [114] focus on repairing an inconsistent database that violates a set of functional dependencies by making the smallest possible value modifications. They define an optimum repair as a database that satisfies the rules, and minimizes a distance measure that depends on the number of corrections made. In the paper, it is proven that checking the existence of a repair within a certain distance of a database is NP-complete and finding a constant-factor approximation for the optimum repair is NP-hard. Hence, they propose an approximation algorithm to produce a repair whose distance is within a constant factor of the optimum repair distance. With the approximation algorithm, the new violations caused by resolving initial violations are fixed by assigning distinct variables to tuples.

NADEEF [101] is an extensible and generic data cleaning system, providing an end-to-end off-the-shelf solution to automate both the detection and the repairing of violations with respect to a set of heterogeneous quality rules. The main design of NADEEF is to separate a *programming interface* which allows users to define their quality rules, and a *core* that implements algorithms to repair dirty data by treating all kinds of rules holistically. Two algorithms are implemented in NADEEF, including a variable-weighted MAX-SAT solver based algorithm and an equivalence class based algorithm.

The holistic data cleaning algorithm [102], proposed by Chu et al., can also be used in the core of NADEEF. In the holistic data cleaning algorithm, all types of quality rules are generalized in one type of rules, *Denial Constraints* (DCs) rules. The subset of data which does not conform to the defined DC rules are encoded in a conflict hypergraph, in which a node is a cell (an attribute in a tuple) and a hyperedge represents a violation. By modifying the values of cells in the minimum vertex cover to satisfy DC rules, the dataset is cleaned.

As nowadays data often grows beyond a single machine's capacity, distributed databases becomes more and more popular. Chen et al. [113] focus on repairing functional dependency violations in a horizontally partitioned database. In particular, they study data repairing with equivalence classes in the distributed setting. They implement distributed equivalence classes by combing the disjoint-set forest data structure with the linked list technique. Their cleaning algorithms are typically designed to minimize data shipment among between multiple sites and parallel computation time.

BigDansing [99] is a distributed version of NADEEF, which also tackles efficiency, scalability, and ease-of-use issues in data cleaning. The system can run on top of common

data processing platforms, such as Apache Hadoop and Apache Spark. It first generates the violation graph as a hypergraph containing the detected violations and their possible fixes. Then, to determine the final fixes, BigDansing partitions the violation graph into multiple connected components so that each can be processed independently in a single machine.

Some works also consider that the data stored in databases can be dynamic, i.e., frequently updated. Since it is prohibitively expensive to recompute the entire violations when the database is updated, Fan et al. [96,97] propose incremental algorithms to detect violations in a distributed database when database is updated, which is similar to the violation detection in Bleach. With the proposed algorithms, both the communication cost and computation cost of violation detection process are linear in the size of database updates and the changes to violations, independent of the size of the base relation. Rather than repairing static data by static quality rules, continuous data cleaning [115] considers repairing both quality rules (FDs) and the data with user intervention in a more dynamic environment, in which data may be updated and quality rules may evolve. This is achieved through a probabilistic classifier that predicts the type of repair (data, FDs, or a hybrid of both) needed to resolve an inconsistency, and that learns from past user repair preferences to recommend more accurate repairs in the future.

All the works introduced above focus on cleaning data stored in data warehouses by batch processing, which achieves high accuracy but suffers high latency. In contrast, stream processing [106, 116, 117] requires to be real-time, a challenge that has drawn increasing attention from researchers. Nevertheless, stream data cleaning approaches are still in their infancy. Many of them simply employ smoothing filters and can only work when the data is a sequence of numerical values, while Bleach focuses on more general cases where data can be both numerical and categorical.

Jeffery et al. [130] focus on cleaning RFID data streams which suffer from low read rates, frequently failing to read tags that are present. Rather than using static smoothing filters that interpolate for lost readings, they propose SMURF, a declarative, adaptive smoothing filter for RFID data. SMURF does not require the application to set a smoothing window size, as it automatically adapts its window size based on the characteristics of the underlying data stream. However, smoothing filters may seriously alter the data without preserving the original information. With the minimum change principle in data cleaning, those originally clean data should not be changed in the data cleaning process.

SCREEN [105] is a constraint-based approach for cleaning stream data, which is based on the widely existing constraints on the speed of data changes, such as fuel consumption per hour or daily limit of stock prices. To support online stream computation, SCREEN achieves the local optimum in a window rather than the global optimum over the entire data stream.

Sequential data cleaning [131] is statistical based cleaning method, which models the likelihood of a repair by observing its speed changes. The paper shows that the speed constraint-based approach either does not precisely repair large spike errors or simply ignore small errors. The motivation of this work is that the speed constraint-based approach either does not precisely repair large spike errors or simply ignore small errors. Since the speed of data changes should not change significantly in a time point, the likelihood-based cleaning aims to maximize the likelihood of speed changes, instead of minimizing the changes as in the speed constraint-based cleaning.

In Spark Streaming [124], a data stream can be cleaned by joining it with precomputed information. However, precomputed information is not always available nor sufficient for accurate data cleaning. To the best of our knowledge, Bleach is the first stream data cleaning system based on data quality rules providing both high accuracy and low latency.

There are many other research work about data cleaning. For example, [108] and [109] are about how to perform data cleaning via knowledge base and crowdsourcing. BART [100] is a dirty data generator for evaluating data-cleaning algorithms. [118] studies the problem of temporal rules discovery for dirty web data. All such works are orthogonal to ours.

4.10 Conclusion

This work introduced Bleach, a novel stream data cleaning system, that aims at efficient and accurate data cleaning under real-time constraints.

First, we have introduced the design goals and the related challenges underlying Bleach, showing that stream data cleaning is far from being a trivial problem. Then we have illustrated the Bleach system design, focusing both on data quality (e.g., Bleach dynamic rule sets and stateful approach to windowing) and on systems aspects (e.g., data partitioning and coordination), which are required by the distributed nature of Bleach. We also have provided a series of optimizations to improve system performance, by using compact and efficient data structures, and by reducing the messaging overhead.

Finally, we have evaluated a prototype implementation of Bleach: our experiments showed Bleach achieves low-latency and high cleaning accuracy, while absorbing a dirty data stream, despite rule dynamics. We also have compared Bleach to a baseline system built on the micro-batch paradigm, and explained Bleach superior performance.

Our plan for future works is to support a more varied rule set and to explore alternative repair algorithms, that might require revisiting the inner data structures we use in Bleach.

Chapter 5

Conclusion and Future Work

In this thesis, we studied the problem of how to accelerate data preparation process for big data analytics, and provided efficient techniques. Due to the complexity of data preparation, we targeted two main steps in data preparation, data loading and data cleaning, and respectively designed and implemented two systems, DiNoDB and Bleach, both of which can help data scientists significantly reduce their time spent on data preparation.

First, we introduced DiNoDB, a distributed system tuned for interactive-speed queries on data files generated by large-scale batch processing frameworks. DiNoDB avoids data loading without loosing efficiency. It seamlessly integrates batch processing systems with a distributed, fault-tolerant and scalable interactive query engine. It uses a decorator mechanism that enhances the standard Hadoop I/O API and piggybacks the creation of auxiliary metadata required for interactive-speed query performance. Our extensive experimental evaluation demonstrated that DiNoDB outperforms other SQL-on-Hadoop solutions for a wide range of ad-hoc analytical workloads. In addition, the decorator mechanism can also be leveraged by other systems besides DiNoDB, which demonstrates that this is a general idea for accelerating data preparation.

Second, we presented Bleach, a novel stream data cleaning system. Unlike other data cleaning systems which mainly focus on batch data cleaning, Bleach performs data cleaning directly on data streams without waiting for all the data to be acquired. Bleach aims to achieve efficient and accurate qualitative data cleaning under real-time constraints. It relies on efficient, compact and distributed data structures to maintain the necessary state to repair data, using an incremental version of the equivalence class algorithm. Our evaluation showed the superior performance of Bleach compared with a baseline system built on the micro-batch paradigm, which indicates that streaming data cleaning is effective in accelerating data preparation.

5.1 Future Work

In this Section, taking our existing works as starting points, we discuss two interesting on-going works which also aim at optimizing the data preparation process.

5.1.1 Stream Holistic Data Cleaning

Besides equivalence class algorithm, other existing data cleaning algorithms can also be plugged in our stream data cleaning system. As the future work, we plan to use a stream version of the holistic data cleaning algorithm in Bleach to support a more varied rule set. To avoid ambiguity, we refer to Bleach with stream holistic data cleaning algorithm as *Bleach-hd*, while Bleach with stream equivalence class algorithm is still referred to as *Bleach*. We first give some background knowledge about the holistic data cleaning algorithm and then discuss the basic design and the optimization challenges in Bleach-hd.

The holistic data cleaning algorithm, proposed in [102], is designed to have a unified cleaning approach. It supports a general kind of rule, *Denial Constraints*, in which FD and CFD rules are both covered. DCs provide a declarative specification of violation rules. A DC φ on a schema R is defined as: $\forall t_\alpha, t_\beta, t_\gamma, ... \in R, \neg(P_1 \wedge ... \wedge P_m)$, where each *predicate* P_i is of the form $v_1 \theta v_2$ or $v_1 \theta c$ with $v_1, v_2 \in t_x.A$, $x \in \alpha, \beta, \gamma, ...$, $A \in R$, c is a constant in the domain of A, and $\theta \in =, <, >, \neq, \leq, \geq$. A relation instance I of R is said to satisfy a DC φ, denoted by $I \models \varphi$, if for every ordered list of tuples $\forall t_\alpha, t_\beta, t_\gamma, ... \in I$, at least one of P_i is false [6].

As an example, given a data stream with schema $S(A, B, C, D)$, a FD rule $r_1 : (A \rightarrow B)$ can be expressed with the following DC:

$$r_1 : \neg(t, t', (t(A) = t'(A)), (t(B) \neq t'(B))).$$

The holistic data cleaning algorithm builds a violation graph called *Conflict Hypergraph* (CH) to present all the detected violations from the dataset. Each node is a cell involved in at least one violation and each hyperedge presents a violation including a set of cells participating in this violation. A cell participating in multiple violations is contained in multiple hyperedges. Then, the minimum vertex cover for the conflict hypergraph, which contains the cells that are mostly likely to be wrong, is found. With the repair requirements of these cells, the cleaning system modifies the values of these cells so that all the violations will be resolved.

Bleach-hd, to achieve stream data cleaning, keeps the dynamic conflict hypergraph in the repair module. Similar to the violation graph in Bleach, the conflict hypergraph in

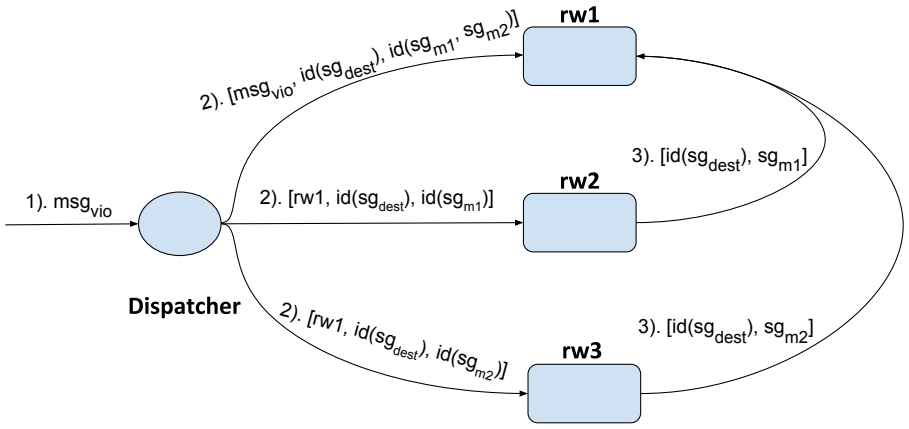


Figure 5.1 – An example with the repair module in Bleach-hd

Bleach-hd is divided into multiple subgraphs according to the connectivity. For each subgraph, Bleach-hd maintains the minimum vertex cover (the cells that are mostly likely to be wrong). Whenever a new violation of the current tuple is detected, Bleach-hd updates the relevant subgraphs (merging subgraphs if necessary) and their minimum vertex covers. If the minimum vertex covers include the cells from the current tuple, Bleach-hd modifies the values of the dirty current cells and outputs the repaired current tuple.

Bleach-hd is designed as a distributed stream data cleaning system to process large amounts of data. Hence, the conflict hypergraph is partitioned in all repair workers (RWs) in the repair module. Unlike Bleach, Bleach-hd distributes the conflict hypergraph based on subgraphs that each subgraph is only stored in one RW. Bleach-hd uses a component called *dispatcher* to distribute violation messages to the RWs. The dispatcher keeps two mappings. One is between cells and subgraph IDs, the other is between subgraph IDs and RWs. With these two mapping, the dispatcher chooses the correct RW to send the violation message if the RW has the subgraph which intersects with the violation message. If more than one subgraphs in different RWs intersect with the violation message, these subgraphs need to be merged into a single subgraph and the violation message is added to the merged subgraph. We define a subgraph to be in the *blocking state* if the dispatcher notifies that the subgraph needs to merge with other subgraphs to process a specified violation message. The subgraph leaves the blocking state when it has merged with all necessary subgraphs. A subgraph in the blocking state can neither be used to make the repair decision nor migrate to other RWs.

Figure 5.1 gives an example where the violation message msg_{vio} intersects three subgraphs sg_{dest}, sg_{m1} and sg_{m2} which are in three different RWs $rw1$, $rw2$ and $rw3$, respectively. The procedure is as following:

1) The dispatcher chooses sg_{dest} as the subgraph that msg_{vio} is added to so that sg_{m1} and sg_{m2} should be merged into sg_{dest}. Then, the dispatcher updates its local mappings.

2) Next, the dispatcher sends to $rw1$ a message including the violation message msg_{vio}, the ID of the subgraph to add msg_{vio}, $id(sg_{dest})$, and the IDs of the other subgraphs to be merged, $id(sg_{m1}, sg_{m2})$; also, the dispatcher sends messages to $rw2$ ($rw3$) instructing that the subgraph sg_{m1} (sg_{m2}) should be merged into sg_{dest} in $rw1$.

3) After merging sg_{dest} with received sg_{m1} and sg_{m2}, $rw1$ makes the repair decision for the current cells contained in msg_{vio}.

The Optimizations are required to improve the performance of Bleach-hd. As the subgraphs may be transferred among RWs, it is essential to compact the conflict hypergraph. Besides the windowing operation which prevents the conflict hypergraph to grow infinitely, we plan to support the super cell optimization, in which the cells conflict with the same set of cells would be compressed into a super cell. The challenge is that the super cells are also dynamic: a current cell might be added to a super cell when it has conflicts with the same set of cells as the other cells in the super cell, and a super cell should be split if any of its compressed cells are detected to have more violations than others.

5.1.2 Unified Stream Decorators

Our DiNoDB I/O decorators currently support batch processing systems such as Apache Hadoop and Apache Spark. To accelerate the analysis of the output data from these systems, DiNoDB I/O decorators piggyback the generation of auxiliary metadata, e.g., indexes and statistics. All algorithms used in the DiNoDB I/O decorators to generate additional metadata must be one-pass algorithms so that the overhead of the DiNoDB I/O decorators remain negligible in the batch jobs.

Considering another important category of data sources, data streams, we find that it is also possible to implement the idea of DiNoDB I/O decorators in stream processing systems, building stream metadata decorators. Due to the nature of data stream, stream processing systems process data in one single pass, that perfectly matches the requirement of the DiNoDB I/O decorators. Therefore, as the future work, we plan to extend the DiNoDB I/O decorators in streaming systems.

All kinds of DiNoDB I/O decorators we discussed in Chapter 3 can be developped in streaming systems, so that metadata streams including indexes, statistics and data samples can also be produced along with the output data stream of the stream processing systems by the stream metadata decorators.. As well as the metadata files produced in decorators in batch processing, the metadata streams can also be used later to accelerate the process of data analysis. Note that the statistics decorator in the batch pro-

Figure 5.2 – The unified stream decorators

cessing output the final statistics until all the data is processed. Nevertheless, in the stream processing, the data in data streams can never be completely processed because a data stream produce an unbounded amount of data. Hence, the statistics decorator in streaming system output the statistics periodically, reporting for the data processed so far. When the windowing operations exist in the stream processing system, the statistics decorator can also output statistics for data divided in each window.

Since it is a general idea that the generation of the metadata streams can be piggybacked in stream data processing systems, we plan to implement stream metadata decorators in multiple stream processing systems, e.g., Apache Storm, Apache Flink and even Apache Kafka with its new feature, Kafka Streams.

Besides generating metadata, we also consider to make the stream decorators have more functionalities, such as data cleaning. As we discussed in Chapter 4 through the example of Bleach, stream data cleaning can achieve high cleaning accuracy under real-time constraints, dramatically reducing the latency from data capture to data analysis. It would be very interesting to integrate the stream cleaning system, with the stream metadata decorators. Namely, given a data source in data stream, we plan to build a model of unified stream decorators as shown in Figure 5.2, which will not only output a cleaned data stream after performing stream data cleaning, but also generate metadata streams including metadata which could be used later in data analysis.

Appendix A

Summary in French

A.1 Introduction

A.1.1 Contexte

Nous vivons à l'ère du déluge de données, où de vastes quantités de données sont générées. Les données apportent d'énormes valeurs et avantages, non seulement dans des domaines scientifiques comme l'astronomie ou la biologie, mais aussi dans des domaines qui sont étroitement liés à notre vie quotidienne, comme le commerce électronique et le transport. Les entreprises peuvent utiliser leurs données collectées pour prendre en charge les décisions humaines, découvrir les besoins des clients et créer de nouveaux modèles commerciaux. Avec l'amélioration technologique au cours des dernières décennies, des ensembles de données massifs peuvent être stockés à faible coût. Par conséquent, de plus en plus d'entreprises commencent à stocker autant de données qu'ils pourraient collecter. Cependant, la conversion des données en connaissances précieuses est encore une tâche difficile.

Les systèmes de gestion de base de données relationnelle (SGBD relationnels) étaient les outils de taille unique pour l'analyse de données au siècle dernier [1]. Dans leur modèle relationnel, toutes les données sont représentées en termes de tuples, regroupées en un ensemble de tables qui sont liées les unes aux autres. Chaque table a un schéma prédéfini que tous les tuples de la table doivent suivre. Au fil des ans, les SGBD relationnels ont soutenu avec succès un grand nombre d'applications centrées sur les données avec des fonctionnalités et des exigences très différentes.

Cependant, au moment où nous entrons au 21ème siècle, le SGBD traditionnel devient un ajustement médiocre dans de nombreux scénarios d'application. L'une des principales raisons pour lesquelles le SGBD traditionnel est obsolète est l'augmentation exponentielle des données. Google, en tant qu'entreprise basée sur les données, traite plus de

3,5 milliards de demandes par jour et stocke plus de 10 exabytes de données. Facebook collecte 600 téraoctets de données par jour, dont 4,3 milliards de contenus, 5,75 milliards "like" et 350 millions de photos. Certaines estimations suggèrent que globalement au moins 2,5 quintiles d'octets de données sont produits tous les jours et 40 zettabytes de données existent d'ici 2020 [3]. De telles quantités énormes sont bien au-delà de la capacité de tout SGBD traditionnel. Nouvelles données efficaces et puissantes des outils d'analyse sont nécessaires pour faire face au grand défi de données.

Ce besoin a attiré l'attention du milieu universitaire et de l'industrie. Par conséquent, ces dernières années, les systèmes modernes d'analyse de données à grande échelle se sont développés, qui apportent d'énormes techniques innovantes et des optimisations. Ces systèmes visent à accélérer les procédures d'analyse de données pour les grands ensembles de données. Cependant, tel que rapporté par de nombreux scientifiques de données [29], seulement 20% de leur temps est passé à effectuer les tâches d'analyse de données souhaitées. Les scientifiques de données doivent passer 80% du temps, parfois même plus, sur préparation de données qui est un processus lent, difficile et fastidieux. Néanmoins, la préparation des données, étape essentielle avant d'effectuer l'analyse des données, n'a pas reçu suffisamment d'attention malgré son importance.

A.1.2 Data Preparation

La préparation des données, également appelée prétraitement des données, se concentre sur la détermination des données, l'amélioration de la qualité des données, la standardisation de la définition et de la structuration des données, collecter et consolider des données, et transformer les données pour qu'elles soient utiles, en particulier pour l'analyse [30]. En bref, la préparation des données augmente la valeur de l'analyse des données. Il comprend les étapes d'accès, de recherche, d'agrégation, d'enrichissement, de transformation, de nettoyage et de chargement des données.

Sans la préparation appropriée des données, l'analyse des données peut générer des résultats trompeurs si les ensembles de données sous-jacents sont sales. Un projet d'analyse peut échouer en raison de problèmes liés à la sécurité et à la confidentialité, si son ensemble de données est négligemment préparé sans cacher des informations sensibles. Les systèmes d'analyse de données modernes exigent tous la préparation des données comme étape préliminaire car ils ne sont pas capables de récupérer des informations à partir de données brutes, à moins que les données ne contiennent des formats appropriés ou chargé dans les systèmes. Par conséquent, la préparation des données est particulièrement cruciale pour le succès de l'analyse des données.

Dans de nombreuses organisations, la préparation des données nécessite des efforts manuels en utilisant des processus difficiles à partager ou même répétés. De plus en plus de scientifiques de données consacrent trop de temps à la préparation des données

et ne peuvent pas avoir suffisamment de temps pour résoudre d'autres problèmes de données difficiles. Par conséquent, le plus gros problème avec la préparation des données est qu'il coûte beaucoup de temps et coûte cher.

Comme nous vivons dans un grand monde des données, avec le volume croissant et la variété des données ces dernières années, la préparation des données est devenue plus exigeante et devient plus longue. Pendant ce temps, les données continuent d'être générées à une vitesse plus élevée. Les besoins commerciaux doivent commencer à exiger des intervalles plus courts et plus courts entre le moment où les données sont collectées et le moment où les résultats de l'analyse des données sont disponible pour la prise de décision manuelle ou algorithmique. Les chercheurs de données souhaitent avoir la possibilité d'analyser des jeux de données dès que possible, par exemple, quelques secondes après la collecte des jeux de données. Lorsque les données brutes sont générées en continu à partir d'un flux de données, les scientifiques de données peuvent même vouloir effectuer leur analyse de données de façon incrémentale en temps réel, plutôt que d'attendre que toutes les données soient acquises. Être capable de prendre des décisions en temps opportun est devenu de plus en plus crucial.

De toute évidence, le processus de préparation de données lent mais indispensable devient l'obstacle pour prendre des décisions en temps opportun. Pour surmonter cet obstacle, dans cette thèse, nous étudions le problème de l'accélération de la préparation des données pour la grande analyse des données. En particulier, nous nous concentrons sur deux sortes d'opérations coûteuses de préparation de données, de chargement de données et de nettoyage de données.

A.1.2.1 Chargement des Données

Le chargement des données consiste à copier et à charger des données d'une source de données vers un entrepôt de données ou tout autre système de stockage cible. C'était déjà un concept populaire dans les années 1970, comme la dernière étape du procédé ETL (Extract, Transform, Load) bien connu dans l'utilisation de la base de données. Le chargement des données peut également inclure l'application de la vérification des contraintes définies dans le schéma de la base de données, telles que l'unicité, l'intégrité référentielle et les champs obligatoires.

De nos jours, le chargement des données n'est pas seulement une opération existant dans les bases de données traditionnelles Mais devient également une étape principale dans de nombreux systèmes d'analyse de données modernes. Pour charger les données brutes, ces systèmes convertissent souvent les données en certains formats de données spécifiques, comme un format de données basé sur une colonne, ou chargent complètement les données en mémoire. Outre la modification de la disposition des données, le processus de chargement des données peut également générer des informations cumulatives

supplémentaires, telles que les statistiques et les index des données, qui peuvent être utilisés pour optimiser les exécutions de requêtes ultérieures. Bien que certains systèmes prétendent avoir la capacité de traiter des données in situ sans chargement, leur performance n'est pas satisfaisante. C'est parce qu'aucune métadonnée utile ou l'optimisation de la disposition des données ne sont possibles, ces systèmes ne peuvent traiter que les données brutes d'une force brute, par exemple, la numérisation à plusieurs reprises de fichiers entiers pour chaque requête.

Comme maintenant, les scientifiques veulent analyser les ensembles de données dès que possible, le chargement lent des données devient un goulet d'étranglement dans l'analyse des données. En particulier, lorsque nous analysons des ensembles de données temporaires, qui seront simplement abandonnés après l'exécution de quelques requêtes, le chargement des données ne convient pas. De toute évidence, nous avons besoin d'une approche qui évite le chargement lent des données, mais qui soit capable d'assurer une exécution efficace des requêtes.

A.1.2.2 Nettoyage des Données

À mesure que nous entrons dans un monde axé sur les données, l'application et le maintien de la qualité des données deviennent des tâches critiques. Selon un sondage auprès de l'industrie [31], plus de 25% des données critiques dans les meilleures entreprises du monde sont viciées. Sans un nettoyage correct des données, les problèmes liés à la qualité des données peuvent conduire à des résultats d'analyse trompeurs sur la base des «ordures dans les ordures ménagères». Par exemple, InsightSquared [32] prédit que des données sales entre les entreprises et le gouvernement couvre l'économie américaine de 3,1 billions de dollars par an.

Le nettoyage des données, également appelé nettoyage de données ou lavage de données, est le processus de détection, de correction ou de suppression, de données de données corrompues ou inexactes à partir d'un ensemble de données. Récemment, deux tendances majeures dans le nettoyage des données sont apparues. La première est une approche quantitative, appelée nettoyage quantitatif de données, qui est largement utilisé pour la détection des valeurs aberrantes en employant des méthodes statistiques pour identifier les comportements anormaux et les erreurs. La seconde est une approche logique, appelée nettoyage qualitatif des données. Le nettoyage qualitatif des données, d'autre part, repose sur la spécification des modèles ou des règles d'une instance de données légales. Il se compose de deux phases, une détection de violation qui consiste à identifier les données qui violent les motifs ou règles définis comme des erreurs, et une réparation de violation qui consiste à trouver un ensemble minimal de modifications qui corrigent les erreurs détectées.

Le nettoyage des données est considéré comme la tâche la plus longue dans la préparation des données, principalement parce qu'il implique souvent des calculs coûteux, tels que l'énumération de paires de tuples et la gestion des unions d'inégalité, qui sont difficiles à mettre à l'échelle des grands ensembles de données. La plupart des solutions de nettoyage de données existantes se sont concentrées sur le nettoyage de données par lots, en traitant des données statiques stockées dans un entrepôt de données, qui découragent les scientifiques de données d'avoir leur analyse de données en temps opportun. Il est devenu urgent d'élaborer de nouvelles solutions innovantes de nettoyage de données qui sont rapides et efficaces.

A.1.3 Contribution

L'objectif de cette thèse est de développer des systèmes avancés pour accélérer le processus de préparation de données qui nécessite beaucoup de temps pour les grandes analyses de données. En particulier, nous nous concentrons sur le chargement et le nettoyage des données. En tant que contributions, nous concevons et mettons en œuvre deux systèmes, DiNoDB, un moteur de requête à vitesse interactive pour les requêtes ad hoc sur des données temporaires qui évite le chargement de données en intégrant de manière transparente les systèmes de traitement par lots, et Bleach, un nouveau système de nettoyage de données de flux, qui vise à un nettoyage de données efficace et précis en contraintes en temps réel.

A.1.3.1 DiNoDB: un moteur de requête à vitesse interactive pour les données temporaires

La première contribution consiste à concevoir DiNoDB, un système SQL-on-Hadoop qui réalise une exécution de requête à vitesse interactive sans nécessiter de chargement de données. Les applications modernes impliquent de gros travaux de traitement par lots sur de gros volumes de données et, en même temps, nécessitent des analyses interactives ad hoc efficaces sur des données temporaires. Les solutions existantes, cependant, se concentrent généralement sur l'un de ces deux aspects, ignorant largement le besoin de synergie entre les deux. Par conséquent, les requêtes interactives nécessitent de charger l'ensemble de données qui ne peuvent fournir un retour d'investissement significatif que lorsque les données sont interrogées sur une longue période de temps.

En revanche, DiNoDB évite la phase de chargement et de transformation coûteuse qui caractérise à la fois les SGBD traditionnels et les solutions d'analyse interactive courante. Il est particulièrement adapté aux flux de travail modernes trouvés dans les cas d'utilisation tels que l'apprentissage machine et l'exploration de données, qui impliquent souvent des itérations de cycles d'analyse par lots et interactifs sur des données qui sont

généralement utiles pour une fenêtre de traitement étroite. L'innovation clé de DiNoDB consiste à transférer sur la phase de traitement par lot la création de métadonnées que DiNoDB exploite pour accélérer les requêtes interactives.

Notre analyse expérimentale détaillée, tant sur les bases de données synthétiques que sur la vie réelle, démontre que DiNoDB réduit considérablement le temps de vision et atteint de très bonnes performances par rapport aux moteurs de requêtes distribués à la fine pointe de la technologie, tels que Hive, Stado, Spark SQL et Impala.

A.1.3.2 Bleach: un système de nettoyage de données de flux distribué

La deuxième contribution est un système de nettoyage de données de flux distribué, appelé Bleach. Les approches de nettoyage de données évolutives existantes se sont concentrées sur le nettoyage des données par lot, qui nécessite beaucoup de temps. Comme la plupart des sources de données sont désormais diffusées, comme les fichiers journaux générés dans les serveurs Web et les achats en ligne, nous nous efforçons d'effectuer le nettoyage des données directement sur les flux de données. En dépit de la popularité croissante des systèmes de traitement des courants, aucune technique de nettoyage de données qualitatives n'a été proposée jusqu'ici. Dans cette thèse, nous comblons cette lacune en abordant le problème du nettoyage de données de flux basé sur des règles, qui définit des exigences strictes en termes de latence, de dynamique des règles et de capacité à faire face à la nature continue des flux de données.

Nous concevons Bleach, un système de nettoyage de données de flux qui réalise une détection de violation en temps réel et une réparation de données sur un flux de données sale. Bleach s'appuie sur des structures de données efficaces, compactes et distribuées pour maintenir l'état nécessaire pour réparer les données. En outre, il prend en charge la dynamique des règles et utilise une opération de fenêtre coulissante "cumulative" pour améliorer la précision de nettoyage.

Nous évaluons Bleach à l'aide de flux de données synthétiques et réels et validons expérimentalement son débit élevé, sa faible latence et sa haute précision de nettoyage, qui sont préservés même avec une dynamique de règles. En l'absence d'une ligne de base de nettoyage de flux comparable existante, nous avons comparé Bleach à un système de base fondé sur le paradigme du micro-lot, et montrent expérimentalement les performances supérieures de Bleach.

A.2 DiNoDB: un moteur de requête à vitesse interactive pour les données temporaires

A.2.1 Introduction

Au cours des dernières années, les systèmes modernes d'analyse de données à grande échelle se sont développés. Par exemple, des systèmes tels que Hadoop et Spark [62, 69] se concentrent sur les problèmes liés à la tolérance aux pannes et exposent un modèle de programmation parallèle simple mais élégant qui cache les complexités de la synchronisation. De plus, la nature orientée vers le lot de ces systèmes a été complétée par des composants supplémentaires (par exemple, Storm et Spark streaming [70, 72]) qui offrent des analyses (proches) en temps réel sur les flux de données. La communion de ces approches est maintenant connue sous le nom de "Lambda Architecture" (LA) [73]. En fait, LA est divisée en trois couches, i) le *batch layer* (basé sur, par exemple, Hadoop / Spark) pour la gestion et le prétraitement des données brutes ajoutées uniquement, ii) le *speed layer* (par exemple, Storm/Spark streaming) adapté aux analyses sur les données récentes tout en obtenant une faible latence en utilisant des algorithmes rapides et incrémentaux, et iii) le *serving layer* (par exemple, Hive [63], Spark SQL [69], Impala [36]) qui expose les vues par lots pour prendre en charge les requêtes ad hoc écrites en SQL, avec une faible latence.

Le problème avec de tels systèmes d'analyse existants à grande échelle est double. Tout d'abord, la combinaison de composants (couches) de différentes piles, bien que souhaitable, soulève des problèmes de performance et n'est parfois même pas possible dans la pratique. Par exemple, les entreprises qui possèdent une expertise dans, par exemple, Hadoop et les SGBD traditionnels (distribués) basés sur SQL, voudraient peut-être exploiter cette expertise et utiliser Hadoop comme couche de traitement par lots et SGBD dans la couche de service. Cependant, cette approche nécessite une phase de transformation/charge coûteuse, par exemple, déplacer des données de HDFS de Hadoop et la charger dans un SGBD [88], ce qui pourrait être impossible à amortize, en particulier dans les scénarios avec une fenêtre de traitement étroite, *i.e.,* lorsque vous travaillez sur *données temporaires* qui peuvent être simplement abandonnées après l'exécution de quelques requêtes.

Deuxièmement, bien que de nombreux systèmes SQL-on-Hadoop aient émergé récemment, ils ne sont pas bien conçus pour les requêtes ad-hoc (à courte durée de vie), en particulier lorsque les données restent dans son format natif, non compressé, tel que les fichiers CSV basés sur le texte. Pour atteindre des performances élevées, ces systèmes [35] préfèrent convertir les données en leur format de données spécifique en colonne, par exemple ORC [87] et Parquet [86]. Cela fonctionne parfaitement lorsque les données et les requêtes analytiques (c'est-à-dire la charge de travail complète) sont

dans leur phase finale de production. À savoir, ces formats de données auto-décrivant et optimisés jouent un rôle croissant dans l'analyse de données moderne, et cela devient particulièrement vrai une fois que les données ont été nettoyées, que les requêtes ont bien été conçues et que les algorithmes d'analyse ont été réglés. Cependant, lorsque les utilisateurs effectuent des tâches d'exploration de données et un réglage d'algorithme, c'est-à-dire lorsque les données sont *temporaire*, le format de données d'origine reste généralement inchangé — dans ce cas, l'optimisation prématurée du format de données est généralement évitée, les formats basés sur les formats CSV et JSON sont préférés. Dans ce cas, les systèmes d'analyse de données intégrés actuels peuvent sous-performer. Notamment, ils ne parviennent souvent pas à utiliser des techniques anciennes pour optimiser la performance des SGBD (distribués), e.x., indexation, qui n'est généralement pas prise en charge.

En résumé, les scientifiques de données contemporains font face à une large variété d'approches concurrentes ciblant le *batch* et la couche *serving*. Néanmoins, nous pensons que ces approches ont souvent une orientation trop stricte, souvent ignorées, ce qui ne permet pas d'explorer les avantages potentiels de l'apprentissage mutuel.

Dans cette section, nous proposons DiNoDB, un moteur de requête à vitesse interactive qui répond aux problèmes ci-dessus. Notre approche repose sur une intégration transparente des systèmes de traitement par lot (par exemple, Hadoop MapReduce et Apache Spark) avec un moteur de requête interactif réparti, tolérant aux pannes et évolutif pour les analyses *in-situ* sur les données *temporary*. DiNoDB intègre le traitement par lot avec la couche de service, en étendant l'ubiquité Hadoop I/O API en utilisant *DiNoDB I/O décorateurs*. Ce mécanisme est utilisé pour créer, en tant que sortie supplémentaire du traitement par lots, une large gamme de structures de données auxiliaires *metadata*, *ie*, telles que les cartes de position et les index verticaux, que DiNoDB utilise pour accélérer l'analyse de données interactives des fichiers de données *temporary* pour l'exploration de données et l'ajustement d'algorithme. Notre solution regroupe efficacement le traitement par lots et la couche de service pour les grands flux de données, tout en évitant tout chargement et données des coûts de mise en forme. Bien que, clairement, aucune solution d'analyse de données ne puisse correspondre à tous les grands cas d'utilisation de données, lorsqu'il s'agit de requêtes interactives ad hoc avec une fenêtre de traitement étroite, DiNoDB surpasse les moteurs de requêtes distribués à la fine pointe de la technologie, tels que Hive, Stado, Spark SQL et Impala.

En résumé, nos principales contributions dans cette section sont les suivantes:

- La conception de DiNoDB, un moteur de requête interactif distribué. DiNoDB exploite les architectures multi-core modernes et fournit des capacités d'interrogation SQL basées sur SQL, efficaces, réparties, tolérantes aux pannes et évolutives pour des données temporaires. DiNoDB (Distributed NoDB) est la

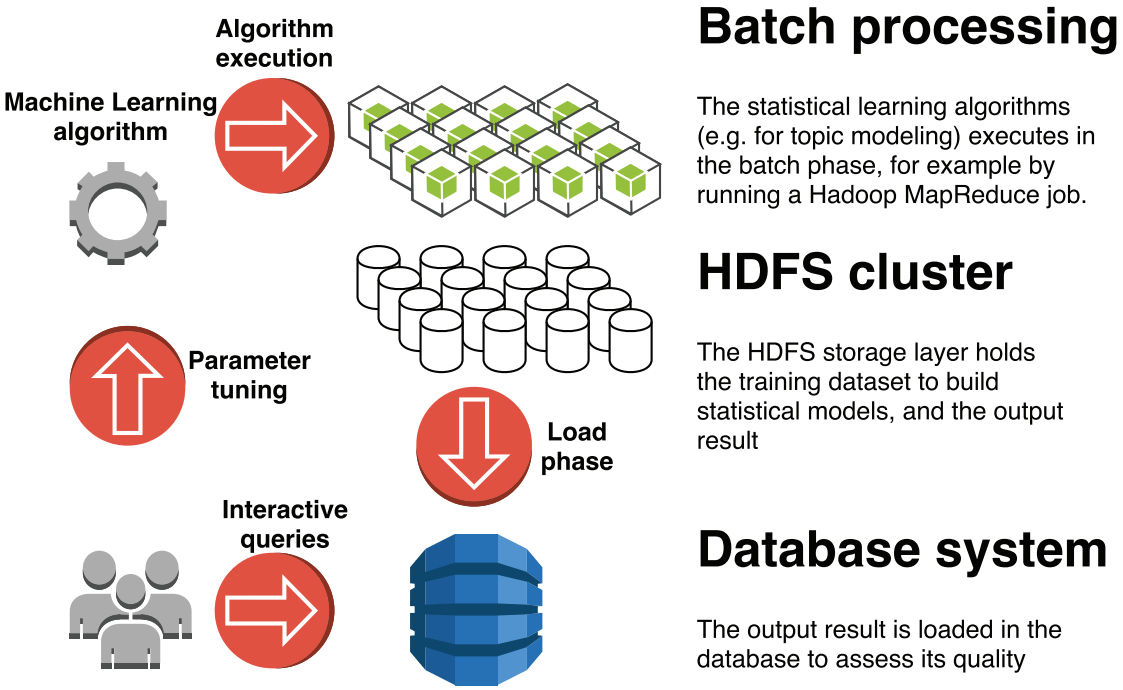


Figure A.1 – Cas d'utilisation de l'apprentissage de la machine.

première instanciation distribuée et évolutive du paradigme NoDB [41], qui était précédemment instancié uniquement dans les systèmes centralisés.

- Proposition et mise en œuvre de l'approche DiNoDB I/O decorators pour interagir le traitement par lots et les moteurs de services de requêtes interactifs dans un système d'analyse de données. DiNoDB I/O décorateurs génèrent, à la suite de la phase de traitement par lot, des métadonnées qui visent à faciliter et accélérer les requêtes interactives ultérieures.
- Évaluation détaillée du rendement et analyse comparative de DiNoDB versus systèmes à la fine pointe de la technologie, y compris Hive, Stado, Spark SQL et Impala.

A.2.2 Applications et cas d'utilisation

Dans cette section, nous examinons un cas d'utilisation contemporaine qui couvre le traitement par lots et les analyses interactives dans les flux d'analyse de données. Nous discutons: i) comment une meilleure communication entre le traitement par lots et la couche de service que DiNoDB apporte peut aider, et ii) l'applicabilité de notre approche d'analyse *données temporaires*.

Dans le cas d'utilisation, nous prenons la perspective d'un utilisateur (par exemple, un scientifique de données) axé sur un problème de cluster de données complexe. Plus précisément, nous considérons la tâche d'apprendre *topic models* [79], ce qui équivaut à regrouper automatiquement et conjointement des mots en "topics", et des documents en mélanges de sujets. Simplement dit, un modèle de sujet est un modèle bayésien hiérarchique qui associe à chaque document une distribution de probabilité sur "topics", qui sont à leur tour des distributions par mots. Ainsi, la sortie d'une analyse de données de modélisation de sujet peut être considérée comme une matrice de probabilités (éventuellement très grande): chaque ligne représente un document, chaque colonne d'un sujet et la valeur d'une cellule indique la probabilité qu'un document puisse couvrir un sujet particulier.

Dans un tel scénario, représenté dans la Figure A.1, l'utilisateur se heurte généralement aux problèmes suivants: i) les algorithmes de modélisation de sujets (par exemple, Collapsed Variational Bayes (CVB) [78]) nécessitent un réglage de paramètres, tel que En sélectionnant un nombre approprié de sujets, le nombre de fonctionnalités uniques à considérer, les facteurs de lissage de la distribution et beaucoup plus; et ii) le calcul de la «qualité de modélisation» nécessite généralement un processus d'essai et d'erreur selon lequel seuls les connaissances de domaine peuvent être utilisés pour discerner un bon regroupement d'un mauvais. En pratique, un tel scénario illustre un flux de travail typique de "développement" qui nécessite: a *phase de traitement par lots* (e.x., en cours d'exécution CVB), et *phase de requête interactive sur des données temporaires* (c'est-à-dire sur des données intéressantes Dans des périodes relativement courtes), et plusieurs itérations des deux phases jusqu'à ce que les algorithmes soient correctement réglés et les résultats finaux répondent aux attentes des utilisateurs.

DiNoDB s'attaque explicitement à ces flux de travail "développement". Contrairement aux approches actuelles, qui nécessitent généralement une phase de chargement de données longue et coûteuse qui augmente considérablement le temps de données à la perspicacité, DiNoDB permet d'interroger des données temporaires in situ et expose une interface SQL standard à l'utilisateur. Cela simplifie l'analyse des requêtes et révèle le principal avantage de DiNoDB dans ce cas d'utilisation, c'est-à-dire la suppression de la phase de chargement *données temporaires*, qui représente aujourd'hui l'un des principaux goulets d'étranglement opérationnels dans l'analyse des données. En effet, la phase de chargement de données traditionnelle est logique lorsque la charge de travail (c'est-à-dire les données et les requêtes) est stable à long terme. Cependant, comme le chargement des données peut inclure la création d'index, la sérialisation et l'analyse des frais généraux, il est raisonnable de remettre en question sa validité lorsque vous travaillez avec des données temporaires, comme dans notre cas d'utilisation d'apprentissage machine.

L'idée de conception clé derrière DiNoDB est celle de déplacer la partie du fardeau d'une opération de chargement traditionnelle vers la phase de traitement par lots d'un flux de travail "développement". Pendant le traitement des données par lots, DiNoDB accroche la création de cartes positionnelles réparties et d'index verticaux pour améliorer les performances des requêtes utilisateur interactives sur les données temporaires. Les requêtes interactives fonctionnent directement avec des fichiers de données temporaires générés par la phase de traitement par lots, qui sont stockés sur un système de fichiers distribué tel que HDFS [43].

A.2.3 DiNoDB conception de haut niveau

Dans cette section, nous présentons la conception d'architecture de haut niveau de DiNoDB. DiNoDB est conçu pour assurer une intégration transparente des systèmes de traitement par lots tels que Hadoop MapReduce et Spark, avec une solution distribuée pour l'analyse de données in situ sur de grands volumes de fichiers de données temporaires et brutes. Tout d'abord, nous expliquons comment DiNoDB étend Hadoop I/O API omniprésente en utilisant DiNoDB I/O decorators, un mécanisme qui génère une large gamme de structures de métadonnées auxiliaires pour accélérer l'analyse de données interactive à l'aide du moteur de requête DiNoDB. Ensuite, nous décrivons le moteur de requêtes DiNoDB, qui tire parti des métadonnées générées dans la phase de traitement par lot pour obtenir des performances de requête à vitesse interactive.

La phase de traitement par lot (par exemple, dans les cas d'apprentissage par machine et d'exploration de données décrits précédemment) implique généralement l'exécution d'algorithmes d'analyse (sophistiqués). Cette phase peut inclure un ou plusieurs travaux de traitement par lot, de sorte que les données de sortie sont écrites sur HDFS.

L'idée clé derrière DiNoDB est de tirer parti du traitement par lots en tant que phase de préparation pour les futures requêtes interactives. À savoir, DiNoDB enrichit Hadoop I/O API avec DiNoDB I/O decorators. Un tel mécanisme accroche la génération de métadonnées par *pipelining* les tuples de sortie produits par le moteur batch dans une série de *decorators* spécialisés qui stockent des métadonnées auxiliaires ainsi que les tuples de sortie d'origine.

En plus de la génération de métadonnées, DiNoDB capitalise sur le prétraitement des données en conservant les données de sortie dans la mémoire. Pour être plus précis, nous configurons Hadoop pour stocker les données de sortie et les métadonnées dans la RAM, en utilisant le système de fichiers `ramfs`comme point de montage supplémentaire pour HDFS.[1] Notre prototype DiNoDB prend en charge les deux ramfs et les points de montage

[1]Cette technique a été considérée de manière indépendante pour inclusion dans un patch récent à HDFS [76] et dans une alternative HDFS en mémoire récente appelée Tachyon [82]. Enfin, il est maintenant ajouté dans la dernière version d'Apache Hadoop [64]

sur disque pour HDFS, un choix de conception qui permet de répondre aux requêtes sur des données qui ne peuvent pas correspondre à la RAM .

Les données de sortie et les métadonnées sont consommées par le moteur de requêtes interactif DiNoDB. Le moteur de requête interactif DiNoDB est un moteur de traitement massivement parallèle qui orchestré plusieurs nœuds DiNoDB. Chaque nœud DiNoDB est une instance optimisée de PostgresRaw [41], une variante de PostgreSQL adaptée à l'interrogation de fichiers de données temporaires produits dans la phase de traitement par lots. Pour assurer des performances élevées et des temps d'exécution de requêtes faibles, nous co-localisons DiNoDB nœuds et HDFS DataNodes, où les deux partagent des données à travers HDFS, et en particulier grâce à sa mémoire, ramfs monter.

Dans le reste de ce section, nous supposons que les données brutes et les données *temporaires* ingérées et produites par la phase de traitement par lots et utilisées dans la phase de service de la requête sont dans un format de données textuelles structurées (par exemple, une valeur séparée par des virgules des dossiers).

A.2.4 DiNoDB I/O decorators

DiNoDB piggybacks la génération de métadonnées auxiliaires sur la phase de traitement par lots en utilisant DiNoDB I/O decorators. DiNoDB I/O decorators sont conçus pour être un mécanisme non intrusif, qui intègre parfaitement les systèmes prenant en charge Hadoop I/O API classique, comme Hadoop MapReduce et Apache Spark. DiNoDB I/O decorators opère à la fin de la phase de traitement par lots pour chaque tâche finale qui produit des tuples de sortie, comme le montre la Figure A.2. Au lieu d'écrire des tuples de sortie vers HDFS directement, en utilisant Hadoop I/O API standard, les tâches utilisent DiNoDB I/O decorators, qui créent un pipeline de génération de métadonnées, où chaque décorateur itère sur des flux de tuples de sortie et calculent les différents types de métadonnées. Actuellement, notre prototype prend en charge quatre types de métadonnées, *positional maps* [41], *vertical indexes*, *statistiques* et *samples de données*.

DiNoDB I/O decorators sont conçus pour être un mécanisme non intrusif, qui intègre parfaitement les systèmes prenant en charge Hadoop I/O API classique, comme Hadoop MapReduce et Apache Spark. DiNoDB I/O decorators opèrent à la fin de la phase de traitement par lot pour chaque tâche finale qui produit des tuples de sortie, comme le montre la Figure A.2. Au lieu d'écrire des tuples de sortie sur HDFS directement, en utilisant Hadoop I/O API standard, les tâches utilisent DiNoDB I/O decorators, qui créent un pipeline de génération de métadonnées, où chaque décorateur itère sur des flux de tuples de sortie et calculent les différents types de métadonnées décrites ci-dessus.

Pour utiliser DiNoDB I/O decorators, les utilisateurs de Hadoop doivent remplacer la classe Vanova Hadoop `OutputFormat` par un nouveau module appelé `DiNoD-`

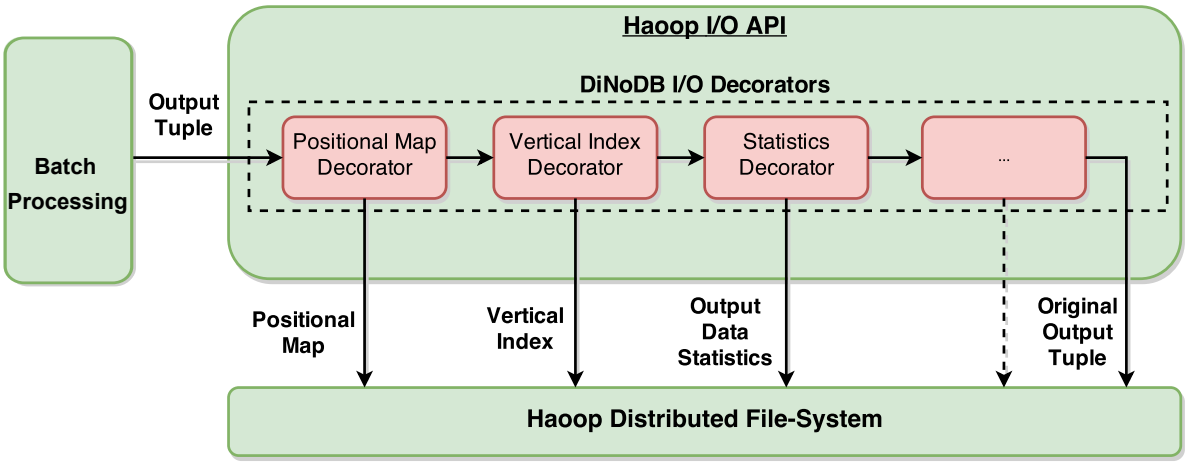


Figure A.2 – Présentation du DiNoDB I/O decorators.

BOutputFormat. Notre prototype supporte actuellement la sous-classe TextOutputFormat, qui permet à DiNoDB de fonctionner sur des formats de données textuels. Plus précisément, le module DiNoDBTextOutputFormat implémente une nouvelle classe DiNoDBArrayWritable qui est utilisée pour générer à la fois les données de sortie et ses métadonnées associées. Si les utilisateurs travaillent avec Spark, avant de sauvegarder leur résultat RDD sur HDFS (par méthode saveAsTextFile), ils doivent d'abord lancer ce résultat RDD dans un DiNoDBRDD, qui utilise en interne DiNoDBOutputFormat comme classe OutputFormat.

DiNoDB I/O decorators sont configurés en passant un fichier de configuration à chaque travail de traitement par lot dans Hadoop ou en définissant des paramètres de DiNoDBRDD dans Spark. Les utilisateurs spécifient les métadonnées à générer et indiquent des paramètres tels que le taux d'échantillonnage à utiliser pour la génération de cartes de position et les attributs clés pour la génération d'index verticaux.

A.2.5 Le moteur de requête interactif DiNoDB

À un niveau élevé (voir la figure A.3), le moteur de requêtes interactif DiNoDB se compose d'un ensemble de noeuds db, orchestrés à l'aide d'un cadre de traitement parallèle massivement (MPP). Dans notre implémentation de prototype, nous utilisons le framework MPD de Stado [14], qui intègre bien les moteurs de base de données PostgreSQL. DiNoDB assure la localisation des données en co-localisant DiNoDB nœuds avec DataNodes HDFS. Dans ce qui suit, nous décrivons d'abord le client DiNoDB et les nœuds DiNoDB.

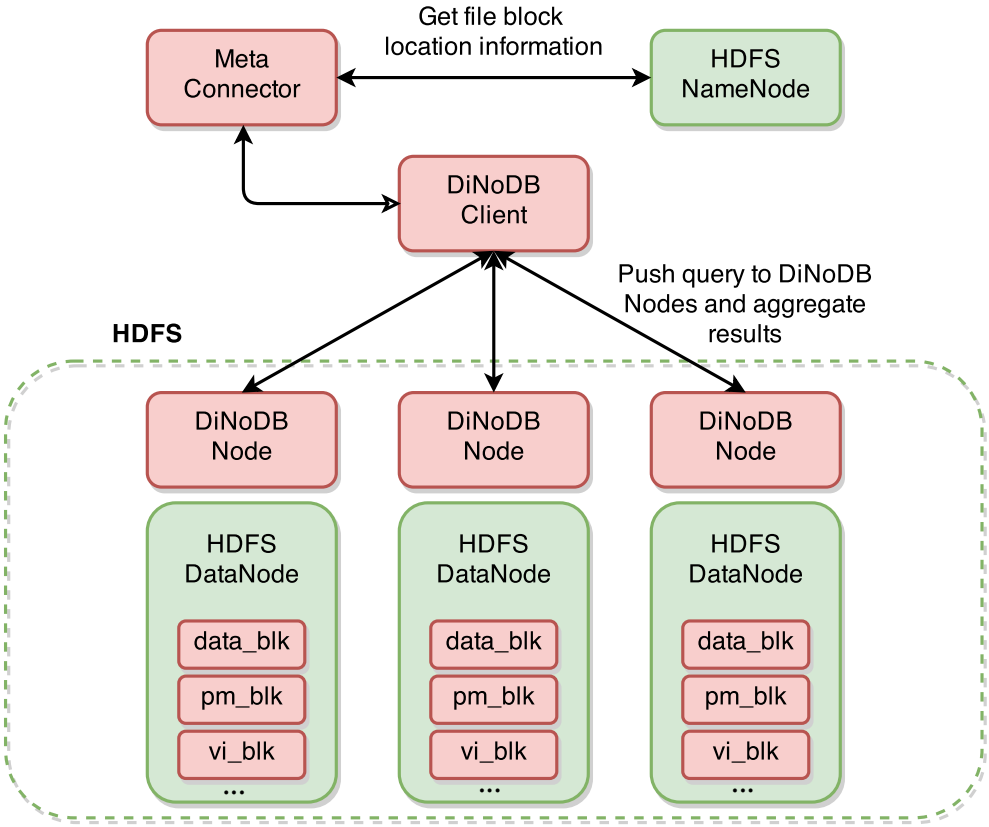


Figure A.3 – Architecture du moteur de requêtes interactif DiNoDB.

A.2.5.1 Clients DiNoDB

Un client DiNoDB sert de point d'entrée pour les requêtes interactives DiNoDB. Il fournit une interface de commande shell standard, cachant la disposition du réseau et l'architecture du système distribué à partir des utilisateurs. En tant que tel, les applications peuvent utiliser DiNoDB comme un SGBD traditionnel.

DiNoDB clients acceptent les requêtes d'application (requêtes) et communiquent avec les nœuds DiNoDB. Lorsqu'un client DiNoDB reçoit une requête, il récupère les métadonnées pour les "tables" (fichiers de sortie de la phase de lot) indiqués dans la requête en utilisant le module *MetaConnector*. Le MetaConnector (voir la figure A.3) fonctionne comme un proxy entre DiNoDB et le NameNode HDFS et est responsable de la récupération des informations de métadonnées HDFS comme des partitions et des emplacements de blocs de fichiers de données brutes. En utilisant les métadonnées HDFS, MetaConnector guide les clients DiNoDB pour interroger les nœuds DiNoDB qui contiennent des fichiers de données brutes en rapport avec les requêtes des utilisateurs. En outre, MetaConnector configure à distance les nœuds DiNoDB afin qu'ils puissent créer le mappage entre les "tables" et les blocs HDFS connexes, y compris tous les blocs de fichiers de données et blocs de métadonnées, par exemple, les blocs de positional maps et les blocs de

vertical indexes. En résumé, l'anatomie d'une exécution de requête est la suivante: i) en utilisant MetaConnector, un client DiNoDB apprend l'emplacement de chaque bloc de fichier brut et pousse la requête aux noeuds DiNoDB respectifs; ii) les nœuds DiNoDB traitent la requête en parallèle; et enfin, iii) le client DiNoDB agrège le résultat.

A.2.5.2 Nœuds DiNoDB

Les noeuds DiNoDB sont basés sur PostgresRaw [41], un moteur de requête optimisé pour les requêtes in situ. Nœuds DiNoDB instancent des bases de données personnalisées PostgresRaw qui exécutent des requêtes d'utilisateurs, et sont localisés avec des DataNodes HDFS. Dans la mise en œuvre de la vanilla PostgresRaw [41], une " table " correspond à un seul fichier de données. Étant donné que les fichiers HDFS sont divisés en plusieurs blocs, les nœuds DiNoDB utilisent un nouveau mécanisme de lecture de fichiers qui peut accéder à des données sur HDFS et mappe une " table " à une liste de blocs de fichiers de données. En outre, l'implémentation vanilla PostgresRaw est un serveur à processus multiples, qui forge un nouveau processus pour chaque nouvelle session client, avec des métadonnées individuelles et un cache de données par processus. Au lieu de cela, nœuds DiNoDB place les métadonnées et les données dans *shared memory*, de sorte que les requêtes des utilisateurs - qui sont envoyées via le client DiNoDB- peuvent en bénéficier dans plusieurs sessions.

Nœuds DiNoDB peut profiter du fait que les données sont naturellement partitionnées en blocs HDFS pour tirer parti des processeurs multi-core modernes. Par conséquent, les données et les métadonnées associées peuvent être facilement accessibles par plusieurs instances de PostgresRaw, pour permettre le parallélisme au niveau des nœuds. Les utilisateurs DiNoDB peuvent indiquer sélectivement si les fichiers de données brutes sont placés sur le disque ou en mémoire. Par conséquent, les nœuds DiNoDB peuvent bénéficier de manière transparente d'un système de fichiers sauvegardé par mémoire pour améliorer considérablement les temps d'exécution des requêtes. Nœuds DiNoDB peut profiter du fait que les données sont naturellement partitionnées en blocs HDFS pour tirer parti des processeurs multi-core modernes. Par conséquent, les données et les métadonnées associées peuvent être facilement accessibles par plusieurs instances de PostgresRaw, pour permettre le parallélisme au niveau des nœuds. Les utilisateurs DiNoDB peuvent indiquer sélectivement si les fichiers de données brutes sont placés sur le disque ou en mémoire. Par conséquent, les nœuds DiNoDB peuvent bénéficier de manière transparente d'un système de fichiers sauvegardé par mémoire pour améliorer considérablement les temps d'exécution des requêtes.

A.2.6 Conclusion

Dans ce travail, nous avons présenté l'architecture de DiNoDB, un système distribué pour les requêtes de vitesse interactive sur les fichiers de données temporaires générés par des cadres de traitement par lots à grande échelle. Comme le montre notre évaluation expérimentale approfondie, pour les caisses d'utilisation DiNoDB- requêtes ad hoc sur une fenêtre de traitement étroite, notre système surpasse les solutions actuelles SQL-on-Hadoop. DiNoDB utilise un mécanisme de décorateur qui améliore Hadoop I/O API standard et permet de créer des métadonnées auxiliaires requises pour la performance des requêtes à vitesse interactive. De plus, DiNoDB I/O decorators s'intègre parfaitement aux cadres existants et aux systèmes de stockage distribués.

Notre évaluation expérimentale, que nous faisons à la fois sur les bases de données synthétiques et sur le monde réel, souligne les principaux avantages de DiNoDB dans un certain nombre de cas d'utilisation importants, ce qui le rend adapté à une large gamme de charges de travail analytiques ad hoc.

A.3 Bleach: un système de nettoyage de données de flux distribué

A.3.1 Introduction

Aujourd'hui, nous vivons dans un monde où les décisions sont souvent basées sur des applications analytiques qui traitent des flux continus de données. En règle générale, les flux de données sont combinés et résumés pour obtenir une représentation succincte de ceux-ci: les applications analytiques s'appuient sur de telles représentations pour faire des prédictions et pour créer des rapports, des tableaux de bord et des visualisations [119, 122, 123]. Toutes ces applications s'attendent à ce que les données et leur représentation respectent certains critères de qualité. Les problèmes de qualité des données interfèrent avec ces représentations et faussent les données, entraînant des résultats d'analyse trompeuse et des décisions potentiellement mauvaises.

En tant que tel, une gamme de techniques de nettoyage de données ont été proposées récemment [99,127,128]. Cependant, la plupart d'entre eux se concentrent sur le nettoyage de données "batch", en traitant des données statiques stockées dans des entrepôts de données, qui prennent beaucoup de temps. Ils négligent la classe importante de données en continu. Dans ce section, nous abordons cette lacune et nous nous concentrons sur *flux de données de flux*. Le défi dans le nettoyage des flux est qu'il nécessite des garanties *en temps réel* ainsi que des exigences *accuracy* élevées, des exigences qui sont souvent en désaccord.

Une approche naïve du nettoyage des données est d'inclure des filtres statiques simples pour traiter des enregistrements sales, mais sa capacité de nettoyage est assez limitée. Une autre approche naïve pourrait simplement étendre les techniques de lot existantes, en tamponnant les enregistrements de données dans un magasin de données temporaire et en le nettoyant périodiquement avant de l'alimenter en composants descendants. Bien que susceptible d'atteindre une grande précision, une telle méthode viole clairement les exigences en temps réel des applications en continu. Le problème est exacerbé par le volume de systèmes de nettoyage de données qui doivent être traités, ce qui interdit les solutions centralisées. Par conséquent, notre objectif est de concevoir un *système de nettoyage de données de flux distribué*, qui réalise un nettoyage efficace et précis en temps réel.

Dans ce section, nous nous concentrons sur le nettoyage qualitatif des données, dans lequel un ensemble de règles spécifiques au domaine définit la façon dont les données doivent être nettoyées: en particulier, nous considérons les dépendances fonctionnelles (FD) et les dépendances fonctionnelles conditionnelles (CFD). Notre système, appelé Bleach, se déroule en deux phases: *violation detection*, pour trouver des violations de règles et *violation repair*, pour réparer les données en fonction de ces violations. Bleach s'appuie sur des structures de données efficaces, compactes et distribuées pour maintenir l'état nécessaire (par exemple, résumés des données passées) pour réparer les données, en utilisant un algorithme de classe d'équivalence incrémentale.

Nous abordons également les complications dues à la nature à long terme et dynamique des flux de données: la définition des données sales pourrait changer pour suivre une telle dynamique. Bleach prend en charge les règles dynamiques, qui peuvent être ajoutées et supprimées sans nécessiter de temps d'inactivité. En outre, Bleach implémente une opération de fenêtre coulissante qui détermine des exigences de stockage modestes supplémentaires pour stocker temporairement des statistiques cumulatives, pour augmenter la précision de nettoyage.

Notre évaluation de performance expérimentale de Bleach est double. Tout d'abord, nous étudions la performance, en termes de débit, de latence et de précision, de notre prototype et nous concentrons sur l'impact de ses paramètres. Ensuite, nous comparons Bleach à un système de base alternatif, que nous mettons en œuvre en utilisant une architecture de micro-émission par lots. Nos résultats indiquent les avantages d'un système comme Bleach, qui tient même avec la dynamique des règles. Malgré un travail approfondi sur le nettoyage des données fondées sur des règles [96, 99, 101, 102, 111, 113, 114, 118], nous ne connaissons aucun autre système de nettoyage de données de flux.

time

	item	category	clientid	city	zipcode
	...	...	...	...	...
t_1	MacBook	computer	11111	France	75001
t_2	bike	sports	33333	Lyon	null
t_3	Interstellar	movies	22222	Paris	75001
t_4	bike	toys	44444	Nice	06000
t_5	Titanic	movies	11111	Paris	null
	...	...	...	...	...

Figure A.4 – Exemple illustratif d'un flux de données composé de transactions en ligne.

A.3.2 Défis et buts

Un système de nettoyage de données de flux idéal devrait accepter un flux d'entrée sale D_{in} et produire un flux propre D_{out}, dans lequel tous *violations de règles* dans D_{in} sont réparés ($D_{out} \models \Sigma$). Cependant, cela n'est pas possible en réalité en raison de:

- **Contrainte en temps réel**: Comme le nettoyage des données est incrémental, la décision de nettoyage d'un tuple (réparer ou ne pas réparer) ne peut être faite que sur la base de ses propres tuples dans le flux de données, ce qui est différent du nettoyage de données dans les entrepôts de données où l'ensemble de données est disponible. En d'autres termes, si un tuple sale n'a que des violations avec des tuples ultérieurs dans le flux de données, il ne peut pas être nettoyé. Une mise à jour tardive pour un tuple dans le flux de données de sortie ne peut être acceptée.
- **Dynamic rules**: Dans un système de nettoyage de données de flux, l'ensemble de règles n'est pas statique. Une nouvelle règle peut être ajoutée ou une règle obsolète peut être supprimée à tout moment. Un tuple de données traité ne peut pas être nettoyé à nouveau avec un jeu de règles mis à jour. Le retraitement de l'ensemble du flux de données chaque fois que le jeu de règles est mis à jour n'est pas réaliste.
- **Données non consolidées**: Un flux de données produit une quantité illimitée de données, qui ne peuvent pas être stockées complètement. Ainsi, le nettoyage des données de flux ne peut se permettre d'effectuer un nettoyage sur l'historique complet des données. À savoir, si un tuple sale n'a que des violations avec des tuples qui apparaissent beaucoup plus tôt dans le flux de données, il est probable qu'un tel tuple ne soit pas nettoyé.

Considérons l'exemple dans la Figure A.4, qui est un flux de données de transactions en ligne. Chaque tuple représente un enregistrement d'achat, qui contient un article acheté ($item$), la catégorie de cet élément ($category$), un identifiant de client ($clientid$), la ville

du client ($city$) et le code postal de cette ville ($zipcode$). Dans l'exemple, nous montrons un extrait de cinq tuples de données du flux de données, de t_1 à t_5.

Maintenant, supposons que nous recevons deux règles FD et une règle CFD indiquant comment un flux de données propre devrait ressembler: (r_1) les mêmes éléments ne peuvent appartenir qu'à la même catégorie; (r_2) deux enregistrements avec le même $clientid$ doivent avoir la même ville; (r_3) deux enregistrements avec le même code postal non nul doivent avoir la même ville:

(r_1) $item \rightarrow category$

(r_2) $clientid \rightarrow city$

(r_3) $zipcode \rightarrow city, zipcode \neq null$

Dans notre exemple, il y a trois violations des règles r_1, r_2 et/ou r_3: ($v1$) t_1 et t_3 ont le même code postal non nul ($t_1(zipcode) = t_2(zipcode) \neq null$) mais différents noms de villes ($t_1(city) \neq t_2(city)$); ($v2$) t_2 les virements bancaires appartiennent à la catégorie sports alors que t_4 classe les vélos comme jouets ($t_2(item) = t_4(item), t_2(category) \neq t_4(category)$); et ($v3$) t_1 et t_5 ont le même $clientid$ mais différents noms de villes ($t_1(clientid) = t_5(clientid), t_1(city) \neq t_5(city)$).

Notez que lorsque un système de nettoyage de données de flux reçoit un tuple t_1, aucune violation ne peut être détectée, comme dans notre exemple t_1 n'a que des violations avec des tuples ultérieurs t_3 et t_5. Ainsi, aucune modification ne peut être faite sur t_1. En outre, le retard de la procédure de nettoyage pour t_1 n'est pas une option possible, non seulement en raison de contraintes en temps réel, mais aussi parce qu'il est difficile de prévoir la durée pendant laquelle ce tuple doit être mis en mémoire tampon pour qu'il soit nettoyé. Par conséquent, le nettoyage des données du flux doit être progressif: chaque fois qu'une nouvelle donnée arrive, le processus de nettoyage des données démarre immédiatement. Bien que l'exécution d'une détection de violation incrémentielle semble simple, la réparation de la violation progressive est beaucoup plus complexe à réaliser. Revenons à l'exemple de la Figure A.4, Supposons que le système de nettoyage de flux reçoit un tuple t_5 et qu'il détecte avec succès la violation v_3 entre t_5 et t_1. Une telle détection n'est pas suffisante pour prendre une décision de réparation correcte, car le tuple t_1 est également en conflit avec un autre tuple t_3. Une réparation supplémentaire dans le système de nettoyage de données de flux devrait également tenir compte des violations entre les tuples antérieurs.

Pour tenir compte des subtilités du processus de réparation des infractions, nous utilisons le concept de *violation graph* [99]. Un graphique de violation est une structure de données contenant les violations détectées, dans lesquelles chaque noeud représente une cellule. Si certaines violations partagent une cellule commune, elles seront regroupées en un seul *sous-graphique*. Par conséquent, le graphique de violation est divisé en petits sous-graphes indépendants. Une seule cellule ne peut être qu'un sous-graphe. Si deux

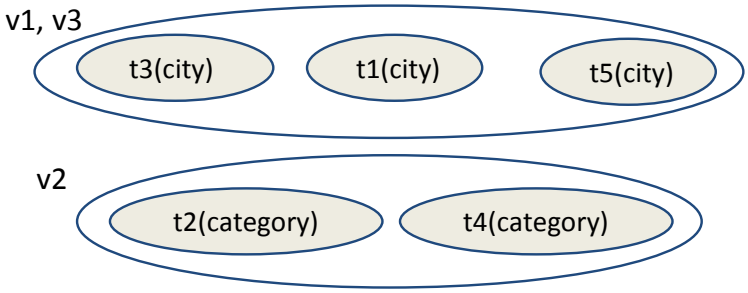


Figure A.5 – Un exemple d'un graphique de violation, dérivé de notre exemple en cours d'exécution.

sous-graphes partagent une cellule commune, ils doivent fusionner. La décision de réparation d'un tuple n'est pertinente qu'aux sous-graphes dans lesquelles ses cellules sont impliquées. Un graphique de violation pour notre exemple peut être vu dans la Figure A.5. Compte tenu de ce graphique de violation, pour prendre la décision de réparation pour le tuple t_5, le système de nettoyage ne peut se baser sur le sous-graphe supérieur qui consiste en violation v_1 et v_3 avec la cellule commune $t_1(city)$. Nous donnons maintenant notre déclaration de problème comme suit.

Statement du problème: Compte tenu d'un flux de données illimité avec un schéma associé[2] et un ensemble de règles *dynamique*, comment concevoir un système de nettoyage de données *incremental* et *en temps réel*, y compris des mécanismes de détection de violation et de réparation de violation, en utilisant des ressources informatiques et de stockage délimitées, pour produire un flux de données nettoyé?

Dans ce qui suit, nous examinons l'architecture Bleach et fournissons des détails sur ses composants. Comme le montre la Figure A.6, le flux de données d'entrée entre d'abord dans **module de détection**, ce qui révèle des violations par rapport à des règles définies. Le flux de données intermédiaire est enrichi d'informations de violation, que le **module de réparation** utilise pour prendre des décisions de réparation. Enfin, le système produit un flux de données nettoyé.

A.3.3 Détection de violation

Le module de détection de violation vise à trouver des tuples d'entrée qui enfreignent les règles. Pour ce faire, il stocke les tuples dans la mémoire, dans une structure de données efficace et compacte que nous appelons le *historique des données*. Les tuples d'entrée sont donc comparés à ceux de l'historique des données pour détecter les violations. La Figure A.7 illustre les éléments internes du module de détection: il consiste en

[2]Notez que bien que nous limitions le flux de données pour avoir un schéma fixe dans ce travail, il est facile d'étendre notre travail pour prendre en charge un schéma dynamique.

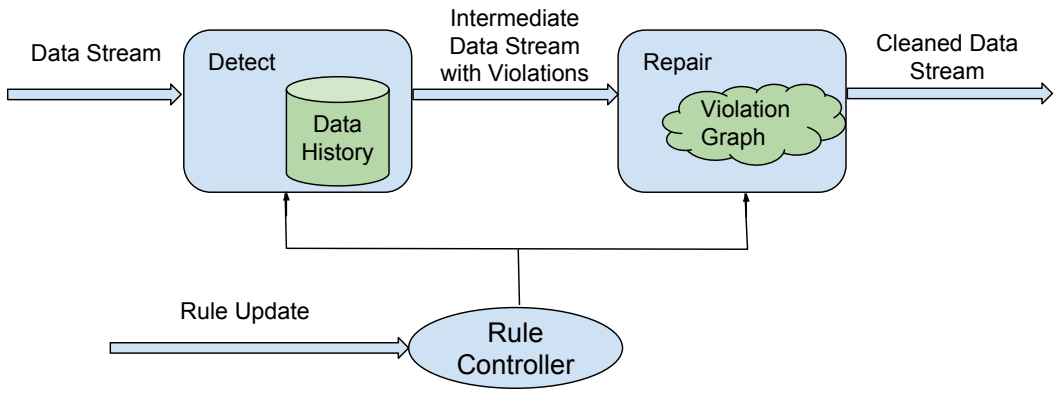


Figure A.6 – Nettoyage des données de flux Vue d'ensemble

un routeur d'entrée, un routeur de sortie et des détecteurs multiples (DW). Les règles de violation des cartes d'éclaircissement à ces DW: chaque travailleur est chargé de trouver des violations pour une règle spécifique.

Chaque DW reçoit une règle et reçoit les sous-éléments pertinents issus du flux d'entrée. Pour chaque sous-tuple, un DW effectue une opération de recherche dans l'historique des données et émet un message dans les composants descendants lorsqu'une violation de règle est détectée. Pour atteindre l'efficacité et la performance, les opérations de recherche doivent être rapides et le flux de données intermédiaire devrait éviter les informations redondantes.

Un DW accumule des sous-tuples d'entrée pertinents dans une structure de données compacte qui permet un processus de recherche efficace, ce qui le rend similaire à un mécanisme d'indexation traditionnel. Tout d'abord, pour accélérer le processus de recherche, les sous-tuples sont regroupés par la valeur de l'attribut LHS utilisé par une règle donnée: nous appelons un tel groupe a *groupe de cellules* (CG). Ensuite, pour obtenir une représentation de données compacte, toutes les cellules d'un CG partageant la même valeur RHS sont regroupées en *super cell* (SC). Les groupes de cellules sont stockés dans une carte de hash en utilisant leurs identifiants comme clés: donc, le DW trouve d'abord le CG correspondant au sous-tuple actuel. Les cellules dans les CG correspondants sont les seules cellules pouvant être en conflit avec la cellule actuelle. Dans l'ensemble, la complexité du processus de recherche pour un sous-tuple est $O(1)$.

DW génère un flux de données intermédiaire de *messages de violation*, qui aide les composants en aval pour éventuellement réparer les tuples d'entrée. L'objectif du DW est de générer autant de messages que possible, tout en permettant une réparation efficace des données. Lorsque le processus de recherche révèle que le tuple actuel ne viole pas une règle, les DW émettent un message de non-violation. Au lieu de cela, lorsqu'une violation est détectée, un DW construit un message avec toutes les informations nécessaires

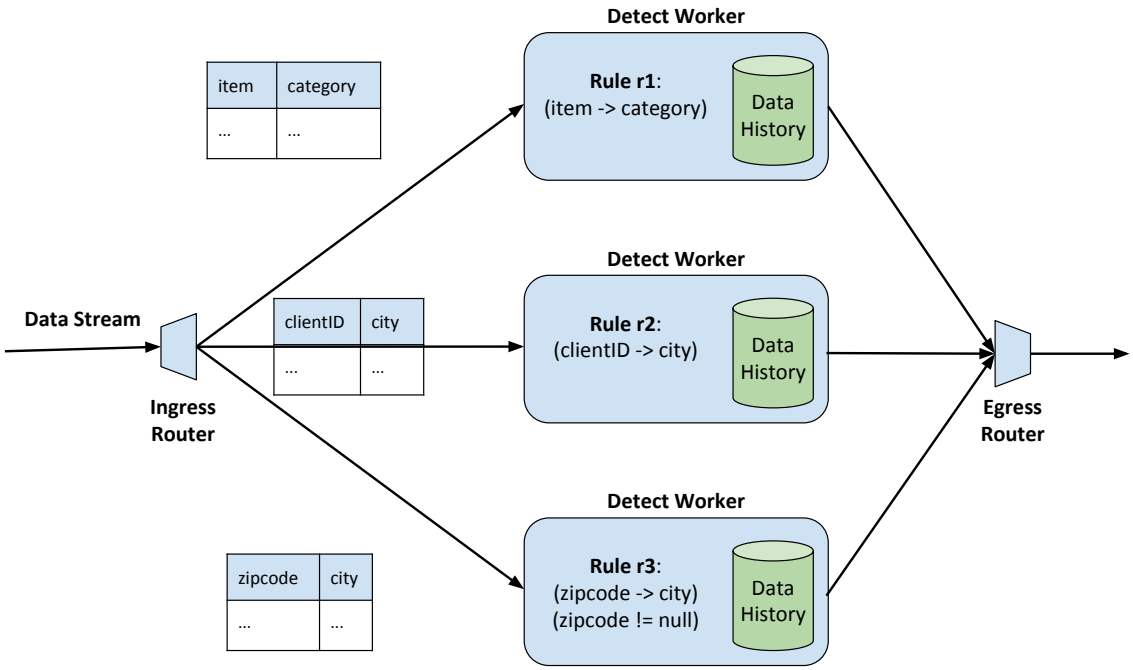


Figure A.7 – Le module de détection

pour le réparer, y compris: l'ID du groupe de cellules correspondant au tuple actuel et les cellules RHS des tuples actuels et antérieurs dans l'historique des données.

Maintenant, pour réduire le nombre de messages de violation, le DW peut utiliser une super cellule à la place d'une seule cellule en conflit avec le tuple actuel. En outre, rappelez qu'un seul CG peut contenir plusieurs super cellules, ce qui nécessite éventuellement plusieurs messages pour chaque groupe. Cependant, nous observons que deux cellules dans le même CG doivent également entrer en conflit les unes avec les autres, dans la mesure où leurs valeurs sont différentes. Étant donné que le module de réparation de données dans Bleach est à l'état, il est prudent d'omettre certains messages de violation.

A.3.4 Réparation de violation

L'objectif de ce module est de prendre les décisions de réparation pour les tuples de données sales, en fonction d'un flux intermédiaire de messages de violation générés par le module de détection. Pour ce faire, Bleach utilise une structure de données appelée *violation graph*. Les messages de violation contribuent à la création et à la dynamique du graphe de violation, qui regroupe essentiellement les cellules qui, ensemble, sont utilisées pour effectuer une réparation de données. Figure A.8 trace les éléments internes du module de réparation: il consiste en un routeur d'entrée, les réparateurs (RW) et un composant d'agrégation qui émet des données propres. Un composant supplémentaire,

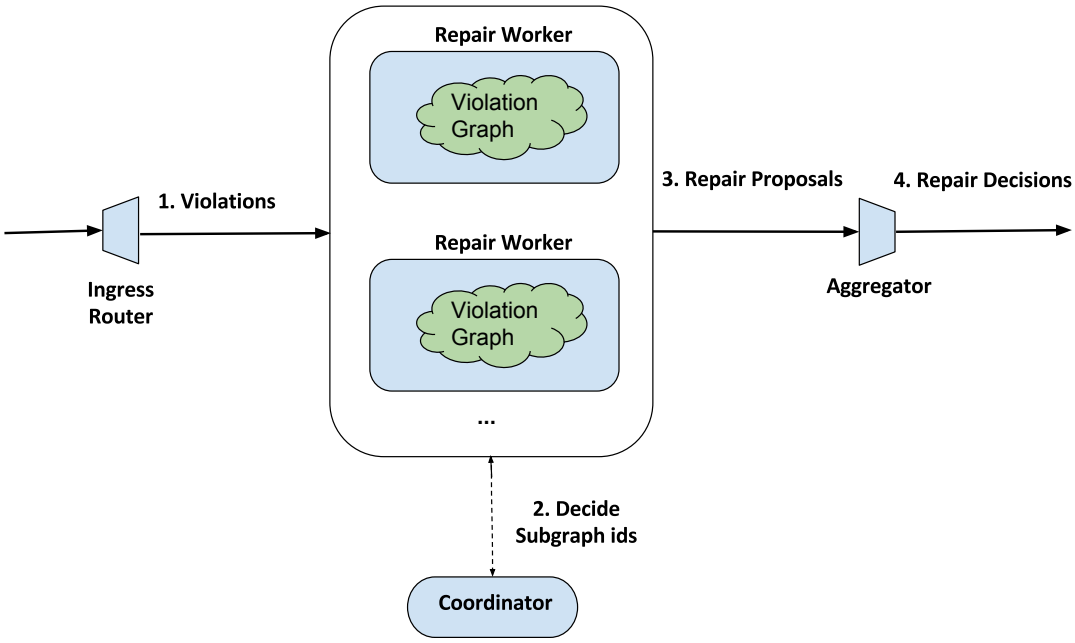


Figure A.8 – Le module de réparation

appelé le coordinateur, gère la gestion du graphe de violation, avec la contribution des RW.

Les algorithmes actuels de réparation de données utilisent un graphique de violation pour réparer les données sales en fonction des règles définies par l'utilisateur. Un graphique de violation est une représentation succincte des cellules (à la fois actuelle et historique) qui sont en conflit selon certaines règles. Un graphique de violation est composé de sous-graphes. En tant que flux de données entrants, le graphique de violation évolue: en particulier, ses sous-graphiques peuvent fusionner ou diviser, en fonction du contenu des messages de violation.

En utilisant le graphique de violation, plusieurs algorithmes peuvent effectuer un nettoyage de données, tel que l'algorithme de classe d'équivalence [125] ou l'algorithme holistique de nettoyage de données [102]. Actuellement, Bleach utilise une version *incrémental* de l'algorithme de classe d'équivalence, qui prend en charge les données d'entrée en continu, bien que des approches alternatives puissent être facilement connectées à notre système. L'idée de l'algorithme de classe d'équivalence est de regrouper tous les éléments qui doivent être équivalents les uns aux autres, puis de décider d'une valeur unique pour les éléments du même groupe. Ainsi, un sous-graphe dans le graphique de violation peut être interprété comme une classe d'équivalence, dans laquelle toutes les cellules sont censées avoir la même valeur.

Le composant agrégateur regroupe toutes les propositions de réparation de RW et sélectionne la valeur candidate pour réparer une cellule donnée comme celle ayant la plus

grande fréquence agrégée. Enfin, l'agrégateur modifie le tuple de données actuel et produit un flux de données propre.

A.3.5 Conclusion

Ce travail a introduit Bleach, un nouveau système de nettoyage de données de flux, qui vise à un nettoyage de données efficace et précis en contraintes en temps réel.

Tout d'abord, nous avons introduit les objectifs de conception et les défis liés au blanchiment, ce qui montre que le nettoyage des données du flux est loin d'être un problème trivial. Ensuite, nous avons illustré la conception du système Bleach, en nous concentrant à la fois sur la qualité des données (par exemple, les ensembles de règles dynamiques de Bleach et l'approche étatique des fenêtres) et sur les aspects systèmes (par exemple, partitionnement et coordination), requis par la nature distribuée de Bleach. Nous avons également fourni une série d'optimisations pour améliorer les performances du système, en utilisant des structures de données compactes et efficaces, et en réduisant les frais généraux de messagerie.

Enfin, nous avons évalué une implémentation prototype de Bleach: nos expériences ont montré que Bleach réalise une faible latence et une précision de nettoyage élevée, tout en absorbant un flux de données sale, malgré la dynamique des règles. Nous avons également comparé Bleach à un système de base fondé sur le paradigme micro-lot et expliqué Bleach performance supérieure.

Notre plan pour les travaux futurs est de soutenir un ensemble de règles plus varié et d'explorer d'autres algorithmes de réparation, qui pourraient nécessiter de revoir les structures internes de données que nous utilisons dans Bleach.

A.4 Conclusion

Dans cette thèse, nous avons étudié le problème de la façon d'accélérer le processus de préparation des données pour la grande analyse des données et fourni des techniques efficaces. En raison de la complexité de la préparation des données, nous avons ciblé deux étapes principales dans la préparation des données, le chargement des données et le nettoyage des données, et avons conçu et mis en place deux systèmes, DiNoDB et Bleach, qui peuvent aider les scientifiques à réduire considérablement leur temps consacré à la préparation des données. .

Tout d'abord, nous avons introduit DiNoDB, un système distribué pour les requêtes de vitesse interactive sur les fichiers de données générés par les cadres de traitement à

grande échelle. DiNoDB évite le chargement des données sans perte d'efficacité. Il intègre parfaitement les systèmes de traitement par lots avec un moteur de requêtes interactif distribué, tolérant aux pannes et évolutif. Il utilise un mécanisme de décoration qui améliore Hadoop I/O API standard et permet de créer des métadonnées auxiliaires requises pour la performance de requêtes à vitesse interactive. Notre vaste évaluation expérimentale a démontré que DiNoDB surpaste les autres solutions SQL-on-Hadoop pour une large gamme de charges de travail analytiques ad hoc. En outre, le mécanisme du décorateur peut également être exploité par d'autres systèmes en plus de DiNoDB, ce qui démontre qu'il s'agit d'une idée générale pour accélérer la préparation des données.

Deuxièmement, nous avons présenté Bleach, un nouveau système de nettoyage de données de flux. Contrairement à d'autres systèmes de nettoyage de données qui se concentrent principalement sur le nettoyage des données par lot, Bleach effectue le nettoyage des données directement sur les flux de données sans attendre que toutes les données soient acquises. Bleach vise à assurer un nettoyage qualitatif efficace et précis des données en cas de contraintes en temps réel. Il s'appuie sur des structures de données efficaces, compactes et distribuées pour maintenir l'état nécessaire pour réparer les données, en utilisant une version incrémentale de l'algorithme de classe d'équivalence. Notre évaluation a montré les performances supérieures de Bleach par rapport à un système de base construit sur le paradigme micro-lot, ce qui indique que le nettoyage des données en continu est efficace pour accélérer la préparation des données.

Bibliography

[1] M. Stonebraker and U. Cetintemel, “"one size fits all": an idea whose time has come and gone,” in *21st International Conference on Data Engineering (ICDE'05)*, April 2005, pp. 2–11.

[2] D. R. V. Turner, J. Gantz and S.Minton, “The digital universe of opportunities: Rich data and the increasing value of the internet of things,” 2014.

[3] Facts and Stats About The Big Data Industry, “Webpage,” http://cloudtweaks.com/2015/03/surprising-facts-and-stats-about-the-big-data-industry/.

[4] M. S. University and M. Stonebraker, “The case for shared nothing,” *Database Engineering*, vol. 9, pp. 4–9, 1986.

[5] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, 2009, pp. 165–178.

[6] I. F. Ilyas, X. Chu *et al.*, “Trends in cleaning relational data: Consistency and deduplication,” *Foundations and Trends in Databases*, vol. 5, no. 4, pp. 281–393, 2015.

[7] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, “Gamma - a high performance dataflow database machine,” in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 228–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671463

[8] Teradata, “Webpage,” https://www.teradata.com/.

[9] Greenplum, “Webpage,” https://www.greenplum.com/.

[10] Netezza, “Webpage,” https://www-01.ibm.com/software/data/netezza/.

[11] MySQL cluster, “Webpage,” https://www.mysql.fr/products/cluster/.

[12] Aster Data, “Webpage,” http://www.asterdata.com/product/ncluster_cloud.php.

[13] Postgres-XC, “Webpage,” https://wiki.postgresql.org/wiki/Postgres-XC.

[14] Stado, "Webpage," https://launchpad.net/stado.

[15] Amazon RedShift, "Webpage," https://aws.amazon.com/redshift/.

[16] ParAccel Analytic Platform, "Webpage," https://www.paraccel.com.

[17] Apache Tez, "Webpage," https://tez.apache.org/.

[18] Streaming Data, "Webpage," https://aws.amazon.com/streaming-data/.

[19] Apache S4, "Webpage," http://incubator.apache.org/s4/.

[20] Apache Samza, "Webpage," http://samza.apache.org/.

[21] Apache Flink, "Webpage," https://flink.apache.org/.

[22] Spark Streaming, "Webpage," http://spark.apache.org/streaming/.

[23] Spark Streaming, "Webpage," http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[24] Apache Storm, "Webpage," http://storm.apache.org/releases/2.0.0-SNAPSHOT/Guaranteeing-message-processing.html.

[25] R. MacNicol and B. French, "Sybase iq multiplex - designed for analytics," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 1227–1230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316798

[26] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1790–1801, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367518

[27] C. Baru and G. Fecteau, "An overview of db2 parallel edition," *SIGMOD Rec.*, vol. 24, no. 2, pp. 460–462, May 1995. [Online]. Available: http://doi.acm.org/10.1145/568271.223876

[28] M. Gorawski, A. Gorawska, and K. Pasterak, "A survey of data stream processing tools," *Information Sciences and Systems 2014*, p. 295, 2014.

[29] Deng *et al.*, "The data civilizer system," in *CIDR*, 2017.

[30] Improving Data Preparation for Business Analytics, "Webpage," https://tdwi.org/research/2016/07/best-practices-report-improving-data-preparation-for-business-analytics.

[31] N. Swartz, "Gartner warns firms of 'dirty data'," *Information Management Journal*, 2007.

[32] InsightSquared, "Webpage," http://www.insightsquared.com/2012/01/7-facts-about-data-quality-infographic/.

[33] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques (Data-Centric Systems and Applications)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[34] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[35] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1295–1306, Aug. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2732977.2733002

[36] M. Kornacker *et al.*, "Impala: A modern, open-source SQL engine for hadoop," in *CIDR*, 2015.

[37] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, February 2013.

[38] Y. Tian, I. Alagiannis, E. Liarou, A. Ailamaki, P. Michiardi, and M. Vukolić, "DiNoDB: Efficient large-scale raw data analytics," in *Data4U*, 2014.

[39] S. R. Labs, http://www.symantec.com/about/profile/researchlabs.jsp.

[40] A. Abouzeid *et al.*, "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads," in *VLDB*, 2009.

[41] I. Alagiannis *et al.*, "NoDB: efficient query execution on raw data files," in *SIGMOD*, 2012.

[42] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.crdrdb.org, 2011, pp. 223–234.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE MSST*, 2010.

[44] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2491245

[45] J. Dean *et al.*, "MapReduce: Simplified Data Processing on Large Clusters," in *USENIX OSDI*, 2004.

[46] J. Dittrich *et al.*, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," in *VLDB*, 2010.

[47] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *in Proc. of VLDB*, vol. 5, no. 11, pp. 1591–1602, Jul. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2350229.2350272

[48] M. Y. Eltabakh *et al.*, "CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop," in *VLDB*, 2011.

[49] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *CoRR*, vol. abs/1105.4252, 2011.

[50] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, 2011, pp. 1199–1208.

[51] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki, "Here are my data files. here are my queries. where are my results?" in *CIDR'11*, 2011, pp. 57–68.

[52] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" *CoRR*, vol. abs/1209.2191, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1209.html#abs-1209-2191

[53] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," in *VLDB*, 2013.

[54] K. Shvachko *et al.*, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10, 2010.

[55] M. Stonebraker and A. Weisberg, "The voltdb main memory dbms," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[56] R. S. Xin *et al.*, "Shark: SQL and Rich Analytics at Scale," in *ACM SIGMOD*, 2013.

[57] M. Zaharia *et al.*, "Spark: Cluster Computing with Working Sets," in *USENIX HotCloud*, 2010.

[58] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *in Proc. of ACM SIGMOD*, 2008.

[59] Apache Pig, "Webpage," http://pig.apache.org/.

[60] Sqoop, "Webpage," http://sqoop.apache.org/.

[61] Vertica, "Webpage," http://www.vertica.com/.

[62] Hadoop, "Webpage," http://hadoop.apache.org/.

[63] Apache Hive, "Webpage," http://hive.apache.org/.

[64] ArchivalStorage, "Webpage," https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html.

[65] "Postgresql." [Online]. Available: http://www.postgresql.org

[66] M. Kornacker *et al.*, "Impala: A modern, open-source sql engine for hadoop," in *In Proc. CIDR '15*, 2015.

[67] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe, "Cod: Database / operating system co-design," in *CIDR*, 2013.

[68] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *in Proc. of USENIX HotCloud*, 2010.

[69] Apache Spark, "Webpage," http://spark.apache.org/.

[70] Apache Storm, "Webpage," http://storm.incubator.apache.org/.

[71] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742797

[72] M. Zaharia *et al.*, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *ACM SOSP*, 2013.

[73] The Lambda Architecture, "Webpage," http://lambda-architecture.net/.

[74] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[75] M. Ester *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining*, 1996.

[76] Discardable Distributed Memory: Supporting Memory Storage in HDFS, "Webpage," http://hortonworks.com/blog/ddm/.

[77] A. Abouzied *et al.*, "Invisible loading: Access-driven data transfer from raw files into database systems," in *EDBT*, 2013.

[78] Y. W. Teh, D. Newman, and M. Welling, "A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation," in *Advances in neural information processing systems*, 2006.

[79] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, March 2003.

[80] Y. Chen, S. Alspaugh, and R. Katz, "Interactive query processing in big data systems: A cross-industry study of MapReduce workloads," in *Proc. of VLDB*, 2012.

[81] K. Ren *et al.*, "Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads," in *Proc. of VLDB*, 2013.

[82] H. Li *et al.*, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *ACM SOCC*, 2014.

[83] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," *DMTCS Proceedings*, vol. 0, no. 1, 2008.

[84] K. Krish, A. Anwar, and A. R. Butt, "hatS: A Heterogeneity-Aware Tiered Storage for Hadoop," in *Cluster, Cloud and Grid Computing (CCGrid)*, 2014, pp. 502–511.

[85] Ubuntu One, "Webpage," https://one.ubuntu.com/.

[86] Apache Parquet, "Webpage," http://parquet.incubator.apache.org/.

[87] Apache Hive - ORC Files, "Webpage," https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC.

[88] J. LeFevre *et al.*, "Miso: Souping up big data query processing with a multistore system," in *SIGMOD*, 2014.

[89] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[90] Y. Cheng and F. Rusu, "Parallel in-situ data processing with speculative loading," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14.

[91] Y. Cheng and R. Florin, "Scanraw: A database meta-operator for parallel in-situ processing and loading," in *ACM Transactions On Database Systems*, 2015.

[92] Y. Cheng, C. Qin, and F. Rusu, "Glade: Big data analytics made easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12.

[93] S. Melnik *et al.*, "Dremel: Interactive analysis of web-scale datasets," in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010.

[94] FITS, "Webpage," https://en.wikipedia.org/wiki/FITS.

[95] Reservoir sampling, "Webpage," https://en.wikipedia.org/wiki/Reservoir_sampling.

[96] W. Fan *et al.*, "Incremental detection of inconsistencies in distributed data," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1367–1383, 2014.

[97] W. Fan *et al.*, "Incremental detection of inconsistencies in distributed data," in *ICDE*, 2012, pp. 318–329.

[98] Gracia-Tinedo *et al.*, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in *IMC*, 2015, pp. 155–168.

[99] Z. Khayyat *et al.*, "Bigdansing: A system for big data cleansing," in *Proc. of SIGMOD*, 2015, pp. 1215–1230.

[100] Arocena *et al.*, "Messing up with bart: error generation for evaluating data-cleaning algorithms," in *Proc. of VLDB*, 2015, pp. 36–47.

[101] Dallachiesa *et al.*, "Nadeef: A commodity data cleaning system," in *Proc. of SIGMOD*, 2013, pp. 541–552.

[102] X. Chu *et al.*, "Holistic data cleaning: Putting violations into context," in *ICDE*, 2013, pp. 458–469.

[103] Jeffery *et al.*, "A pipelined framework for online cleaning of sensor data streams," Tech. Rep., 2005.

[104] Zhao *et al.*, "A model-based approach for rfid data stream cleansing," in *Proc. of CIKM*, 2012, pp. 862–871.

[105] S. Song *et al.*, "SCREEN: stream data cleaning under speed constraints," in *Proc. of SIGMOD*, 2015, pp. 827–841.

[106] Q. Lin *et al.*, "Scalable distributed stream join processing," in *Proc. of SIGMOD*, 2015, pp. 811–825.

[107] Elseidy *et al.*, "Scalable and adaptive online joins," in *Proc. of VLDB*, 2014, pp. 441–452.

[108] Wang *et al.*, "Crowd-based deduplication: An adaptive approach," in *Proc. of SIGMOD*, 2015, pp. 1263–1277.

[109] Chu *et al.*, "Katara: A data cleaning system powered by knowledge bases and crowdsourcing," in *Proc. of SIGMOD*, 2015, pp. 1247–1261.

[110] Beskales *et al.*, "Sampling the repairs of functional dependency violations under hard constraints," in *Proc. of VLDB*, no. 1-2, 2010, pp. 197–207.

[111] P. Bohannon *et al.*, "Conditional functional dependencies for data cleaning," in *ICDE*, April 2007, pp. 746–755.

[112] Interlandi *et al.*, "Proof positive and negative in data cleaning," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on.*

[113] Q. Chen *et al.*, "Repairing functional dependency violations in distributed data," in *DASFAA*, 2015, pp. 441–457.

[114] Kolahi *et al.*, "On approximating optimum repairs for functional dependency violations," in *Proc. of ICDT*, 2009, pp. 53–62.

[115] M. Volkovs *et al.*, "Continuous data cleaning," in *ICDE*, 2014, pp. 244–255.

[116] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," in *Proc. of VLDB*, 2015, pp. 1792–1803.

[117] Fernandez *et al.*, "Liquid: Unifying nearline and offline big data integration." in *CIDR*, 2015.

[118] Abedjan *et al.*, "Temporal rules discovery for web data cleaning," in *Proc. of VLDB*, 2015, pp. 336–347.

[119] Spark Summit 2015 Use Case, "Webpage," https://spark-summit.org/east-2015/streaming-machine-learning-in-spark/.

[120] Kafka, "Webpage," http://kafka.apache.org/.

[121] Xin *et al.*, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13, 2013.

[122] Recordedfuture, "Webpage," https://www.recordedfuture.com/.

[123] Gdeltproject, "Webpage," http://gdeltproject.org/.

[124] Spark stream cleaning, "Webpage," http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[125] Bohannon *et al.*, "A cost-based model and effective heuristic for repairing constraints by value modification," in *Proc. of SIGMOD*, 2005, pp. 143–154.

[126] Stonebraker *et al.*, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, pp. 42–47, 2005.

[127] Trifacta, "Webpage," https://www.trifacta.com/.

[128] Openrefine, "Webpage," http://openrefine.org/.

[129] Geerts *et al.*, "The llunatic data-cleaning framework," in *Proc. of VLDB*, 2013, pp. 625–636.

[130] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for rfid data streams," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, pp. 163–174. [Online]. Available: http://dl.acm.org/citation.cfm?id=1182635.1164143

[131] A. Zhang, S. Song, and J. Wang, "Sequential data cleaning: A statistical approach," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 909–924. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915233