UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUCCA SERGI BERQUÓ XAVIER

Evaluation and Performance of Reading from Big Data Formats

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Porto Alegre, Maio 2021

RESUMO

A emergência de novos perfis de aplicação ocasionou um aumento abrupto no volume de dados gerado na atualidade. A heterogeneidade de tipos de dados é uma nova tendência: encontram-se tipos não-estruturados, como vídeos e imagens, e semi-estruturados, tais quais arquivos JSON e XML. Consequentemente, novos desafios relacionados à extração de valores importantes de corpos de dados surgiram. Para este propósito, criou-se o ramo de *big data analytics*. Nele, a performance é um fator primordial pois garante análises rápidas e uma geração de valores eficiente. Neste contexto, arquivos são utilizados para persistir grandes quantidades de informações, que podem ser utilizadas posteriormente em consultas analíticas. Arquivos de texto têm a vantagem de proporcionar uma fácil interação com o usuário final, ao passo que arquivos binários propõem estruturas que melhoram o acesso aos dados. Dentre estes, o *Apache ORC* e o *Apache Parquet* são formatos que apresentam uma organização orientada a colunas e compressão de dados, o que permite aumentar o desempenho de acesso. O objetivo deste projeto é avaliar o uso desses arquivos na plataforma *SAP Vora*, um sistema de gestão de base de dados distribuído, com o intuito de otimizar a performance de consultas sobre arquivos CSV, de tipo texto, em cenários de *big data analytics*. Duas técnicas foram empregadas para este fim: *file pruning*, a qual permite que arquivos possuindo informações desnecessárias para consulta sejam ignorados, e *block pruning*, que permite eliminar blocos individuais do arquivo que não fornecerão dados relevantes para consultas. Os resultados indicam que essas modificações melhoram o desempenho de cargas de trabalho analíticas sobre o formato CSV na plataforma *Vora*, diminuindo a discrepância de performance entre consultas sobre esses arquivos e aquelas feitas sobre outros formatos especializados para cenários de big data, como o Apache Parquet e o Apache ORC. Este projeto foi desenvolvido durante um estágio realizado na *SAP* em Walldorf, na Alemanha.

Keywords: Sistemas distribuídos, Big Data Analytics, Formatos de arquivo.

ABSTRACT

The emergence of new application profiles has caused a steep surge in the volume of data generated nowadays. Data heterogeneity is a modern trend, as unstructured types of data, such as videos and images, and semi-structured types, such as JSON and XML files, are becoming increasingly widespread. Consequently, new challenges related to analyzing and extracting important insights from huge bodies of information arise. The field of *big data analytics* has been developed to address these issues. Performance plays a key role in analytical scenarios, as it empowers applications to generate value in a more efficient and less time-consuming way. In this context, files are used to persist large quantities of information, which can be accessed later by analytic queries. Text files have the advantage of providing an easier interaction with the end user, whereas binary files propose structures that enhance data access. Among them, *Apache ORC* and *Apache Parquet* are formats that present characteristics such as column-oriented organization and data compression, which are used to achieve a better performance in queries. The objective of this project is to assess the usage of such files by *SAP Vora*, a distributed database management system, in order to draw out processing techniques used in *big data analytics* scenarios, and apply them to improve the performance of queries executed upon CSV files in *Vora*. Two techniques were employed to achieve such goal: *file pruning*, which allows Vora's relational engine to ignore files possessing irrelevant information for the query, and *block pruning*, which disregards individual file blocks that do not possess data targeted by the query when processing files. Results demonstrate that these modifications enhance the efficiency of analytical workloads executed upon CSV files in *Vora*, thus narrowing the performance gap of queries executed upon this format and those targeting files tailored for big data scenarios, such as Apache Parquet and Apache ORC. The project was developed during an internship at SAP, in Walldorf, Germany.

Keywords: Distributed Systems, Big Data Analytics, File Formats.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CSV	Comma Separated Values
JSON	JavaScript Object Notation
XML	Extensible Markup Language
RCFile	Row Columnar File
SaaS	Software-as-a-Service
PaaS	Platform-as-a-Service
IaaS	Infrastructure-as-a-Service
EC2	Elastic Compute Cloud
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
SQL	Syntax Query Language
MIT	Massachusetts Institute of Technology
DSS	Decision Support Systems
EOF	End-of-File
LF	Line Feed
ORC	Optimized Row Columnar
HDFS	Hadoop Distributed File System
AWS	Amazon Web Services
S3	Simple Storage Service
IoT	Internet-of-Things
MQTT	Message Queuing Telemetry Transport
RLE	Run-length encoding
SoF	SQL on Files
SIMD	Single Instruction Multiple Data
AFSI	Abstract File System Interface

CONTENTS

1 INTRODUCTION

1.1 Context

In the era of Big Data, new challenges arise from the need of analyzing and extracting important insights from large datasets. There has been a shift in the profile of modern applications, which now increasingly deal with sheer amounts of information, generated in an accelerated rate (KAMBATLA et al., 2014). The advent of video streaming, social media and wearable devices, for instance, have facilitated such phenomenon. With this new paradigm, the need of handling data efficiently - and more importantly, draw value from it - has become a recurrent task. Furthermore, the proliferation of unstructured (e.g., videos, images, PDF) and semi-structured (e.g., XML, JSON) data took place, as opposed to the traditional relational models, which were the norm in previous decades.

These three dimensions - velocity, volume and variety (LEE, 2017) - have called for a new class of highly-scalable and distributed systems, which need to cope with sizable quantities of data. In this scenario, resources need to be made scalable and broadly available for usage. *Cloud computing* has emerged as a solution which abstracts and provides capabilities on-demand over the Internet: some examples are *Software-as-a-service* (or *SaaS*) which makes applications available over the cloud (e.g., Google Drive, Dropbox) and *Platform-as-a-service* (or *PaaS*), in which computing resources such as virtual machines and storage are provisioned via a network connection (e.g., Amazon EC2, Google App Engine).

In this context, data has been made accessible via distributed storage systems that can accommodate large volumes of information. These repositories, also known as *data lakes*, represent a novel method of integrating data, in which many kinds of information are stored: text documents, images, audio are a few examples. To the detriment of classical approaches that force every record to be placed in a common schema, this solution relaxes standardization and delegates modeling to client components that consume its content, resulting in extended operational insight. Many companies started then to uncover the value hidden in data. Amassing and storing it has little value: in fact, this information has to be cleansed, processed and analyzed until the true value behind it comes to light. This realization paved the way for *big data analytics*, which groups a set of technologies and techniques for extracting valuable insights from huge datasets. *Descriptive analytics* is employed to reveal what has occurred, and it is placed at the core of big data analytics. OLAP and data visualization are tools used to that end. On the other hand, *predictive analytics* serves to anticipate and explain what may occur based on existing records. Techniques and methods of machine learning and document mining are actively applied in that sense.

Therefore, many modern applications are heavily data-oriented. In the business land-

scape, however, this profile is even more apparent: according to a study conducted by the MIT Sloan School of Management (LAVALLE et al., 2011), senior executives state that their organizations have more data than they can use effectively. Moreover, they strive for data-driven business decisions, and want analytics to exploit their growing collection of information and computational power to improve efficiency and innovate their companies. Hence, more than ever, it is important to measure and optimize how computing solutions designed for analytical processing behave, in regards to how data is accessed and stored.

1.2 Motivation & Objectives

The project was developed during an internship at SAP in Walldorf, Germany. It's centered around *Vora* (SAP Vora, 2021), an in-memory, distributed database management system built for ingesting sizable volumes of information and dealing with big data scenarios. This platform is a key component in the company's landscape, as it provides a solution for scalable storage, and SAP's main products use it. A SQL-based language for executing queries called *VORA SQL* is provided.

Vora has the ability of consuming data from external databases and, more importantly for the project, from different data lakes. In such systems, information is often stored in raw binary data or in files. The former approach is mostly used when unstructured data is involved, such as audio, image and video contents. The latter option is employed when a certain data organization is present, but it is not enforced by a higher level schema. For instance, in a power plant, time-based records may be generated by sensors and grouped in log files, which later can be persisted in data lakes.

Generally, files can be classified into two types - binary and text. The former category presents sequences of bytes arranged in a custom way which gives meaning to the information conveyed in the file. They often contain headers, which are special sections of bytes that identify the file's contents and represent metadata, such as the file extension and other descriptive information. Text files, in their turn, possess textual data, meaning that the bytes stored by them represent a certain text character. Different encodings can be used in that sense, such as UTF-8 or ASCII. These files contain control characters, which have no visual representation and are used to exert a certain effect in the file. Some examples are the *End-of-file* (*EOF*) character and *line feed* (*LF*).

Vora has a built-in functionality called *SQL on Files*, allowing queries to be performed upon semi-structured data stored in other databases or data lakes. Hence, it is leveraged in analytical scenarios, in which queries are used to profile information contained in files. In that context, different file types are supported. Among them, *Apache ORC* and *Apache Parquet* are binary formats which present characteristics such as high compression, columnar organization and embedded statistics to increase performance when handling data. In fact, they are widely adopted in many organizations to optimize read and write times (ORC Adopters, 2019) (Parquet Adopters, 2019). Due to this fact, they are often referred to as "*big data file formats*". Another widely used format is *CSV* (which stands for *Comma Separated Values*), a widespread text file format due to its readability and simplicity. It can easily be used to represent spreadsheets, for instance.

Hence, motivated by an increasing interest in big data analytics as a way to generate business value, performance is a crucial requirement as it enables a time effective and efficient processing of analytical workloads. When considering queries made upon files, format particularities play a central role in the efficiency. While Parquet and ORC are tailored for fast data processing, text formats incur a costly overhead of parsing textual

information and converting it to data types supported by the target system (PALKAR et al., 2018). Another challenge is the lack context when dealing with raw data chunks (GE et al., 2019): text formats do not possess any header or metadata section that indicate the size of a column, how many records it possesses, start and end offsets for lines or what data type is encompassed. All of these issues need to be addressed when handling text files. Yet, there is an increasing interest in extracting insights from raw data, such as text files (IDREOS et al., 2011), as many applications generate textual information. Among them, CSV is widely used due to its simple format (TAPSAI, 2018).

Thus, the first objective of this work is to analyze how *big data file formats* are handled in *Vora*. From this analysis, two techniques are derived: *predicate pushdown*, which relies on the query predicate to skip the parsing of unnecessary file blocks, and *file pruning*, which eliminates the processing of entire files that are not relevant to the query. Given that parsing CSV files suffers from a hefty processing - which involves steps such as finding line positions and transforming raw data into textual information - these techniques are extended to this format. With the help of predicate pushdown, a new technique called *block pruning* is proposed, which is used to skip the download and parsing of CSV file blocks. Moreover, file pruning is used to omit irrelevant CSV files that do not possess relevant data for the query. The implementation of both techniques relies on calculating minimum and maximum statistical values for CSV columns. A final evaluation of processing times and performance of analytical queries with CSV will be done to assess the proposed solutions.

1.3 Organization

This work is organized as follows: Chapter 2 introduces concepts relevant to the project's development: technical aspects about Vora, the field of big data analytics, data lakes, file formats and related work. Chapter 3 elaborates the work objectives, while Chapter 4 explains how the solutions proposed in this work were devised. Chapter 5 demonstrates what improvements in the CSV processing were implemented in Vora.

In Chapter 6, the results obtained after the solution implementation are analyzed and performance evaluations are made. Finally, Chapter 7 presents conclusions and future work.

2 RELATED CONCEPTS

This chapter introduces concepts relevant to the current work. Details about the Vora platform which will aid in comprehending the proposed solutions will be discussed. Moreover, the subject of big data analytics, file formats and data lakes will be characterized, an their relation to the project will be presented.

2.1 SAP Vora

As introduced previously, this component is a distributed database management system designed for a massive scale-out, which supports a SQL-like language for interaction, called *SAP Vora SQL*. It serves as a base element offering scalable storage, upon which systems can build their functionalities. Vora targets clusters possessing a number of commodity hardware, which can range from a single-digit up to hundreds of machines. Aiming to serve the needs of Big Data, such as large data volumes, fast processing and data variety, Vora has adopted some important design principles:

- Support of multiple data models and engines, configuring a *poly-store* model, designed to unify querying over multiple data schemes.
- Embracing other open-source Big Data systems, such as Hadoop, HDFS and Spark, allowing the system to deal with existing solutions.
- Use the *Kubernetes* platform to run on all major cloud vendors (eg., AWS, Azure).
- Shared-nothing architecture, meaning that the system is ready for deployment in nodes that are independent and self-sufficient. This mitigates the influence of *single points of failure* and alleviates bottlenecks for scaling.

This platform uses a multi-component approach instead of presenting a monolithic architecture. For instance, the database system is composed of different services interacting with each other, which are designed for accomplishing different tasks. To name a few, Vora has separate processes for metadata handling, persistence and processing. To the outside, it behaves like one logical system. This organization is achieved by containerizing the different components present in the platform. Vora clusters use *Kubernetes* for container orchestration, and its services are containerized using *Docker*.

Multiple processing engines are embedded in this product. They are responsible for taking care of incoming requests, performing the actual processing work on the data, and finally forwarding it to be stored in a persistent way. The **in-memory relational engine** is responsible for loading relational data into main memory to achieve a fast access and

efficient query processing. Also, there are dedicated engines for graph and time-series processing, as well as a disk-based engine.

The system can load data from third-party data lakes, such as Google Cloud Storage, HDFS and Amazon S3, which can be later used in queries and analyzed via Vora's engines. It also integrates with Spark, which allows the reuse of data processing solutions written for this framework.

2.1.1 Architecture

The strongest architectural trait of Vora is the ability to support online transaction processing (OLTP) while being able to scale out to handle online analytical processing (OLAP) for large data flows. In fact, these two capabilities in a single data management system that can be scaled to multiple nodes to better manage memory utilization was the primary motivation for the product's creation. Figure 2.1 illustrates the overall architecture of Vora.

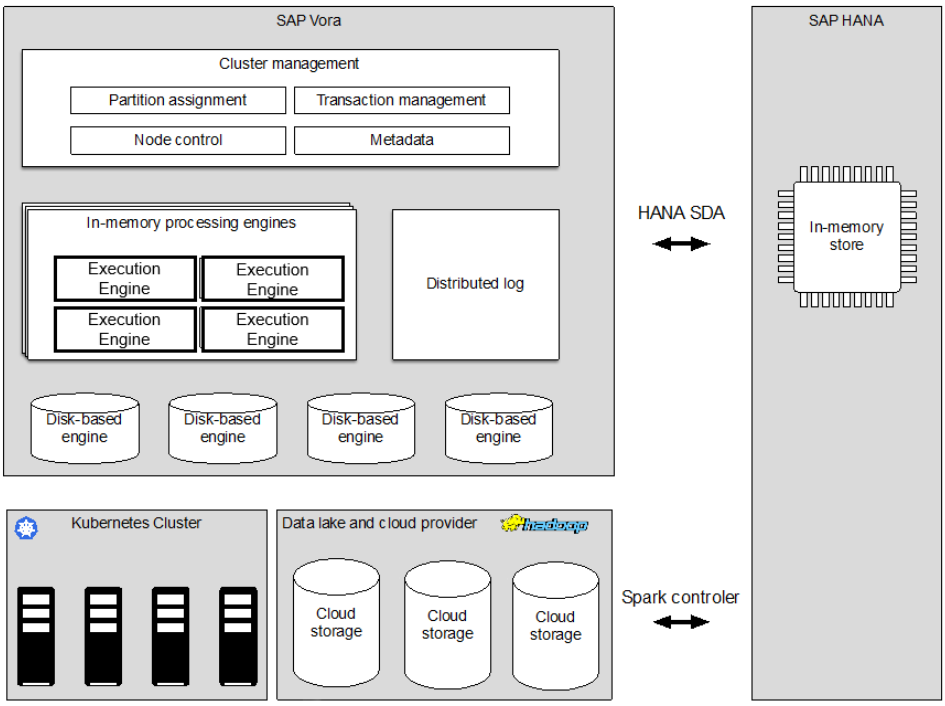


Figure 2.1: Overview of SAP Vora architecture. *Source: SAP Vora documentation*

Three major components interact to manage the data and conduct query execution. The **query engine** (referred to as *engine nodes*) is a core constituent which relies on SQL-to-C code generation to seek for high-performance. Its main task is to apply the demanded operations on the appropriate data, and it resides in the compute nodes. The transaction coordination is carried out by a **transaction broker**. Its purpose is to hold a shared state, which is used to process transactions. Finally, the **distributed log** tends to the durability of transactions. It is shared by the different architectural entities of Vora, and it is built out of servers called *storage units* - usually aggregated in a individual cluster. In this organization, a transaction executor modifies this log by writing results in it, and a commit occurs when the correspondent commit log entry is made persistent.

To favor query processing, tables are horizontally divided in *slices*, which correspond to a disjoint set of rows. This organization is important since these slices are large enough

to exploit compressed data processing, but small enough to permit an effective distribution and swift replication among nodes. They are directly placed in engine nodes, which may hold data for many snapshots simultaneously. In that context, a design choice was made so that these nodes are not aware and hence do not participate directly in the database transactions. Theoretically, they operate on read-only database *snapshots*. The architectural construction of the Vora cluster proposes two distinct subsets of engine nodes: one for handling OLAP workloads, and another for OLTP ones. This allows different trade-offs to be made for each type of node, as OLTP deals with frequently accessed (or *"hot"*) data, while OLAP wants to process its entirety.

2.1.1.1 Transaction coordination

The transaction broker is the component responsible for the transaction management in the system. It keeps a table of running transactions and their state, issuing read timestamps to incoming ones, as well as a version table which is used for storing write-sets and detecting possible write conflicts. It is important to note that the broker does not maintain a persistent state - all information needed for describing it is stored in the log. Consequently, any node can be elected as the new broker after a failure.

In Vora, read-only transactions receive a read timestamp from the transaction broker, and then are marked as *running* in the transaction table. Then, they are forwarded to the query engine cluster responsible for the database. If the data slices are not up-to-date with the desired version, the log is consulted for updates, and the data is refreshed. It may be that the compute nodes has the required updates in its cache: in that case, there is no need to consult the log.

Once the data slices receive the desired version, the query can be executed against the correct data, and the result is passed to the application. This scenario configures a *pull-based*, or lazy execution. Instead, the engines may subscribe to changes at the log, so modifications can be forwarded and applied to the local snapshot without any outside request from a query. In that way, a *push-based* strategy, or eager execution, is adopted. Vora can shift between these modes of execution, according to the workload nature.

In the context of read-write transactions, the same execution path is followed: the transaction broker assigns a read timestamp to it, and the queries are also passed to the engine node cluster. However, the execution in these nodes does not modify any database entity. Alternatively, the updates are executed in a read-only mode (changes are cached locally), and the engine returns a set of row IDs modified by the executed operations.

When the transaction completes, the broker checks for conflicts in the write set. If no issue is found, a commit timestamp is created, changing the transaction state to *pre-commit*. The transaction writes a commit record in the log, at the position given by the previous timestamp. Upon success, the broker publishes this commit timestamp, and the transaction is marked as *committed*.

2.1.1.2 Data replication and consistency

The distributed shared log is the sole mechanism used by transactions to commit their changes in the data. This component is essential as it represents the truthful copy of the database. It ensures data replication, as any node can consult the log to build updated snapshots for query processing. Also, it is used in disaster recovery situations, in order to reestablish the system state. Due to its importance, the log is implemented as a cluster of storage units, possessing features such as partitioning and replication.

The log is composed of entries, which hold all the updates and modifications per-

formed by a specific transaction. These records are indexed by commit timestamps, also known as log sequence numbers (LSN). Therefore, each transaction is mapped to a unique log entry. In this way, the broker can use them to reference a specific slice in the compute nodes, thus creating consistent copies of data across the system.

A compute node can read from the log in order to build new versions of the hosted snapshot slices. It should be pointed out that there is no need for different snapshot replicas to be synchronized to the same LSN, since any replica can be updated by simply reading the log. In this configuration, the system assumes an eventual consistency protocol.

The interface exposed by the distributed log provides a total ordering over all writes (by the log sequence number). To that end, write-once semantics and chain replication (VAN RENESSE; SCHNEIDER, 2004) are employed.

2.1.1.3 Distributed Query Processing

The system possesses a cluster of dedicated nodes that adhere to a robust distributed query processing service. Its main purpose is to manage the mapping of table slices to the compute nodes. In addition, it also produces a distributed execution plan so that incoming queries can be executed via the query engines located in the nodes. Although the **in-memory relational engine** is the most important and optimized one, Vora also supports a disk-based engine.

The query engine is equipped with an execution stack, composed of units such as a parser, a semantic analyzer and an optimizer. This mechanism is used to transform the query plan into C code, followed by a translation into an executable binary format. The main benefit gained from this approach is the reduction of data transfers between CPU and the main memory, given that many operations (such as filter, project, join) can be applied on cached data objects.

2.2 Big data analytics

Many authors use three dimensions to characterize the field of Big Data: *velocity*, due to the accelerated rhythm of data generation, *variety*, as different types of data in various degrees of structuring are observed, and *volume*, given that the amount of information available is immense (LANEY, 2001). Beside these three, another equally relevant dimension is *value*.

With the recent advance of Web technologies and the dissemination of portable devices, data has become widely available. However, static information has little to no interest in real-world applications. Frequently, datasets need to be cleaned, analyzed and even compared against others so that new insights can emerge - consequently, data is regarded as a precious asset nowadays. For that reason, increased attention is being given to *big data analytics*, which encompasses tools and techniques dedicated to examine datasets and extract value from large bodies of data. The field of *analytics*, nonetheless, is by no means a recent development. Its very meaning is broad and susceptible to many interpretations (WATSON, 2013). In the 1970's, the *Decision Support System* (DSS) concept was consolidated as a category of interactive information systems that used data and models to help managers analyze semi-structured problems. In the following decades, many decision support tools and techniques became popular, which led to the establishment of *business intelligence*. The latter term groups systems, methodologies, and applications that analyze critical data to help an enterprise make timely business decisions (CHEN;

CHIANG; STOREY, 2012). *Online Analytical Processing* (OLAP) is an example of approach that is used to process multi-dimensional data in that context. As these data analysis tools and techniques developed and scaled out, the term *analytics* became to be used interchangeably. More recently, *big data analytics* has been used to describe applications in which datasets are so large and diverse (data types are complex and disparate) that special storage, management and analysis are required. Advances in CPU capabilities, storage, virtualization and distributed computing have enabled this kind of analytical workload.

There are three main types of analytical processing:

- ***Descriptive analytics*** is used when users need to understand *what has occurred*; in this situation, new findings of past activities often emerge from the collected data. For instance, a company might analyze yearly sales reports from many branches and find out the most profitable location, and what products were the most demanded.

- ***Predictive analytics*** is employed for finding trends that the stored data may express; in that way, users may have clues of *what will occur*. Financial applications are notably prominent in that area, in which past financial markers are analyzed to extract a possible forthcoming market tendency. Methods and algorithms of machine learning, for instance, are widely used for that purpose.

- A rising method of analytics, which derives from the *predictive* variation, is ***behavioral analytics*** . It involves processing large amounts of behavioral data (i.e., associated with actions of people), collected from sources such as wearable devices and ambient sensors, in order to identify patterns and foresee customer actions.

These forms of analysis are often used together to extract new information from datasets, and have been sought out by many businesses. In regards to their implementation, many strategies can be adopted to enhance analytical processing. **In-memory analytics** leverages the faster access times offered by random access memory (RAM) to attain fast query processing. By storing data on RAM instead of the physical disk, access time is improved by orders of magnitude (Garber, 2012). This is useful in OLAP scenarios, for instance, in which many dimensions need to be handled. However, as it relies on costly technologies, such as DRAM, it is still considered to be relatively expensive price-wise, so it is not well suited for all use cases. **In-database analytics** offers a way to mitigate data movement through a network, as analytical tools are embedded in the database software. This avoids migrating datasets to a central server, making the totality of data, not just a representative subset, available for analysis. It can be implemented through libraries or user created functions that offer analytic or data mining capabilities.

The placement of rows can be another point for performance improvement. **Columnar databases** hold their records in a column-oriented organization, instead of the traditional row-oriented format observed in many relational databases. In most cases, this variation improves the overall processing time of analytical workloads, because they typically access a large number of rows, but a reduced amount of columns. Thus, by holding data in a column-oriented fashion, columns can be accessed individually, as their values are stored separately, and hence information that is irrelevant to the query can be skipped. For instance, in a row-oriented database, the processing would possibly be less efficient, since all rows would be scanned, and many column values would be skipped, as they would not be needed in the analytical query's context. This results in a better utilization of data by columnar storages, as there is little need to read and discard columns that are

not needed, resulting in a better utilization of available I/O and CPU-memory bandwidth for analytical queries.

Another advantage of columnar databases is that they can leverage a greater compression ratio when using compressing techniques. Since column values pertain to the same domain, better ratios can be achieved, as data is naturally related. Also, since columns may often present repeated or frequent values, encodings can benefit from the repetition. In big data analytics scenarios, even though improving storage usage is important, speeding up query processing times takes precedence. Hence, encodings that yield a non-optimal compression ratio, but offer fast decompression are preferred. Some examples are *Dictionary Encoding*, *Run Length Encoding* (RLE), delta encoding and bit-vector encoding. Moreover, as columns are stored independently, each one of them can be compressed with a compression scheme most suitable for their data type.

2.2.1 Areas of application

The extensive generation of data in many application domains has caused the adoption of large-scale analytical solutions. Some areas that are increasingly adopting big data analytics are:

- **Natural and geographical applications**: a variety of data related to humankind's environmental footprint is being created. This information is gathered by sensing devices, satellites and monitoring equipment (e.g., drones, marine probes), to name a few. Researchers and governments use this kind of material to track events such as deforestation, ice-cap coverage, and weather phenomena, which may have a direct economic impact. For instance, the Sentinel project, coordinated by the European Space Agency, provides a data repository of satellite imagery which are used in real-world applications, such as crop monitoring (Farm Sustainability Tool (FaST), 2020) and to measure droughts (Drought monitoring in Romania, 2020).

- **Business and commercial systems**: one of the most prominent usages of analytics is business and commerce applications. Inherently, enterprises that use such appliances gather a vast amount of customer data - location, preferences, age, etc. - which can create new economic opportunities. Predictive analytical methods are crucial in that area. Nowadays, a typical use case is recommendation systems that apply machine learning models trained upon large quantities of data to suggest new products to customers. Descriptive methods can also be applied, which help businesses to better understand their operation, summarize different aspects and provide new market insights or points of improvement.

- **Social networks and Internet**: as more people make use of social networking platforms, more effort is spent in analyzing interactions and user activity in order to understand the emergence of behavioral patterns, shape information flow, predict future trends and manage resources (PENG; WANG; XIE, 2016). Big data analytics provides important tools for analyzing unstructured data - graph mining techniques, image and video indexing are some examples.

By no means these applications are exhaustive. The adoption of analytical technologies grows by the year, and are used in many other fields.

2.2.2 Big data analytics in Vora

Many companies face the challenge of having to combine their in-house data (stored in databases or data warehouses, for instance) with other "big data sources" (e.g., stored in data lakes, in Hadoop clusters, etc.). Hence, Vora is designed to address this issue. By providing such integration, combined with the in-memory capabilities of the platform, companies can execute their analytical workloads on a heterogeneous data landscape. It also allows Spark to be used for data analysis.

Therefore, one use case in which Vora is used is the ingestion of IoT data. Figure 2.2 illustrates such scenario. Sensor information generated from IoT devices are distributed by a message broker, which writes information to a set of topics (e.g., location, type of measurement, etc.). In this case, MQTT, a network protocol based on broker-message communication, is used. Then, Kafka is used to process such streams of information, which can then be consumed by Spark or by a component called *vFlow* - a proprietary software which allows the construction of data pipelines. Once in this system, Vora can consume this data via its Spark integration, for instance, and combine it with data coming from external relational databases - in the figure, HANA, a SAP database is used. Furthermore, additional data coming from data lakes (e.g., Amazon S3) can be integrated. The query engine can be used to construct queries upon the combined dataset, and information can be stored back in Vora or in the data lake of choice.

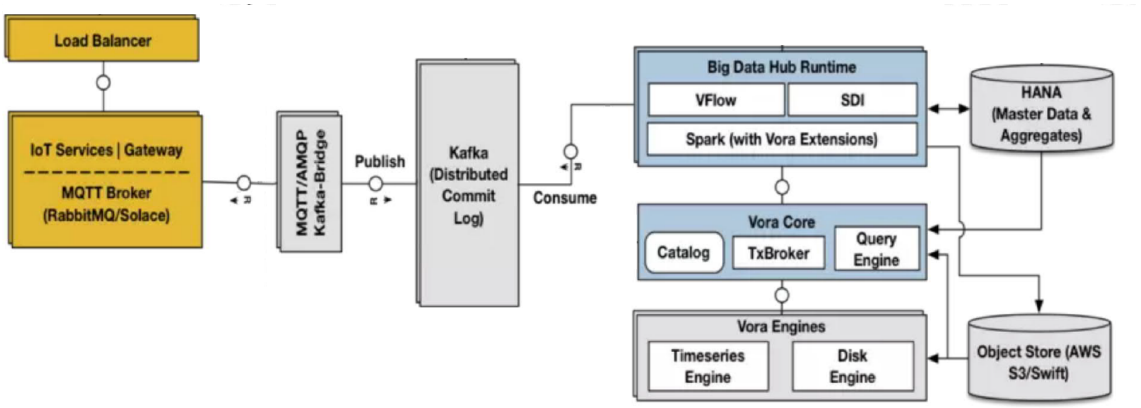


Figure 2.2: Overview of a scenario in which Vora is used to ingest IoT data.

Another analytical capability of Vora is **SQL on Files**, which resides in the core subject of the presented work. With this functionality, queries can be executed upon files coming from external data lakes, and the results may be used in conjunction with other sources of data (for instance, in-house data or relational data coming from other database, for which Vora provides a connectivity).

2.2.3 State of the art

This section present some alternatives to Vora that also provide similar analytical and storage capabilities in big data scenarios.

In the big data analytics sphere, one of the most fundamental and evolving paradigm is **MapReduce**, used for processing and handling large data sets. This programming model is centered around two basic routines: *map*, that processes a key/value pair to generate a set of intermediate key/value pairs, and *reduce*, which combines all intermediate values associated with the same key. It has been essential in the big data field as it allowed

to scale computations to many machines in a distributed manner, and thus permitting applications to deal with a larger amount of information in a scalable fashion.

Although also integrated with Vora, **Apache Hadoop** is a framework that gained notoriety for leveraging the *MapReduce* model. It manages the data spread across various machines in a cluster through its distributed file system, HDFS (*Hadoop Distributed File System*). Typically, the Hadoop infrastructure runs MapReduce programs (written in a certain programming language, such as Scala or Java).

Apache Pig is platform that offers a high-level procedural programming language, called *Pig Latin*, which allows users to write MapReduce programs that run on *Hadoop*. It has a SQL-like notation, which makes building parallel programs that run in a cluster easier. Another tool that is part of the *Hadoop* ecosystem is **Apache Hive**. Similar to *Pig*, it offers a SQL-like language for building data reports; however, it abstracts the complexity of building MapReduce programs, offering a declarative language for analyzing data in a Hadoop cluster. It can also be considered a *data warehousing* solution.

Another tool for performing analytics on large data sets is **Apache Spark**. It can be defined as an analytical engine for data processing (Apache Spark Official Website, 2020), offering libraries to deal with structured and semi-structured data, streaming analytics, machine learning and graph processing.

Among proprietary software solutions, **Amazon Athena** (Amazon Athena, 2021) is a query service that allows analyzing data in Amazon S3 using standard SQL language. It includes support for many file formats, including Parquet, ORC and CSV. Another similar solution is **Amazon Redshift Spectrum** (Amazon Redshift Spectrum, 2021), retrieve structured and semi-structured data from files in Amazon S3. It allows concurrent queries to the same dataset without having to copy the layer in the cluster nodes. An example of tool which can also query files in data lakes is **Dremio** (Dremio Query Engine, 2021), an analytical engine designed for fast data access envisioning use cases such as data science and business applications.

Regarding columnar databases, **HBase** is a solution that features in-memory computation, column compression and a NoSQL operation model, in which data is presented as key-value pairs. It is built on top of Hadoop, and its tables may be used as input for MapReduce jobs. HBase is often used in scenarios in which a small portion of items needs to be located among a huge quantity of data. Another similar alternatives are **Amazon Dynamo** and **Cassandra**. These two provide a highly available, distributed, fault-tolerant column store. Lastly, **MongoDB** is a document-oriented database representing information in a JSON-like structure.

2.3 Data lakes

As a consequence of an increased generation of unstructured data, a new category of systems to store and organize this information has been created - they are called *data lakes*. They refer to scalable storage repositories that contain raw data, or in other words, information that does not abide by a higher level fixed schema. Instead, it is stored in its native format, and may be consumed by other applications for tasks such as visualization, and analytics, for instance. These systems can be found as *on-premise* solutions, meaning that data centers within an organization are used to implement them, or as *cloud* products. In the latter category, *Google Cloud Storage* and *Amazon S3* are examples of proprietary data lakes available as Infrastructure-as-a-service (*IaaS*) offerings. A Hadoop cluster can also be used to implement a data lake, as HDFS provides a scale-out architecture that can

accommodate large influxes of data.

Information is often found in the form of *files* in data lakes. This is because files are used in many cases to record values generated by a given application, and persisted in these repositories to be used in other scenarios. Notably, files are used to convey important information such as logs and sensor data, customer records and website click-streams. Thus, there is an increasing interest in using files stored in data lakes in analytical workloads. As these repositories can accommodate files coming from different sources, organizations may use them for analysis and reporting, without incurring the cost of transforming these data collections - that may differ in format and nature - in a common schema (Fang, 2015).

2.4 File formats and organizations

The format of the file chosen to be used in a given application may impact performance, as many formats and organizations can be employed.

One of the most straightforward ways to convey information is by representing it via **text files**. Being a simple and human-readable type, it can be used to display information for the end-user easily. However, its organization poses some drawbacks when it comes to the processing of the data. These files include control characters to separate values, which increase their total size and require extra computation when parsing. Another issue is the multiple encodings that a text file can take, which increases the complexity to handle them. Some examples of these files are **CSV** and **JSON**.

Most Big Data tools support the usage of text file formats. They can be ingested from a certain source, and stored in persistent storage (such as in a data lake). Nevertheless, they are far from being considered as the most appropriate file format in this domain. Text ingestion is slower to process if compared to other formats, mainly because encoding and text formats are not bound to each other, and thus are not easily detected at runtime. For instance, a CSV file may come with an *ASCII*, *UTF-8* or any other kind of encoding.

Consequently, queries made upon a text file become less efficient. A naive query engine would have to scan the whole file to get the result, as the values selected could be anywhere in the file. This fact, added up with the delimiter character handling, makes this kind of file inefficient in terms of query execution, if not handled properly. Thus, it is essential to apply processing techniques to enhance the performance of these types in Big Data applications.

When dealing with large-scale data, very often intermediate tables are generated. These contain transitional information representing a step in the chain of data treatment. Hence, they are not ready to be used by the end-system; yet, they need to be stored, so further computation can be executed upon them. As pointed out previously, text files are not an optimal choice of file format to represent such tables. The memory and calculation overhead that they bear would be a considerable performance penalty for the targeted systems,and especially when answers are demanded within a few minutes and not potentially hours.

Consider a table represented in a file which contains numerous rows with user interactions and only some few of columns. One common case in analytical applications is a query or a report interested in fetching and reading a small amount of the data (e.g., values of a single column). If performance is a concern, it would not be feasible to scan the whole table to retrieve only a reduced portion. The optimal solution would be concentrating only on the searched slice of data, and ignoring the unwanted ones. Therefore,

row and column skipping represent a great economy in both time and performance.

Following the requirements for efficient data handling, file formats can present modified structures to enhance the scan of data or the write performance. **Row group** is a concept used in many file formats, representing a horizontal partition of data that encompasses several records. Within this group, the data is arranged in a column-oriented fashion. Figure 2.3 illustrates such a structure. On the right-hand side, the row group contains records for columns *A*, *B*, *C* and *D*, and the information is laid out in a column-major order.

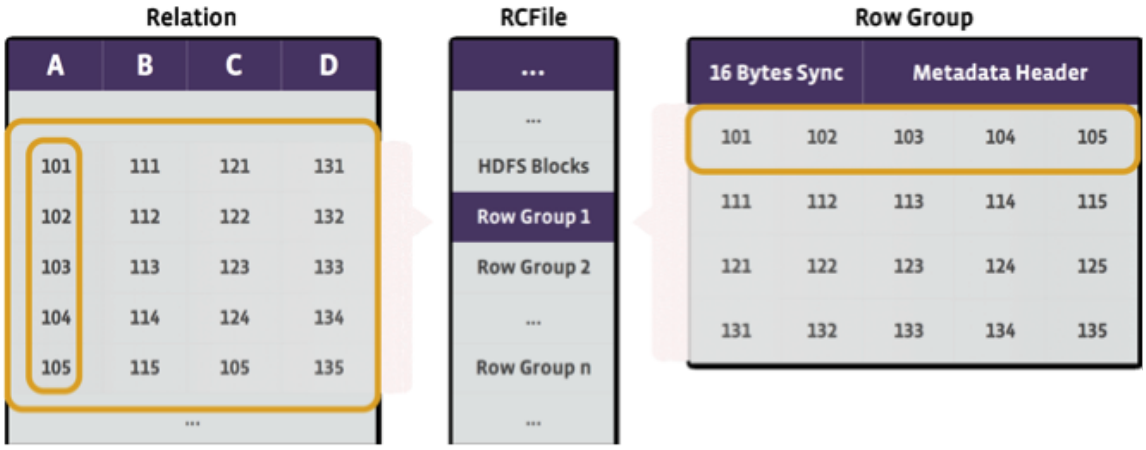


Figure 2.3: The RCFile illustrates the usage of row groups. Note that, on the right, the row group displays the information in a column-oriented format *Source: Facebook Code - Facebook Engineering Blog. https://code.fb.com/core-data/scaling-the-facebook-data-warehouse-to-300-pb/*

One example of format adopting this strategy is the *RCFile*, which stands for *row columnar file*. Hadoop, for instance, makes use of this kind of file. In this case, one or several row groups are stored in a HDFS file. A key point of using such an organization is the parallelism used, since the row groups of different files are disseminated redundantly across the cluster and treated simultaneously. Subsequently, each node reads only the columns important to the query from a file - and the appropriate data portion - skipping irrelevant ones. Additionally, more space can be saved, as the applied compression takes advantage of the similarity in a column.

Another construct embedded in some file formats to enhance data retrieval is **column statistics**. Usually, these measurements are calculated not only for the whole file, but also for certain blocks (e.g., for each group of thousand rows). Different statistic markers can be used: minimum, maximum, median and mode are commonly employed examples. This allows skipping an irrelevant set of rows that may not be used for a certain file query, and may reduce the number of data read to get the result.

2.4.1 ORC Files

The **ORC** (*Optimized Row Columnar*) (Apache ORC Specification, 2021) file format was proposed to enhance further the time, cost and performance gain in the Big Data storage contexts. It benefits from a columnar-based organization and makes use of row groups and file statistics.

The ORC files retain information about the column types, as specified in the table definition. Consequently, different compression techniques (e.g., dictionary encoding, bit packing, RLE) can be applied according to the type of a column, as to obtain smaller files.

In addition, there are specific readers and writers for each column type.

ORC can apply generic compression, using zlib or Snappy, on top of the aforementioned compression techniques for further reducing the size of the files. So, storage savings are considerable with this format. However, compression is not done over the whole file, but only in some parts, as to allow individual de-compression of file chunks. There are indexes that include the minimum and maximum values of each column, for each set of 10,000 rows (located at a stripe's footer) and for the entire file (locate at the file footer), so the reader can skip sets of rows not concerned in a query. Furthermore, ORC supports projection, which applies the query in a subset of the columns, so predicates that include only a few of them will result in reading only the required bytes. Figure 2.4 shows the general structure of a ORC file.

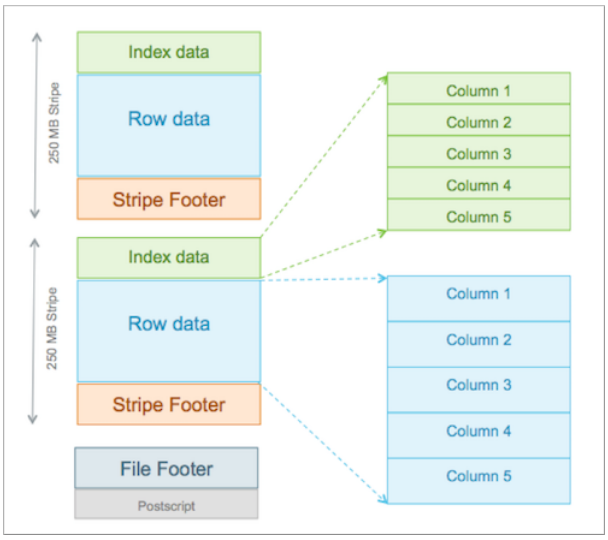


Figure 2.4: Structure of a ORC File. *Source: Hortonworks Docs*

The ORC file format is composed basically of three parts:

- The header: small portion of the file, composed of three bytes indicating the magic number "ORC"
- The body of the file, which is divided into **stripes**. Each stripe is self contained, meaning that it can be read separately from the rest of the file, and contain only entire rows. Stripes are subdivided into three sections: a set of indexes for the rows within the stripe, the data itself and a stripe footer. Both the indexes and the data sections are divided by columns so that only the data for the required columns needs to be read.
- The file tail, which possesses metadata about the file and other details. It contains the **file footer**, which encapsulates the layout of the file's body, the type schema information, the number of rows, and the statistics about each of the columns. Also, the **metadata** portion contains column statistics (minimum and maximum values) at a stripe level granularity so that stripes can be skipped. Finally, the *postscript* has information to interpret the rest of the file, including the length the last two sections of the tail.

Major companies in the tech industry validate the benefits of the ORC format: Facebook has published an article (Facebook Inc, 2019) that demonstrates the advantages of

this file configuration when applied to a large data warehouse. It resulted in a boost in compression ratios on the warehouse data, and a write performance increase.

2.4.2 Parquet Files

The Parquet format, similarly to ORC, aims to reduce the amount of I/O used to read data from a file. It stores binary data in a column-oriented way, benefiting from the advantages of compressed columnar data representation. Also, it is built to support very efficient compression and encoding schemes. The former methods are specified on a per-column level and are future-proofed to allow adding more encodings. Its main inspiration is Google's Dremel paper (ZAHARIA et al., 2012), which describes the company's query system for analysis of read-only nested data.

The specification defines a partitioning in row groups, whose size varies from 128MB to 1GB. The values for these sizes are not arbitrary - they were chosen to allow row groups to fit in one or more HDFS blocks, which are 128MB in size. Internally, a row group is organized in "column chunks", which are batches of values belonging to a certain column. Encoding methods can be applied to them. These chunks are further sub-divided into pages of 1MB, to which compression algorithms (e.g., snappy, zlib and LZO) can be applied.

2.4.3 Performance in analytic scenarios

The aforementioned file structures and formats can yield an increase in performance when dealing with analytical workloads. However, choosing the appropriate format is a case-by-case task, as a certain variation may be optimal for one scenario, but sub-optimal in others. An experiment presented in the Hadoop Summit San Jose 2016 compared the performance of certain *Hive* queries executed upon different file formats: JSON (text format), ORC, Parquet and Avro (binary formats). The datasets chosen were:

- **NY Taxi dataset**, containing information about taxi rides in New York from 2009. Comprised of 18 columns, having double, integers, decimals and strings as types. The data collection is available at (TLC Trip Record Data, 2020).

- **Sales dataset**, constructed from a real Hive deployment. It has 55 columns with a considerable amount of null values, and the information is randomized. The contained data types are strings, longs, timestamps and booleans.

The type of query considered in the experiment was reading all columns in the file. It is a simple test, but provides a rough comparison of the reading performance in each format. The test measurements were done using *JMH* (Java Microbenchmark Harness). By default, JMH executes 10 forks, in order to get isolated execution environments, 20 warmup cycles with no measurements, as to allow the JVM to optimize code before the benchmark starts, and finally 20 real measurement iterations.

Figure 2.5 shows the result for full table scan scenario in the taxi ride dataset. The JSON format presented the worst performance, as reading the records involves parsing strings, finding delimiters and dealing with special characters. On the other hand, reading the binary text formats was more efficient, mostly due to the fact that they present structures which reduce the total size of the dataset, and they do not have to deal with the overhead of converting text.

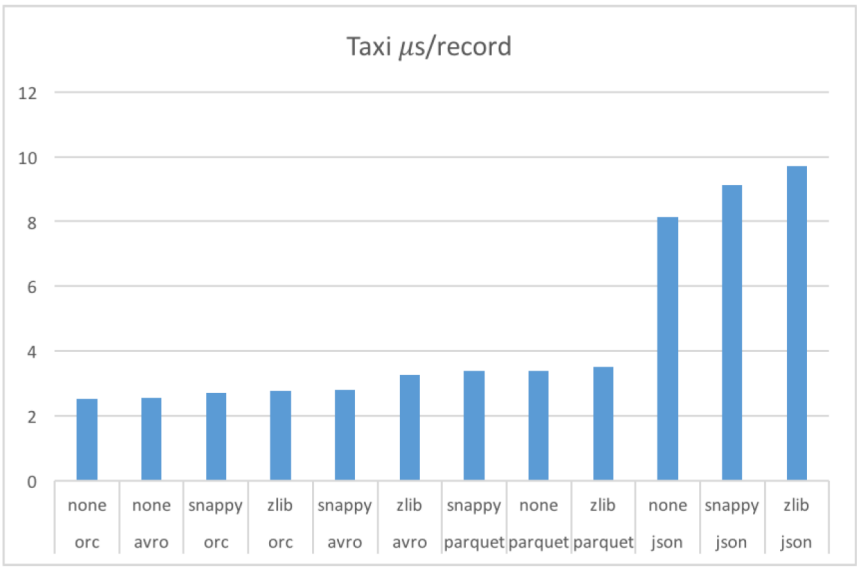


Figure 2.5: Comparison of the amount of time needed to read each record when performing a full table scan query in the NYC Taxi dataset. The type of compression is also a considered variable. *Source: Owen O'Malley - Hadoop Summit June 2016. https://www.slideshare.net/oom65/file-format-benchmarks-avro-json-orc-parquet*

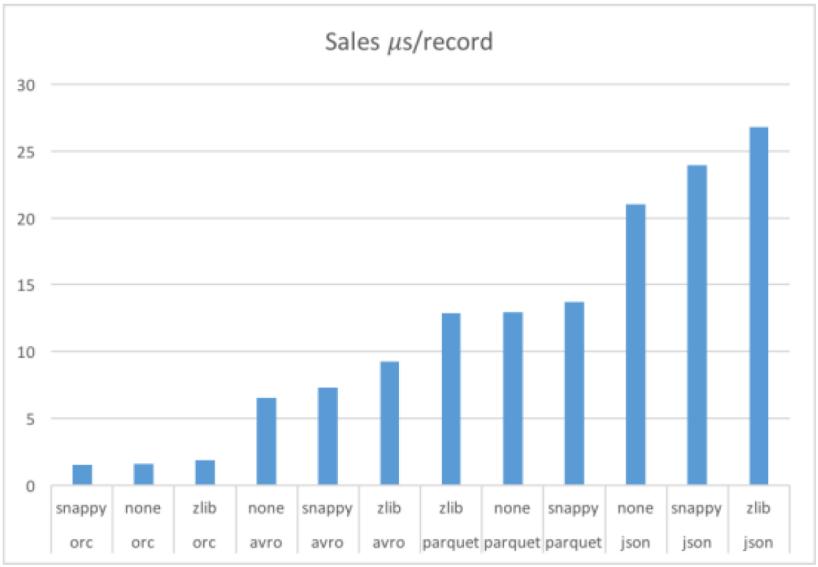


Figure 2.6: Comparison of the amount of time needed to read each record when performing a full table scan query in the sales dataset. The type of compression is also a considered variable. *Source: Owen O'Malley - Hadoop Summit June 2016. https://www.slideshare.net/oom65/file-format-benchmarks-avro-json-orc-parquet*

The results on the other dataset for the same type of query is shown in Figure 2.6. Once again, JSON presents the worst performance among the file types. However, we see that ORC presents a better performance in this dataset, whereas Parquet presents a significantly larger record processing time. Another key factor is the choice of the compression used upon the files, as it presents a space-time trade-off: a certain method might turn the overall size of the dataset smaller, but it will incur a time overhead when decompressing the information found in the file.

Although simple, this experiment highlights that the choice of a file format can impact largely the overall performance of queries, specially considering that datasets may contain thousands or millions of rows.

2.5 Related Work

Characteristics of related works on file processing in the context of big data analytics are presented and compared to the current work in this section.

2.5.1 Analysis

As the generation of unstructured data formats increases, many works of research focus on applying efficient data processing techniques on such types of information. In (ALAGIANNIS et al., 2012), authors discuss a new paradigm for achieving efficient query processing upon raw files, called *NoDB*. Through an adaptive indexing mechanism implemented over *PostgreSQL*, their work strives for efficient file data access by storing positional information using a caching mechanism. Performance bottlenecks in file processing are also discussed, particularly tokenizing, parsing and data conversion costs. These challenges are also observed in *Vora* and are discussed in section 4.3.

The work of (JAIN; DOAN; GRAVANO, 2008) studies SQL queries over text databases, focusing on information extraction techniques over text documents to obtain structural relations that may hide in the textual information. These traits are then taken into account by techniques that combine query predicate to enhance query execution times. A family of select-project-join SQL queries is employed to validate their proposal. Data quality is also considered when analyzing experiment results. In this research, authors recognize the considerable cost incurred to extract information from text documents, and aim to minimize irrelevant accesses to bodies of information. The proposed solution discussed in chapter 5 presents a way of mitigating such issue.

In (KARPATHIOTAKIS; ALAGIANNIS; AILAMAKI, 2016), query engine optimizations and design techniques for dealing with heterogeneous datasets are proposed. In particular, the authors present the implementation of an engine that executes queries over binary data, CSV and JSON, offering support to joint datasets (e.g., join on CSV and JSON files). By using code generation, adaptations are made to the engine's architecture based on the query predicate and targeted format. The work also mentions analytical queries over CSV data, affirming that these workloads usually require fast response times. Authors propose structural indexes that store the binary positions of data columns in each row for CSV files. Results showed that their implementation are suitable for analytical scenarios, proving efficient in synthetic and real-world workloads. However, unlike Vora, the engine supports does not support the ingestion from external data sources.

CSV processing is also at the core of (GE et al., 2019). This work highlights the inherent costs of ingesting CSV data in the context of analytical applications. Namely, the parsing of raw data is identified as one of the major difficulties of systems that support analytics on text files. One problem cited by the authors is the lack of context when processing file chunks: for instance, finding the beginnings and ends of rows and records. In the paper, authors propose a speculation-based distributed parsing approach for the CSV format. Their solution divide text files by chunks, which are distributed among worker nodes that execute the parsing step in parallel. The proposed implementation is validated in Apache Spark using real-world datasets, and yields a 2.4X speedup over existing methods when parsing CSV files. Nevertheless, this work does not explore the

usage of such files in analytical queries.

Authors of (MITLöHNER et al., 2016) examine characteristics of open data CSV collections. More than 230 portals were analyzed, and a total of 413 GB of information was scanned. The study concentrates on the structure of encompassed data, such as column types and value distributions. Main findings are that an average CSV file has 365 rows and 14 columns. Moreover, approximately 50% of columns in the study dataset consisted of either numerical values or text IDs, and approximately 14% of the data columns were sorted. In Vora, analytical scenarios rarely deal with open-source or publicly available data. Instead, the observed file datasets are often structured (e.g., a file directory whose sub-folder are partitioned by a certain column) and are property of a company (i.e., private datasets). Nevertheless, the work provides an interesting insight on how CSV is used to store data.

Finally, (IVANOV; PERGOLESI, 2019) discuss performance on columnar files, such as ORC and Parquet. Authors execute a series of benchmarks for these formats on Hive and SparkSQL. The obtained results confirm that file format selection and their configuration considerably affect the global performance of workloads: while ORC presented better measurements in Hive, Parquet was the appropriate pick for SparkSQL. Compression levels were also factors that impacted performance on both file formats.

2.5.2 Discussion

The aforementioned works share some characteristics with the implementation of techniques to enhance CSV processing in Vora. Data parsing and skipping irrelevant data accesses are recognized challenges and considered during the implementation of the proposed solutions in the current work. None of the works explore the execution of analytical queries upon files in data lakes nor what practices can be implemented to enhance performance when dealing with text formats in that scenario. The present work will discuss these themes. In addition, another factor that distinguishes Vora from the presented systems is its capability for processing data coming from heterogeneous sources, such as data lakes, third-party databases, and even data processing frameworks, such as Apache Spark.

3 OBJECTIVES

The objective of the current work is to enhance the performance of queries made upon CSV files in Vora. Modifications in the *SQL on Files* functionality are proposed: this feature allows querying data contained in files of various formats. For that end, the algorithm used for processing "big data formats" (ORC and Parquet) in analytical workloads is analyzed to identify what techniques are used to enhance query execution time. They are then extended to CSV to address the main difficulties of processing this format in Vora, namely file block fetching and raw data parsing. Finally, performance tests are made to assess the impact of the proposed changes on queries executed upon CSV files in Vora.

3.1 SQL on Files

The SQL on Files feature in Vora (hereby denoted as *SoF*) allows queries upon files stored in data lakes. These repositories hold *cold data* - that is, huge bodies of data that remain unclaimed, without being accessed for long periods of time. By nature, this type of information does not require its entirety to be accessed frequently. Hence, loading all this data and converting it into database tables would be a waste of resources.

SoF allows loading parts of file collections during query execution to optimize the system's performance - thus, named *SQL on files*. Users can use it to aggregate data coming from data lakes and use it alongside other sources, such as relational databases. If needed, tables can be created in Vora to store the query results. By providing a way to unify the data, these tables can then be consumed by other solutions to build further data enhancements. For instance, Vora has connectivity with a data pipelining solution provided by SAP, which allows users to cleanse the data, apply machine learning methods, among other use cases. Thus, SoF can be considered as a data aggregation feature in Vora.

The main use case targeted by this functionality are:

- **Selective queries** involving wide tables, targeting a specific set of columns and a certain value range. In these scenarios, users want to get some specific information from the dataset (e.g., select the days in which the total income was greater than 100).
- **Data profiling**, in which a table is temporarily created to reference a specific file or, in most cases, a folder of files. Descriptive queries are executed without many filters or joins. For instance, find the minimum or maximum value in a given file column, gather its sum, or count the occurrences of a certain value.

Typical scenarios include files whose content are static - meaning that their data is hardly (if ever) modified. Queries are emitted in parallel and possess high selectivity - in

other words, they concentrate on a small subset of the data, without joins.

In contrast, there are scenarios that SoF does not target. Queries involving many joins and lacking selective filters are one of them. In this context, a user has many tables whose content is hardly modified, and complex join-intensive queries are fired over them, often requiring complete scans of all data due to the lack of highly selective filters. Likewise, many tables with foreign key relations and files with data being modified regularly are not ideal execution scenarios.

3.2 CSV file performance

Regarding the file formats, the most used binary types are ORC and Parquet; CSV is the only text format supported by the functionality. Other kinds of less relevant files that are not included in this project's scope are *Avro* and *RCFiles*.

Even though text files do not represent an optimal choice in analytical workloads, many companies that make use of Vora still use them, as they represent a simple and widespread option for storing data, and can be used easily by the end-user (e.g., spreadsheets represented as a CSV file). However, there is an enormous performance gap when comparing CSV with other "big data formats" in analytical scenarios, as the former types of file possess specific structures that improve processing time. Therefore, the changes proposed in the current work will contribute to lower the discrepancy between execution times of CSV queries and those executed upon ORC or Parquet, for instance.

4 MODELLING

This chapter presents how the solutions proposed to enhance the execution time of queries made upon CSV files in Vora are modeled. Details about how the relational engine in Vora processes files are studied. Moreover, algorithms for reading big data formats and CSV are examined. The characteristics drawn from such analysis motivated the implementation of the proposed solutions.

4.1 Processing files in Vora

The SoF feature is embedded in Vora's relational in-memory engine via direct interaction with Vora's file loader, also known as the **file importer** or simply **importer**. All processing of files is done in the main memory, and no data is swapped to the disk. Hence, all extensions made to the relational engine are available for *SQL on files*, and vice-versa.

Given that SoF relies on the relational engine, which executes computations exclusively in the main memory, the execution is bound by the available memory. However, filters (such as `WHERE` clauses) and aggregations (`GROUP BY`) are implemented to be computed in a pipelined fashion. Thus, the memory consumption can be reduced if queries have a highly selective filter or if the applied aggregations only generate so many groups.

When executing queries with the SoF functionality, the totality of data is not loaded into memory (nor into the disk). Since, in many cases, the total size of the targeted file dataset is huge, it does no t make sense to waste resources by storing all the information in Vora. Instead, the execution is carried out by loading files by blocks. For Parquet and ORC, they have a default size of 128MB and 250MB, respectively, whereas, for CSV, it is 100MB. This number presents a reasonable trade-off between network bandwidth consumption and processing time, and it was a design choice made by the importer. As the data pointed by an SoF table is not loaded into the database, there is no entity mapping the file data slices to nodes. In other words, no partitioning scheme is attached to the SoF table.

The pseudo-code exposed in 4.1 presents how a file stored in a data lake can be referenced in Vora.

```
CREATE TABLE customer (
c_custkey INTEGER NOT NULL,
c_name VARCHAR(25) NOT NULL
) WITH TYPE DATASOURCE ENGINE 'FILES';

ALTER TABLE customer ADD DATASOURCE ds1 PARQUET(S3('customer.
    parquet' ));
```

```
USING CONNECTION s3_conn ON ERROR ABORT;

LOAD TABLE customer;
```

Listing 4.1: Example of a Vora table definition targeting a Parquet file stored in Amazon S3

The `CREATE TABLE` statement describes the format of the data coming from the source file. Next, the `ALTER TABLE` command adds a data source to the *customer* table, by referencing the file "*customer.parquet*" stored in Amazon S3. Optionally, instead of a file path, users may specify a folder, which will prompt the engine to execute queries against all files in that directory.

Finally, the `LOAD TABLE` statement is required to load information about the file into the *Catalog* - the metadata store for the relational engine. This command will not load the file data into memory nor persistent storage. In contrast, it will check the given file path and store additional information about the source. After this point, subsequent selects may be used upon the declared data.

Striving to better serve numerous formats, the importer relies on an abstraction layer called **Abstract File System Interface** (AFSI). It groups a set of basic operations for file I/O offered by data lake providers, and supports sources such as:

- Hadoop Distributed File System (HDFS) and WebHDFS
- Amazon S3
- Google Cloud Storage (GCS)
- Azure Data Lake (ADL)

The most important functionalities provided are opening a connection, file operations (such as *seek*, *read*, *write*), file properties (e.g., *size*, *chmod*) and folder operations (e.g., *mkdir*, *ls*). Given that many data sources are supported, the AFSI component loads a dynamic library for correctly managing the type of connection upon opening a file. For instance, if the user specifies an HDFS URL for a file, it will load the appropriate `.so`, `.dll` or `.dylib` library responsible for dealing with the Hadoop file system.

4.1.1 SoF: processing big data formats

As presented in previous sections, SoF avoids loading the whole file dataset into main memory, as most often queries are interested in a reduced portion of the information. File formats such as **Parquet** and **ORC** present an excellent alternative in that context since they possess structures that allow loading certain portions of a file. Moreover, the embedded file statistics allow the query engine to use the column predicates and skip loading irrelevant file blocks. This also avoids fetching whole files and not using them, especially when queries present restrictive predicates that target value ranges that do not pertain to the files involved in the query - selecting a column value that does not exist in a file is a notorious example.

Another use case that benefits from the clever organization of such files is when queries are made upon file directories. In many cases, the sub-folders have a certain hierarchy, as data is partitioned by column value ranges. For instance, imagine a directory containing files that store temperature data coming from IoT sensors. Sub-folders can

be organized by date: the first folder hold files whose values were measured in January, the next one stores measurements from February, and so forth. In that case, upon firing a query that targets the temperatures collected in May, for example, it would be reasonable to assume that only the folder storing values from May is opened and retrieved from the data lake. If somehow the query engine associates statistical values (e.g., minimum and maximum values from the "*Date*" file column) to the folders, such execution can take place.

Hence, Vora leverages the statistical information contained in such files and maintains a **file statistic cache** to allow query optimization. Whenever a file is read, the statistics associated with particular blocks are cached. Then, the importer can use those markers to avoid loading file portions that contain irrelevant data in the query context, so file processing can be optimized. In that sense, query predicates are moved "closer" to the data, as the importer now will consider them when loading files. This technique is called **predicate pushdown**. Query performance can be enhanced by using this method, as data is handled more efficiently: only relevant file chunks are processed by the importer.

The same rationale can be applied when dealing with file directories. As stated previously, queries that are executed against a directory tree should only touch the appropriate folders. Loading any file that does not contain pertinent information to the query would be a performance penalty, as it would not impact the final result. Therefore, just as done with files, folder statistics are also cached in Vora. This extends the benefits of predicate pushdown to queries executed upon folders. However, the performance gain is considerably larger: instead of skipping file chunks, whole sub-directories that do not yield actual results to the query can be ignored. Hence, this technique is referred to as **file pruning**.

Figure 4.1 represents a fictional dataset represented in a file directory. Individual files hold a certain value range, and are organized in sub-folders. In this example, there is no overlapping of values between files. File pruning can be applied in this example when a query is made upon the directory "*/file_directory*": if it selects data greater than 900 and smaller than 3000, then only "*file_20.orc*", in blue, should be read. Predicate pushdown is used when individual files are targeted - if "*file_01.orc*" is queried for data between 15 and 30, only the green block should be fetched and processed.

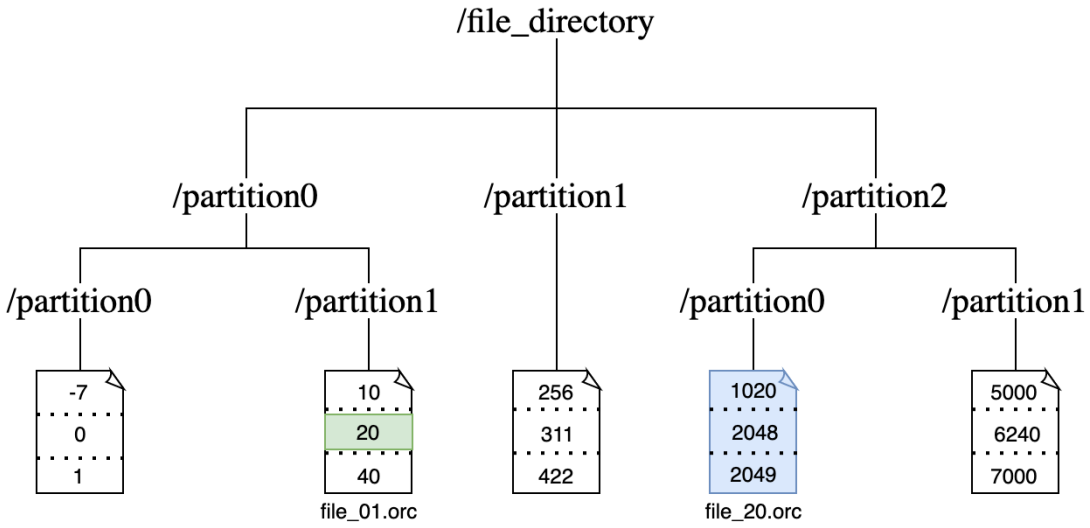


Figure 4.1: Example of a file directory that partitions data in sub-folders. Files have only one column, and are placed on folders which represent a certain value range. Each file is subdivided in blocks, which hold only one column value

4.1.2 File query execution

In order to analyze how Vora deals with executing queries upon big data file formats, and how the aforementioned techniques - predicate pushdown and file pruning - are employed, the query execution path in Vora will be presented.

The engine's critical path for executing SoF queries is the following: a sequence of commands is processed by the **parser**, which is a component that checks the syntax of a SQL query and transforms it into an Abstract Syntax Tree (AST). Then, the **semantic analyzer** checks the semantics of the SQL query and annotates the received AST with type information. The **catalog**, which contains metadata about the tables, is consulted for obtaining information about the data source and column types. Next, a logical execution plan is generated by the **plan generator**, and then forwarded to the **optimizer**.

The latter is a key component to the importer because time-saving modifications are made to the original logical plan. Most importantly, the pushing down of query filters - in other words, **predicate pushdown** - are made here, using the cached file statistics maintained by the engine. Pruning of irrelevant table slices and exclusion of hosts containing them are taken into account on the host assignment, if the tables are maintained in memory, in order to apply **file pruning**.

As output, the optimizer creates a physical plan, based on the aforementioned enhancements. Then, the *Landscape Manager*, which holds information about the worker nodes and data slices they contain, is used to decide the optimal plan distribution as plan fragments are sent to the assigned nodes.

One important remark is that for SoF queries, since the file data is not persisted in the database, the nodes do not hold the data slices pertaining to the files. A query plan is sent to nodes, containing a data source description along with the statistics (if applicable). The importer component of the node's relational engine opens a connection with external source (i.e, data lakes) and retrieves the file data.

4.1.3 File statistics cache

The file statistic cache is implemented in Vora through a distributed dictionary, called RDS (*Replicated Data Structure*) Map. It provides a consistently replicated map of string keys to object payloads, and it is backed up by Vora's distributed log. This ensures that all operations are durable before they are applied to the in-memory structure. Moreover, it guarantees that if an API call modifying the data structure (e.g., compare-and-swap on an RDS Map) returns successfully, then the modification is guaranteed not to be lost, even under client or server failure.

The cache identifies specific files by an entity tag (ETag), which most data lake providers support. It is included in HTTP response headers when the engine accesses files in a given data lake, and it serves to identify a specific version of a file resource. If ever the file's content is changed, the respective ETag is also modified. Thus, it can be consistently used in cache validation.

The user can configure the percentage of the node memory that is allocated to the cache. A special table called `INTERNAL_FILESCAN_CACHE_DETAILS` holds details about the cache, such as filenames, hashes, access counts, among others.

Each cache entry consists of an object which contains overall statistics about the file, such as files accesses, number of accessed blocks, number of rows read, among others. It also holds column statistics, which are divided by file blocks. They are used by the optimizer when building the execution plan, and are passed onto the importer to implement

predicate pushdown and file pruning. Hence, every time that the importer accesses a file specified by the query plan, the ETag contained in the response is compared to the one associated with the received statistical information, to verify whether it can be correctly used during the query execution. The cache uses a LRU policy to manage its entries.

The cache stores statistical data of ORC and Parquet files, since they already contain this kind of information on their body (conveniently for the cache, they are also divided by blocks). Prior to the project, the CSV format was not supported by the cache, as no statistical information was available for this type of file. Hence, CSV could not leverage the optimizations available for the previous formats.

4.2 Analysis of ORC and Parquet processing algorithm

By analyzing how the importer processes the two main "big data formats", techniques that are used to enhance file processing become evident. This analysis was carried out to identify them, so that these can be extended to the CSV text format, which lacks ways of improving the processing time.

The pseudo-algorithms presented below are implemented by the importer, in C++. Therefore, when analyzing such algorithms, the execution path presented in the previous sections is considered have been carried out. They are executed by the importer, so file statistics and data source descriptions are considered to be available to this component.

The processing of ORC and Parquet files is done by loading individual file blocks and retrieving the rows which match the query predicate. Algorithm 1 describes how the importer retrieves file data for the latter formats. A handle is used to reference the file involved in a SoF query. It also may contain statistical information about the file columns, as a result of the optimizer retrieving statistics associated with that file from cache. The first step taken is to verify whether the query predicate match the statistical information associated with the file columns, in line 3. If not, then there is no need to load any file data, and the access information is updated.

Algorithm 1: Global query processing for ORC/Parquet files

```
   Result: A set of result rows that satisfy the query parameters
 1 Handle ⟵ data source description object;
 2 Rows ⟵ [];
 3 if !AreColumnFiltersValid(Handle.GetFileStatistics()) then
 4 |   UpdateAccessStatistics();
 5 else
 6 |   while RetrieveNextRowGroup(Handle) do
 7 |   |   while Handle.LoadNextColumnTableChunk() do
 8 |   |   |   rows ⟵ Handle.ProcessBlock();
 9 |   |   |   Rows ⟵ append(Rows, rows);
10 |   |   end
11 |   end
12 end
13 Handle.UpdateFileStatistics();
14 return Rows
```

This verification consists of creating value ranges for each column that composes the

query predicate, and comparing with the cached minimum and maximum values of the file columns. For instance, consider the predicate exhibited in Listing 4.2:

```
SELECT sensor_id, c_date, c_temperature
FROM sensorORC
WHERE c_date >= 2020-01-09 AND c_pressure < 1
```

Listing 4.2: Example of a query made upon an ORC file

For such query, the range `[2020-01-09, )` is created for the *date* column, and `(, 1)` for the *pressure* column, as per defined in its predicate. If the overall statistics of a given file says that it holds values greater than 3 for the *pressure* column, then there is no need to read the file data. Otherwise, the importer will proceed to evaluate the file blocks.

ORC and Parquet have a special section in their body which describes the positions of individual blocks. This is used by the importer to retrieve one specific block at a time, which is done inside the outermost while loop. The file handle is responsible for actually downloading the specific portions of the file containing file statistics. Blocks are retrieved in batches, as displayed in line 6. The innermost loop will call the processing function for each individual file block. A specialized reader for parsing the data will be used. As a result, a set of rows that matches the query is produced.

Finally, when all blocks are fetched and parsed, the final group of rows that satisfy the query is returned. Furthermore, some information about the file is updated and stored in cache, such as number of blocks read, number of rows skipped, among others.

Algorithm 2 describes in more detail how the importer parses file blocks to get rows that are included in a SoF query. It roughly corresponds to line 8 in algorithm 1. Technically, ORC and Parquet have specific ways of parsing the block data, as their inner structure are different. However, apart from these format-related methods, their reading is performed in the same way - this is what algorithm 2 summarizes.

The importer receives a data source description object along with the offset within the file that points to the beginning of the block. Firstly, the cached statistics for the file columns of the current block are compared to the query predicate, just as presented previously. This avoids loading blocks that contain irrelevant information.

For ORC and Parquet, the blocks are subdivided into smaller chunks, which allows a fine-grained loading of data. In the former, chunks are called "*strides*", which by default contain 10000 rows, whereas in the latter, they are called "*pages*". The importer loads them one by one, and decides which rows inside them have to be returned.

There are two types of columns considered while loading file data: *filter columns*, which are the ones specified in the query predicate (i.e., the *where* clause), and regular columns, which are the ones being selected by the query. For instance, in query 4.2, *c_pressure* is a filter column, *sensor_id* and *temperature* are regular columns, and *c_date* is both a filter and regular column.

A boolean vector with size corresponding to the length of the current chunk is used to mark which rows will be included in the query. Initially, its values are set to *true*, which represents that all rows will be included. The cached statistics are used to determine if the values of the filter columns in that chunk are completely inside the query predicate. If that is the case, then all rows will be included, and there is no need to find out which ones match the predicate. Contrarily, the filter columns are read in line 9, and their row values are compared to the respective column condition set in the predicate. Due to the columnar organization of the files, the importer can access columns individually, and it does so in

Algorithm 2: Loading ORC/Parquet file chunks for processing query results

```
Result: A set of file values that satisfy the query parameters
Input: data source description, block offset
Reader ⟵ ORC/Parquet file reader;
Rows ⟵ []
if query filters are not inside block column statistics then
    UpdateAccessStatistics();
else
    while Reader.BlockHasFurtherRows() do
        chunk ⟵ Reader.GetCurrentChunk();
        rows ⟵ boolean vector set to true
        if chunk column statistics are not completely inside query filters then
            parallel for each filter column
                Reader.ProcessRowsToInclude(rows);
            end
            wait for threads;
        end
        if rows is not completely set to false then
            parallel for each column
                Reader.ReadColumn(rows);
            end
            wait for threads;
        end
        Rows ⟵ append(Rows, Reader.GetRows());
    end
end
if statistics not present for file then
    Reader.ForwardColumnStatisticsToCache();
end
return Rows
```

parallel. For instance, the output of such procedure for the query 4.2 can be summarized in Figure 4.2

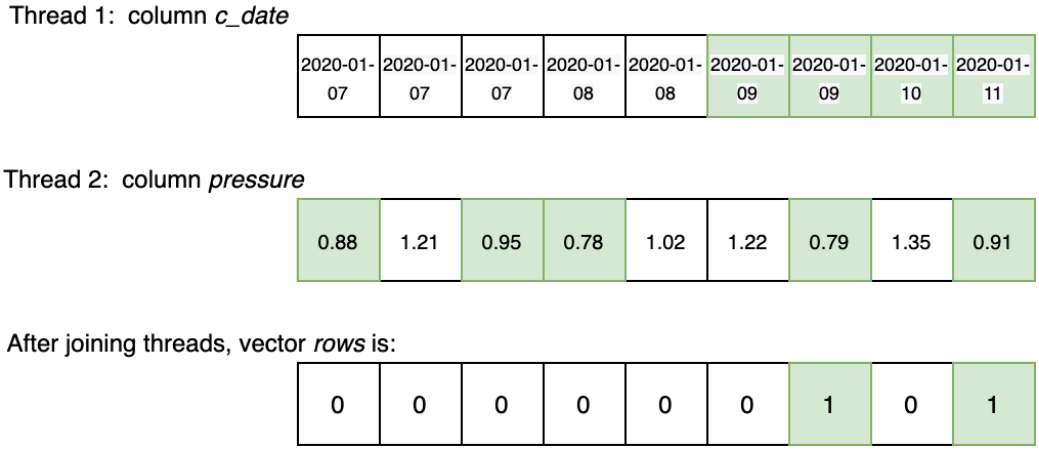


Figure 4.2: Filter columns are read in parallel, and their values are compared to the respective column predicates to determine which rows satisfy the query. Vectors represent a certain column, and each case a row value. Marked in green are the values that satisfy the query predicate set for the column, based on query 4.2

After this procedure, the importer has a vector describing which row in the chunk will be included in the query, and which one that will not. First, if the vector *rows* has no element set to “true", then no row in that chunk match the query - hence, no need to load any value. Otherwise, the regular columns are read in parallel to retrieve the rows specified by the vector. Again, the columnar organization allows the parallel read of data.

If a given file is processed for the first time by the importer, then the statistical information it contains is collected while the chunks are read. These statistics are stored in cache at the end of the query, as expressed in line 26. Consequently, if a query is executed again upon the same file, this information can be used to skip downloading unnecessary blocks.

Another optimization adopted by the importer is the download of upcoming blocks in the background: while the first block begins to be parsed, the download of the second one begins. Usually, ORC and Parquet blocks sizes are approximately the same, so the execution is bound to the time it takes to download chunks from the data lakes.

The biggest advantages of processing such files are:

- Parallel read of columns, due to the columnar file organization, which enhances performance while reading file blocks.

- Application of file pruning, as represented in line 3 of algorithm 1, and predicate pushdown, as shown in lines 3 and 9 in algorithm 2. As the statistic cache holds minimum and maximum values of different columns, they can be used to skip chunks and file blocks, and thus minimize the amount of data processed.

- Greater level of customization, as ORC and Parquet files can be constructed with different row group and chunk sizes to reflect a specific characteristic of the dataset encapsulated by the file.

- Possibility of using compression methods in blocks and chunks, to decrease the quantity of downloaded data. This comes with a trade-off: while parsing the blocks, the information will have to be decompressed. Usually, the most suitable compression method varies with the type of workload being executed.

On the other hand, some of its weaknesses are:

- Slow start of the statistical cache. When reading for the first time a file dataset (e.g., a huge folder with multiple files in it), no statistical information is present in the engine. This may induce longer reading times, as extraneous blocks can be fetched and processed.

- File modifications invalidate the existing statistical information. If a source file which has its column statistics cached in Vora is changed, then a different ETag will be generated for it, causing the importer to ignore the existing statistical information and gather new values when processing it. Hence, SoF queries are not optimized for files whose content is changing constantly (i.e., *hot data*).

- Execution is bound to the available memory in the worker node. As the blocks are processed in main memory, a worker node can only process so many file blocks at a given time. This bottleneck can be alleviated by adding more worker nodes to the Vora cluster (i.e., *horizontal scaling*).

4.3 Analysis of CSV processing algorithm

The processing of ORC and Parquet benefits from the usage of column minimum and maximum statistics, as they are used to skip blocks and chunks. This increases processing time for the SoF use cases, which targets highly selective queries made upon static files. Hence, the statistics are used by the importer to load data effectively, by means of file pruning and predicate pushdown. They serve to migrate computations to the portions of the file that contain the needed data.

CSV, on the other hand, does not present such features. Being a text format, there is a large overhead for transforming parsing the data and interpreting the encodings. It also does not benefit from a columnar organization, which means that columns can not be processed independently. For instance, if a CSV file has 10 columns and only 2 are being targeted by the query, when parsing a file row, all the columns have to be parsed too, as the delimiters have to be found in order to determine the column positions.

The following sub-sections will present how CSV was processed prior to the implementation of proposed enhancements, and the characteristics that motivated them.

4.3.1 Reading CSV files

This section presents how the importer loaded CSV files prior the introduced modifications. CSV files are also processed in chunks. However, different than ORC and Parquet, the file body is not inherently divided in row groups. Hence, the importer loads CSV files by blocks. In production, the block size is 100MB, which represents a compromise between processing times and network latency. If the block size was too small, then the system would make more requests to fetch the file data, and thus the performance would suffer from many accesses to the data lake, resulting in a high latency overhead.

On the other hand, if the block size was too big, then the importer would stay idle while the block is transferred from the data lake.

Other than being downloaded and processed, CSV blocks need to be parsed. This is because they contain textual data, tied to a specific character encoding. Different char sets are supported (for instance, Chinese and Japanese characters are allowed). Values need to be transformed from a string representation to the actual column type. For example, a CSV column might contain the text value `"true"`, which has to be interpreted by the importer as a boolean value. This extra calculation affects the performance of queries executed upon CSV.

While a given block is being processed, the upcoming one is downloaded in the background, as shown in Figure 4.3. The file processing steps are comprised of parsing the textual data, transforming it to value supported by Vora, and applying the query to it, which involves actions such as selecting the needed columns and applying column filters.

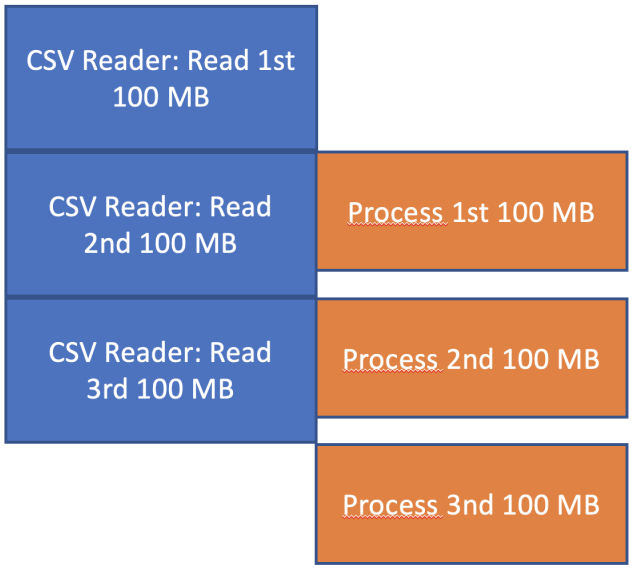


Figure 4.3: CSV blocks are fetched in the background while others are being processed. The latter procedure includes parsing the text data and filtering the data needed for the query

Since parsing is a relatively costly operation, a block is further divided into row groups, which are parsed in parallel. Figure 4.4 illustrates such division. In order to extract complete sets of lines from chunks, the binary data is interpreted as a string of characters, and the end-of-line markers "`\n`" are located. By applying this process, groups composed of complete lines can be formed, which simulates the concept of row groups. They are used to read the column values.

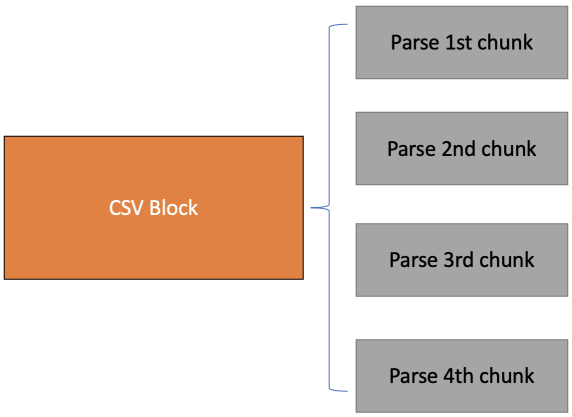


Figure 4.4: Blocks are subdivided chunks, which are parsed in parallel.

During this processing, a concern for the importer is that not necessarily blocks contain entire lines. When a is block read from the data lake source, the importer has no gurantee that it has a full line, nor that all the lines it contains are complete. They initially have no semantic for the importer - they are just a fixed-size sequence of bytes. Therefore, the CSV reader can process blocks that possess partial lines, so it has to locate the position of complete lines in a raw file block, and keep track of their positions in the file stream.

Figure 4.5 illustrates the processing steps mentioned in the previous paragraphs. Initially, a CSV file - located on a data lake - is accessed, and its first 100MB block is downloaded, as indicated by number 1 in the figure. After this step, a raw block of bytes is available for the importer. Next, the end-of-line markers will be located to define how many lines are present in the block. The total number depends on the amount of columns and the values they hold.

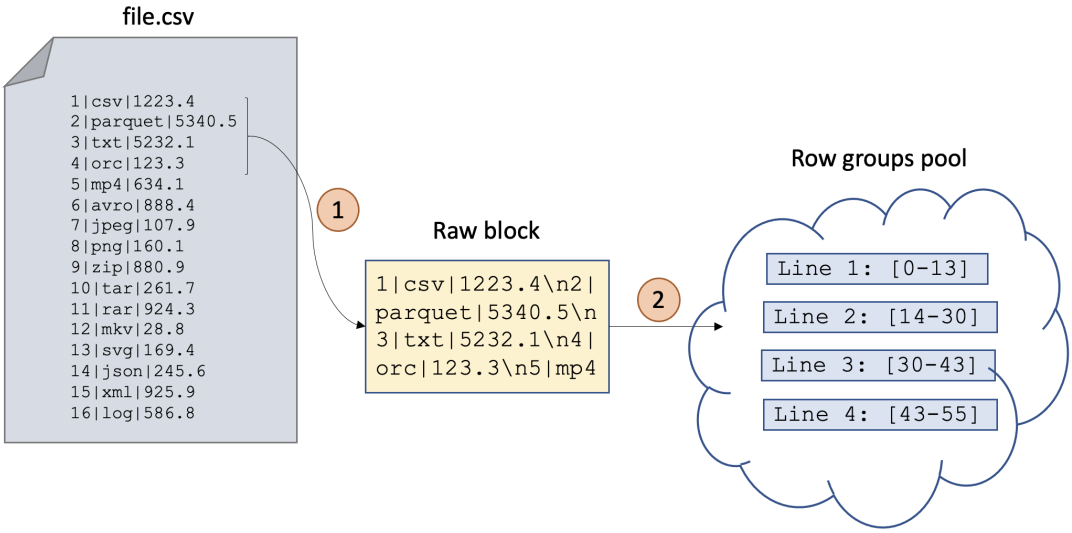


Figure 4.5: The original CSV file (on the left), located on a data lake, is used to fetch a first block. End-of-line markers are located, and row groups are formed.

The importer leverages the capabilities of Intel's SSE (Streaming SIMD Extensions) instructions to enhance the search of end-of-line markers in the block. SSE extends the instruction set architecture to include operations that execute on multiple data, which fits

Flynn's SIMD (Single Instruction Multiple Data) classification (FLYNN, 1972). With that approach, multiple blocks of data can be compared in parallel to a mask containing the end-of-line markers ("`\n`"), and thus the offsets representing the beginning and end of lines are located. This alternative yields better performance results if compared to reading each character and comparing them individually to "`\n`".

As the line positions are found, groups of lines are formed and put in a task pool, which will be processed in parallel. This scenario is displayed in Figure 4.6. The representation of such groups is done by a structure called `CSVParallelReadingRangeItem`, which simulates a row group of a big data format. It encapsulates the beginning and the end file offsets of the lines composing a given group, and it may contain up to 10.000 rows or at most 20MB. This boundary condition was a design choice made by the importer, to avoid having groups containing few lines, which would result in a big number of tasks to be executed, or groups having all the lines in the current block, which would eliminate the benefits of a parallel task execution.

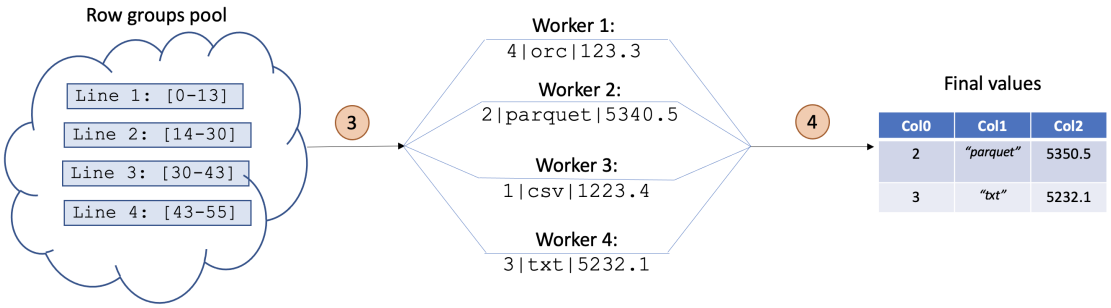


Figure 4.6: Row groups are passed to worker threads, which will process the group lines. In this example, each row group contains only 1 line, but normally thousands of lines are present in them.

The library **Intel Threading Building Blocks** (TBB) is used to implement the parallel execution of tasks. After forming the row groups, an instance of the library class `tbb::task_group` is responsible for processing them. A pool of worker threads is instantiated, and each row group is assigned to a worker, whose objectives are reading the appropriate column values, applying projections and comparing them to the query predicates. A barrier waits for all the threads to complete, and finally a group or rows that satisfy the query is returned.

In Figure 4.6, the predicate selects rows which have a value greater than 5000 in their third columns. Consequently, workers 1 and 3 will discard their rows, and the result will be created with the values processed by the other two workers. Also, in the beginning of the execution, the block contains a fraction of the fifth line, as shown in Figure 4.6. The importer, however, will ignore this portion while processing the first block, and concatenate it to the beginning of the second block when processing it.

The advantages of the implementation described above are:

1. The importer leverages the capabilities of Intel TBB to process the row groups of each block in parallel.

2. SSE instructions are used to alleviate the overhead of finding the end-of-line positions in the file stream, which is a challenge inherently associated with text formats, such as CSV.

3. Even though each block is processed sequentially, while a block is being parsed, the next one is fetched from the external source in the background.

Nevertheless, it also has some drawbacks:

1. There is no support for predicate pushdown. Therefore, in every execution, the entirety of the CSV file is read. This fact also extends to file directories: queries executed upon them will forcibly cause the importer to fetch and read all the file data.

2. The importer does not cache any information about the data read from the file blocks. Hence, it is not able to skip the loading of individual blocks or row groups, as observed in the big data formats.

3. CSV requires heavy string manipulations, as it is a text format. Therefore, the importer should avoid having to parse blocks that do not contain the needed query information.

5 PROPOSED SOLUTIONS

Analyzing how the importer handles the processing of big data formats and comparing it to the CSV format processing highlighted some improvement points and techniques that can be applied to the latter module. This chapter will now present what was implemented to optimize the performance of queries made upon CSV files.

5.1 Enhancing CSV processing time

The following modifications were implemented in the CSV processing module of the importer:

- Statistical calculation for CSV files. When reading a file for the first time, the maximum and minimum values of the columns for whole file and for the individual blocks will be computed and stored in the statistical cache.
- Partitioning CSV blocks in smaller chunks while reading them, with the help of Intel's TBB parallel computing library.
- Given the enhancements above, the technique of **file pruning** was added when performing queries upon CSV. Therefore, files that do not possess relevant data for the query can be discarded altogether.
- **Block pruning** for CSV files, which consists on extending the file pruning to individual blocks. This will allow the importer to have a better control of which portions of the file needs to be ignored.
- Also, use the minimum and maximum statistics to ignore the download of unnecessary blocks, decreasing the amount of data transferred over the network.

5.2 Parallel reduction of CSV blocks

The importer leverages cached minimum and maximum column statistics for enhancing the processing of ORC and Parquet files. Thus, in order to have the same benefits for CSV files, these statistics have to be collected while reading the CSV file, and passed to the statistical cache, given that they are not naturally present for the format.

Although a parallel data read results in a significant gain in performance, it poses some difficulties to the minimum and maximum column value calculation. For one, different threads will be responsible for separate sub-ranges of the file. While doing its work, one thread may find that *10* is the maximum of the first column, for instance, while another one

may find that *7* is the maximum value. If not designed properly, the statistical calculation for CSV files may result in race conditions and produce incorrect values.

An additional concern is the performance of the CSV reader. Parsing textual data is for itself a costly operation. Adding a statistical calculation on top of such processing might increase the time it takes to read the CSV rows, since individual column values have to be tested against the minimum and maximum statistics.

The nature of the CSV statistical calculation requires a certain processing to be made in multiple ranges of the file block, which will result in minimum and maximum values of each column on the sub-ranges. Moreover, these results have to be compared against each other to finally produce the overall minimum and maximum values of each column in the whole block. These steps suggest that a *reduce* operation should be employed, since a set of smaller outputs must be combined to form a final result.

Therefore, the implemented solution relies on TBB's **parallel_reduce** routine, which replaces the usage of a task group. This function computes a reduction of a given *range* by recursively splitting it into sub-ranges up to the point such that each sub-range is not divisible anymore. The divisibility criteria depends on the granularity set by the function caller. Once a sub-range is granular enough, the library assigns a *task* to it, and the processing starts.

The reduction function can be represented by a class containing the reduction logic, expressed by an `execute()` method. An object of such class will be mapped to a *task*, which is a library construct. Tasks are typically a small routine which are mapped to logical threads. According to Intel's official documentation (Task-Based Programming, Intel TBB Documentation, 2021), tasks are much lighter than logical threads, as they do not need to keep track of system resources such as register states and stack. Thus, tasks have the advantage of being higher-level constructs that encapsulate a single routine, which are consumed by threads. Instantiating and terminating a task in Linux systems is about 18 times faster than doing so with a thread, according to the same reference.

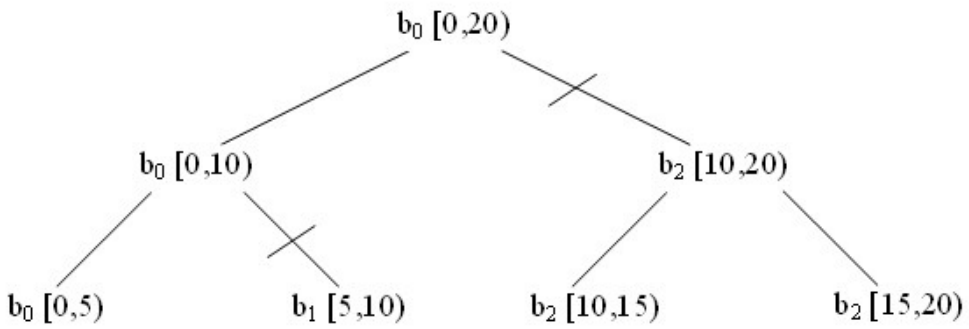


Figure 5.1: An hypothetical split of a block range done by the TBB library. *Source: Intel Developer Zone: https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation*

Once mapped to a thread, the task is bound to it until the processing returns. During that time, the task may wait for the completion of some child task or nested parallel construct. In that occasion, the thread assigned to it may run any available task, including unrelated tasks created by other threads, instead of sitting idle. TBB's internal task scheduler is responsible for doing such work.

For every range split, a *join* method is invoked after the respective tasks finish, in order

to merge the results from the sub-ranges. It is the caller's responsibility to define a proper implementation for the join procedure.

In Figure 5.1, an hypothetical split of a range is shown. Consider that initially there is a block that ranges from byte 0 to byte 19, and that the granularity is set to 5. Once passed to the *tbb::parallel_reduce* routine, a task b_0 is associated with the initial range. Since it is not granular enough, it is split in half by the library and a new sub-range from 10 to 19 is assigned to a task b_2. The splitting is done recursively until the sub-ranges attain the desired granularity, at which point the task starts processing the range bytes. In the example, there are 4 leaves representing the final sub-ranges. They are evaluated by three separate tasks. Notice that task b_2 ends up evaluating two sub-ranges: this is one possible execution, as the splitting depends on factors such as the amount of cores and worker threads available. At the end, the following join operations will be invoked:

1. Range `[0,5)` will be merged to `[5,10)` through the medium of *join()*; The same goes to ranges `[10,15)` and `[15,20)` . The order that such merges occur are non-deterministic.

2. Range `[0,10)` will be merged to `[10,20)` via *join()* after the previous merges have occurred.

Finally, once the last join occurs, a single task will remain, and the result of the reduction will be returned to the caller.

5.3 Implementation of statistical calculation for CSV

The parallel reduction computation is used to read the CSV blocks concurrently, and while doing so, calculating the CSV statistics for the columns they contain. This section describes what changes were done to the processing described in 4.3 to implement the statistical calculation for CSV.

TBB defines a signature for classes that can be used with *tbb::parallel_reduce*. Therefore, a new class ***CsvReaderTask***, which encapsulates the operations needed for parsing the sub-ranges, was introduced. It contains a `execute()` method which performs the steps needed to process lines - parsing characters, calling the respective column readers to read the data and applying the query filters.

After locating the end-of-line inside a block, lines are individually put into instances of a structure that holds their beginning and end offsets. Then, these objects are put inside a range, which will be passed to **tbb::parallel_reduce** to begin the processing. After splitting the initial range recursively, a granular sub-range will contain a group of lines to be parsed. At this point, an instance of *CsvReaderTask* will be assigned to a task, which calls the `execute()` method. At that instant, there are multiple tasks concurrently reading the values of CSV lines.

The statistical calculation is introduced in this phase: each *CsvReaderTask* object will contain an auxiliary structure, called ***CsvStatisticNode***, responsible for keeping track of column statistics on a given sub-range. It possesses two attributes for storing the minimum and maximum values of a certain column section of the file. Therefore, every time a task reads a certain value, a local copy of this object is consulted to evaluate if such value represents a minimum or maximum statistic for the given column. Once the task finishes, a vector of *CsvStatisticNode* will have the minimum and maximum statistics of each column for the given sub-range.

When two sub-ranges processed by some tasks are merged, the *join()* operation will compare the vectors of*CsvStatisticNode* to compute the minimum and maximum statistics for the joint range. This sequence of joins is done recursively until the full range is reconstructed, at which point the column statistics of the whole block are returned to the caller.

Since TBB controls task execution and the merging of tasks, race condition issues are eliminated. Intermediate column statistic values are combined by means of the *join()* method, which is invoked properly by *tbb::parallel_reduce*. Therefore, *CsvStatisticNodes* can be compared in correct manner.

Figure 5.2 shows the processing of CSV blocks by *tbb::parallel_reduce*. Lines are found inside the block and placed in a vector. Each element will contain the beginning and end offsets of the line in the file stream. Then, the TBB library is responsible for splitting the sub-ranges. Once they reach a certain granularity, line parsing is done. The operations performed in that stage are the same as described in section 4.3. The only additional step is the statistical comparison performed with the help of *CsvStatisticNode* objects. This addition will incur a certain overhead, which will be measured in the next chapter.

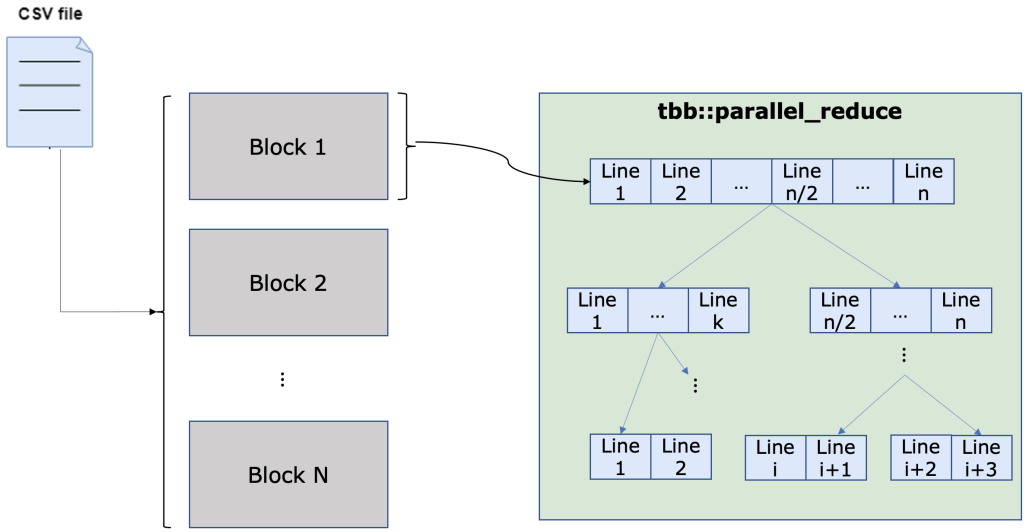


Figure 5.2: Line positions are located inside a block, and a range is formed. When passed to *tbb::parallel_reduce*, sub-ranges processed concurrently by tasks

,

The calculation of column statistics is depicted in Figure 5.3. When a task processes a sub-range with an instance of *CsvReaderTask*, the *CsvStatisticNode* objects are consulted every time a new value is read from the lines. Since different tasks will have local copies of these objects, race conditions are not injected during this computation. After processing their lines, tasks are joined, and as a result, column statistics are combined

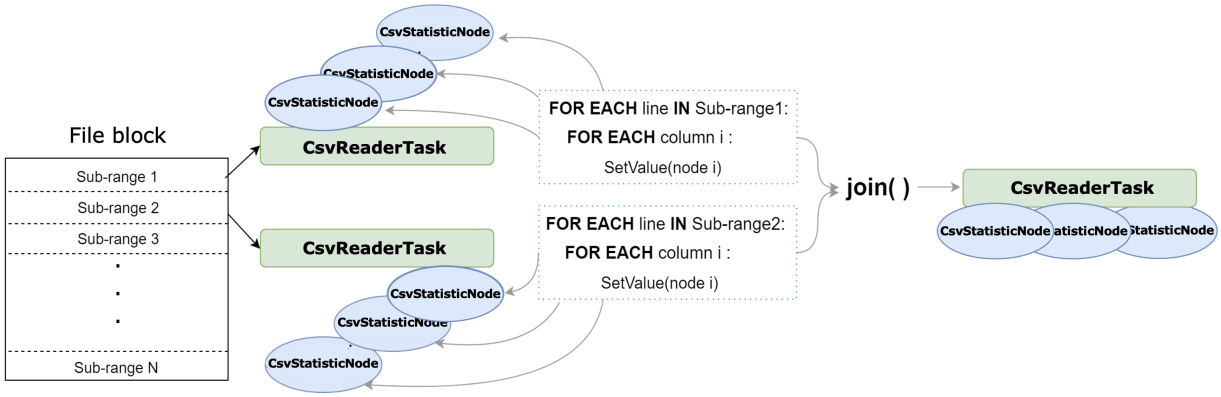


Figure 5.3: Sub-range column statistics are computed and combined after a *join()* operation.

5.3.1 Granularity of sub-ranges

The *granularity* parameter controls the size of sub-ranges upon which tasks will perform their work. A certain range is considered to be divisible if its size exceeds the granularity passed to the parallel reduce operation. However, TBB might produce ranges containing fewer elements than the specified grain size. According to the official documentation, the task partitioner will ensure that the sub-range size sits between `G/2` and `G`, where G is the granularity.

Therefore, the parameter affects directly parallelism, since it defines a partition of the input range, which will be distributed among tasks. Defining a large value for the parameter may unnecessarily restrict parallelism: each task will have to process a big amount of lines, instead of sharing this work with other threads that may be available. On the other hand, a small granularity might result in many tasks being created, introducing a scheduling overhead.

In the context of CSV block processing, the granularity criterion was defined in terms of *block size* rather than *number of lines*. This is because different CSV files may have lines possessing variable sizes. For instance, it the file defines hundreds of columns, its lines will have a greater size than those of another file that has only one column per line, as more text translates into a bigger line size. Therefore, processing different files would produce distinct number of sub-ranges, which could affect parallelism.

Figure 5.4 shows how many sub-ranges would be created for a 100MB block if the granularity criterion was "*number of lines per sub-range*". Three CSV files are considered, with varying line sizes: 20, 100 and 500 bytes. If the granularity was set to a 5000 lines, for instance, the red file, whose lines have 20 bytes in size, would produce 1000 sub-ranges. On the other hand, for the same parameter, the green file which possess 500 bytes lines will have only 20 sub-ranges created. This demonstrates that choosing *number of lines per block* as the granularity criterion would affect the parallelism observed in the processing of CSV files. For that reason, a different criterion was selected: ***amount of bytes per sub-range***. This guarantees that, no matter the line size, a similar amount of sub-ranges will be created for every CSV file being processed.

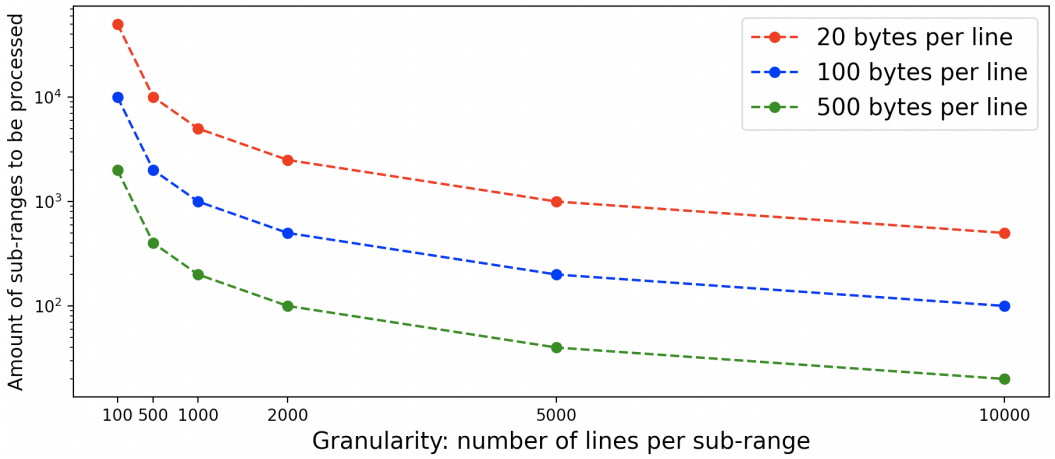


Figure 5.4: Graph showing how many sub-ranges would be created by parallel reduce operation for a 100MB block, depending on the line granularity. Y-axis is in logarithmic scale. Different kind of files with varying sizes of line are shown.

The following experiment was conducted to define a fitting value for the amount of bytes a sub-range should have:

1. Initially, a high value was picked (e.g., 10MB)

2. The *tbb:parallel_reduce* operation was executed on a 100MB CSV block for a fixed number of iterations (e.g., 30). The execution times were averaged to get a final number.

3. Finally, the size was iteratively decreased, and the last step was executed again.

Graph portrayed in Figure 5.5 displays the measured execution times:

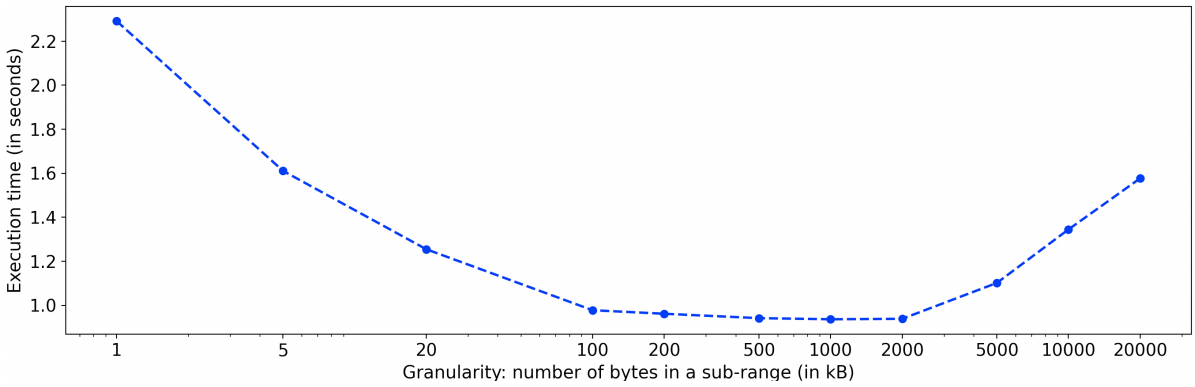


Figure 5.5: Graph shows the execution time for processing a 100MB CSV block in function of the number of bytes present on each sub-range (granularity) .

Therefore, the experiment shows that the execution time lowers as the granularity value approaches 100KB. From that point on, increasing the grain size has little impact on the measured processing time. The value of 1000kB, however, yields the best execution time (0.927s). This value was chosen as the block granularity for the processing of CSV files

5.3.2 Block and file pruning for CSV files

With the calculation of statistical values for CSV columns, the **file pruning** capability of the relational engine could be extended for CSV files. Initially, the importer was modified to forward the statistical information about the whole file to the engine. This could be achieved because an interface defined the operations needed to interact with the file statistics cache. Therefore, after this first step, entire files could be skipped if the query predicate did not match the overall maximum and minimum column statistics for the file.

Even tough such modification already represented an improvement, there were still scenarios in which the statistics were not efficiently used. For instance, if only one block of the file contained relevant data for the query, the importer would still read the entire file, and perform unnecessary work reading the remaining blocks. That is because the *block wise* statistics were not being forwarded to the cache. Therefore, the second improvement was to also cache the statistics of each block. With that addition, **block pruning** was introduced, and the importer could achieve a finer-grained pruning while reading the file, as unnecessary portions could be skipped. Moreover, the block statistics can be used to avoid not only the *parsing* of irrelevant data, but also *downloading* such kind of information.

Figure 5.6 shows the class diagram for the implemented solution. The important point to notice is that the existing interface *CImporterCacheData* was implemented by a new class *CImporterCacheCsvData*, that allowed file pruning to occur.

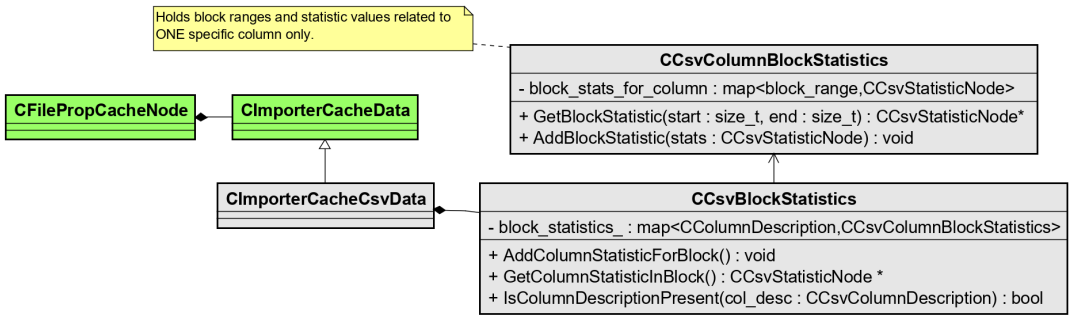


Figure 5.6: Class diagram for the file statistics cache in the importer. Classes in gray were introduced to implement block statistics.

The cache entry associated with a given block statistic is identified by the file ETag concatenated to the block's byte range. Thus, the importer can individually retrieve statistical values for the block being processed. This also means that, if the file content was changed at the data lake source, its ETag would be modified, invalidating the calculated statistics for that file.

The statistical calculation described in this section is only executed if a certain block does not have a statistic entry in the cache. This avoids having to recalculate statistics for blocks which already have a valid up-to-date entry in the file statistics cache. Moreover, given that statistics are related to a certain block range, the implementation will generate new statistical values every time the block size is modified. For instance, when a CSV file is processed with block size configured to 100MB, a set of statistical column values is computed and stored in the cache. If the block size is changed to 20MB, and the same file is read, then statistics for the 20MB blocks will be calculated. This makes the proposed solutions extensible for many configurations of the importer.

6 RESULTS AND PEFORMANCE EVALUATION

The modifications added to the importer module make predicate pushdown available for CSV files through column statistic calculation. Nonetheless, the usage of threads and auxiliary comparisons for computing minimum and maximum values introduces an overhead when reading a file for the first time.

Therefore, this section will be dedicated to evaluating the performance variations between the original implementation, in which no statistic calculation was done, and the one containing the new statistical features. Moreover, the performance of analytical queries will be measured to observe how the execution time was impacted.

6.1 Statistical calculation overhead

The proposed modifications represent a trade-off to be made: at the expense of an extra step while reading CSV blocks - calculating minimum and maximum values for each column - unnecessary accesses to files and blocks can be avoided in subsequent executions.

In order to better examine the extra cost for calculating statistics, execution times for data profiling queries (as in Pseudo-code 6.1), which selects all file rows, were measured. A bare-metal *Vora* cluster with a single node (Linux, 6-core Intel Core i7, 32GB main memory) was used in this test. Two variations of the importer were taken into consideration:

1. Statistical calculation for CSV is turned off; this corresponds to the original implementation of the importer.

2. Statistical calculation for CSV is enabled; thus, all modifications presented in the previous chapter will be included in the importer.

Pseudo-code 6.1: Data profiling query used in test

```
SELECT MIN(col1), MAX(col2), SUM(col3), COUNT(DISTINCT col4)
FROM csv_table;
```

Initially, a CSV file extracted from a production system was used. It possessed a total size of 583.8 MB, and 34 columns of different types, such as `DOUBLE`, `INTEGER`, `TIMESTAMP` and `VARCHAR`. The file was put in the local cluster node, so the latency of fetching blocks from an external data lake does not affect the query execution time. Five “warm-up" iterations were executed, followed by 20 executions of the query, which were the ones used to define execution times. Figure 6.1 presents the results gathered for

both importer variations. It can be observed that the overhead introduced by the statistic calculation was approximately **3%** - this accounts for the extra computation that the reader tasks have to perform. The average query processing time for each implementation is represented by a dashed horizontal line.

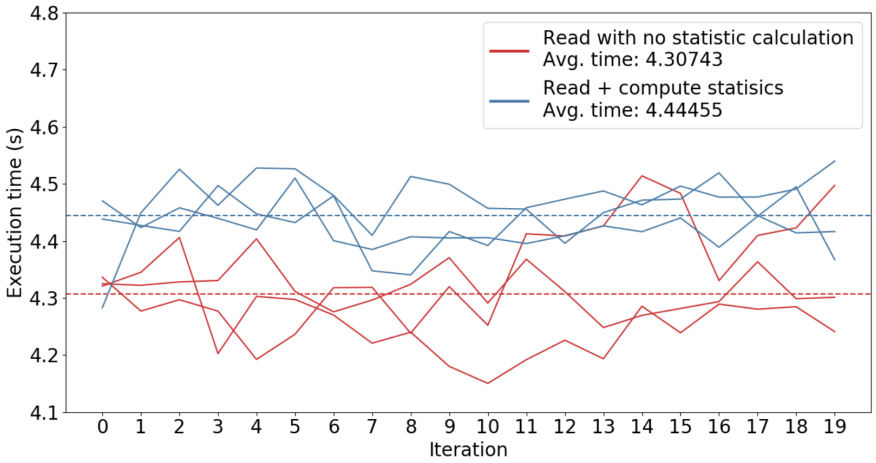
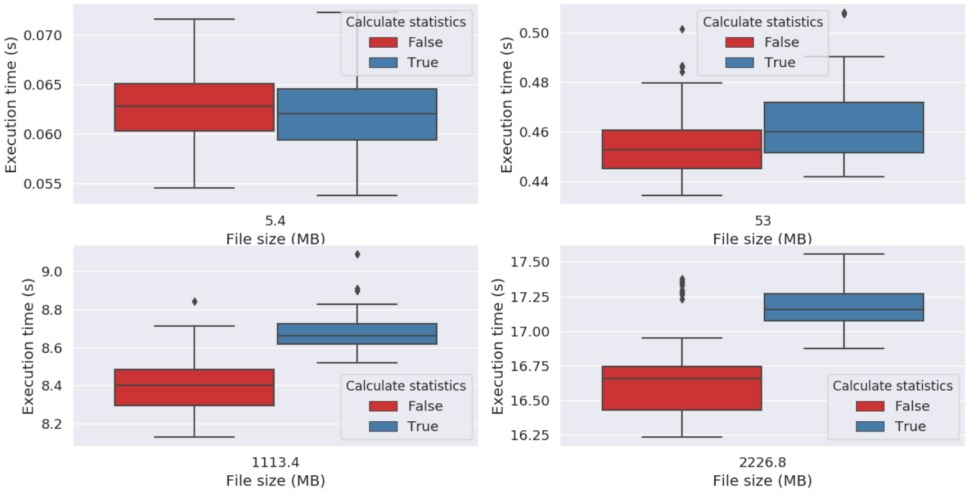


Figure 6.1: Test instances executed on a 583.8MB file.

The test was also performed with 4 additional files of different sizes, generated artificially by a script. They contained 14 columns: 4 of them had an integer type, 3 of them double, followed by 2 timestamp and 4 string (`VARCHAR`) columns. They were also stored in the local cluster node. Execution times for the test instance is displayed in Figure 6.2.



File Size (MB)	Average time with statistic calculation	Average time without statistic calculation	Overhead
5.4	0.0618621 s	0.0630054 s	-1.8146%
53	0.4631354 s	0.4552393 s	+1.7345%
1113.4	8.680269 s	8.41722 s	+3.12514%
2226.8	17.177008 s	16.66392 s	+3.07904%

Figure 6.2: Execution times for different file sizes. Every colored box encompasses results for 3 instances of the test.

The median is depicted by an horizontal line, outliers are displayed as dots, and the average execution times are shown in the annexed table. The observed overheads for statistic calculation fluctuated around **1.7%** and **3.1%**, demonstrating that the extra cost introduced by the proposed solution remains fairly stable when executed upon different file sizes.

Notably, when the file size is small, the additional calculation performed by the reader tasks is not substantial enough to affect execution time. The graph presented on the top left corner of Figure 6.2 shows that the test instances performing the statistical calculation presented a slightly smaller execution time than the ones not executing it, which in theory were supposed to run faster.

When the amount of data increases to substantial sizes (such as 1GB and 2GB), the overhead observed was approximately 3%, in the same margin as the one measured for the original version of the file, with 583.8MB.

6.2 File pruning in directory queries

The main motivation for implementing predicate pushdown and file pruning for CSV files was to minimize file accesses and irrelevant data processing when executing analytical queries. One way of evaluating the effectiveness of the implemented statistical calculation was to execute queries upon entire directories.

The intent behind this type of test is to select a reduced portion of records contained in the folder, as to check if files not concerned by the predicate were pruned by the engine, based on the calculated column statistics.

6.2.1 Single-level range partitioning

The dataset used in this test instance consisted of a directory containing CSV files placed in different sub-folders. The data was divided among files following a single-level range partitioning, in which one column was used as the partitioning key. Files in the directory contained disjoint ranges of values in that column, and were initialized in the following manner:

1. An initial range `R = [0,b)` was chosen, where `b` is a positive integer. Files in the directory would encompass values within that range.

2. A partitioning column *p*, which held values of `R`, was defined. It partitioned the initial range into `N` disjoint sub-ranges, and each one of them was put inside a different file.

Given the data partitioning scheme, the test folder was initialized by choosing `N=16` and `b=10000`. It was structured as a balanced binary search tree with height 5 - thus having 16 leaves, which represented the actual CSV files containing the data. Each file had 1MB in size, and possessed only the partitioning column. Figure 6.3 shows an example of a folder with the same organization, but with `N=4` (thus, with height 3).

The test consisted on executing queries selecting a particular range for the file column. Statistical calculation for CSV blocks was enabled, and the directory was uploaded to *Google Cloud Storage*. Pseudo-code 6.2 shows the test query syntax.

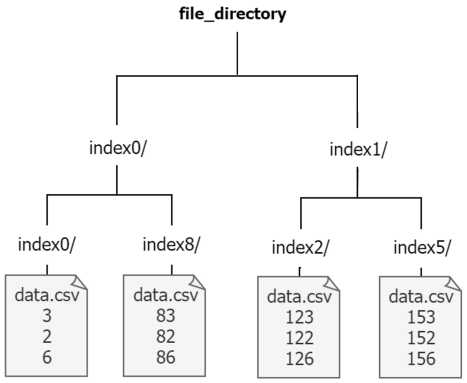


Figure 6.3: Files are represented as leaves in the directory tree, and each node represents a folder. Files hold the values of a exclusive sub-range, and no value is observed in more than one file

.

Pseudo-code 6.2: Query syntax

CREATE TABLE csv_partitioned (
col0 BIGINT
) WITH TYPE **DATASOURCE ENGINE** 'FILES';

ALTER TABLE csv_partitioned **ADD DATASOURCE** CSV('file_directory')

SELECT col0 **FROM** csv_partitioned **WHERE** [**predicate**];

Table 6.1 shows the result for multiple executions of the test query, according to a specific predicate that targeted a portion of the file. As a result, it can be observed that the file statistics are used by the importer to skip files that are irrelevant to the query. This situation is more evident in equality predicate, for instance: only one value is selected, so the importer will use the statistics to define which file encompassed the needed data, and only this one will be processed.

Table 6.1: Percentage of data selected in the folder compared with number of files read by test queries

Predicate	Percentage of data in folder selected	Number of files read
`col0 > 0`	99.99	16
`col0 != 0`	99.99	16
`col0 > 625`	93.74z	15
`col0 > 4999`	50	8
`col0 < 625`	6.25	1
`col0 <= 625`	6.26	2
`col0 > 9374`	6.25	1
`col0 >= 9374`	6.26	2
`col0 = 100`	0.01	1

On the other hand, the original implementation generated 16 file reads for every query - even if only a small percentage of the data was required for the result. In conclusion, the benefits of having file statistics are shown by this test instance, as the implemented

solutions resulted in less data being read by the importer while processing queries. Although the data was artificially generated, there are real-world scenarios in which file data is partitioned by a certain key (e.g., sales record per month).

6.2.2 Composite partitioning on a large dataset

While the previous test showed that file accesses can be avoided, the dataset size was not considerable. In this test instance, IoT sensor data was sampled from a *Vora* customer dataset, forming a folder with 12.1GB in CSV files. The goal is to measure the gains in terms of execution time when querying data from a sizable directory source.

The directory is structured following a *composite range-range partitioning* technique. In this approach, data is first mapped to partitions based on ranges of values of a first-level partitioning column. Then, each generated partition is further divided in the same way, following a second-level partitioning column. Thus, this organization enables a logical range division along two dimensions.

The folder containing the test dataset was hosted in *Amazon S3* platform, and a two-node *Vora* deployment in *Amazon EC2*, Amazon's cloud-computing hosting platform. Machines were equipped with 32GB of memory and 16 virtual CPUs. Files were put in the same region as the one in which the cluster was instantiated, in order to reduce network latency.

Particularly, the test dataset stored measurement values for three different sensors on a three-year observation period. The underlying file schema presented 16 different columns, with types `INTEGER`, `DOUBLE`, `TIMESTAMP` and `DATE`. A first-level partitioning was created using the `sensor_id` column: since there were three devices, their values were put in separate folders. Then, a second-level partitioning was created on the `month` column, creating three distinct sub-folders: the first one with values from January to April; the second, ranging from May to July; and the third one, holding values from September to December. Thus, 9 CSV files were present. Figure 6.4 illustrates the folder organization.

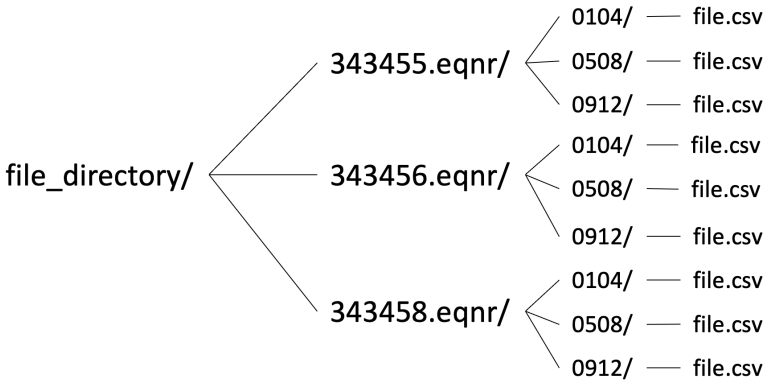


Figure 6.4: A first-level partitioning was created on the "*sensor_id*" column, followed by a second-level partitioning on the "*month*" column

.

Five queries with different levels of selectivity were executed upon the directory:

1. Select all entries measured in December by sensor *343458*. Only one file holds values for this query, with size 1242MB - thus, 12 blocks of 100MB and one with 42MB to read.

2. Select all measurements gathered in January. Three files and 36 blocks are expected to be processed.

3. Values for sensors other than *343455*. Six files hold the query result, with a total of 115 blocks.

4. Measurements gathered in 10th of June by sensor *343455* whose ingestion time was greater than 23PM. Only one file holds values for this query, and since it has a high selectivity, only one block of that file should be read.

5. Values measured by sensor *343458* on January. Only one file has the required data, which is located in the first 4 blocks.

Figure 6.5 shows the query execution times and the number of blocks that contain the required data, represented by the bars and by the dashed yellow line, respectively. Two importer variations are considered: the original one and the version with block wise statistical calculation. Queries were executed on a fresh cluster - thus, the file statistic cache was initially empty. Moreover, each query was executed 10 times, and their results were averaged.

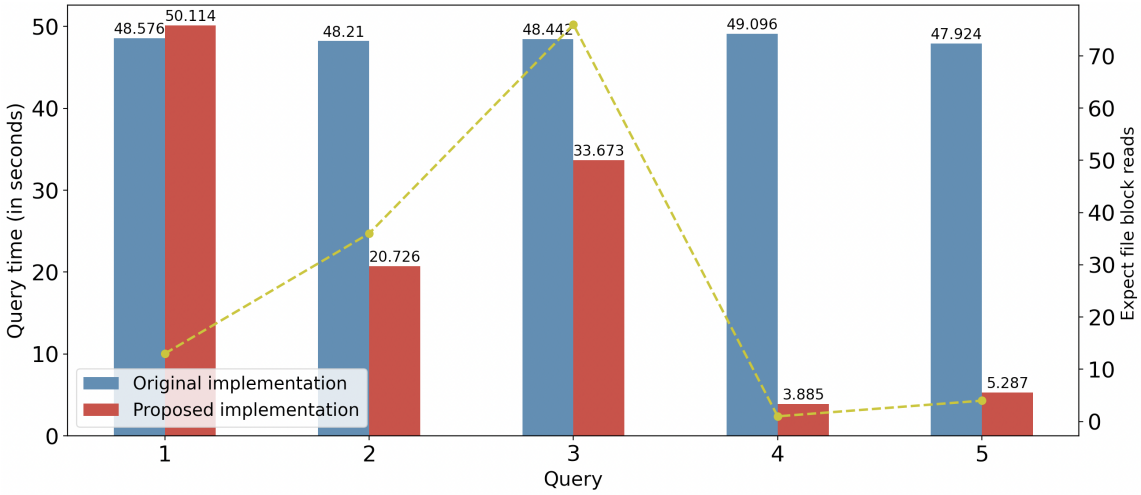


Figure 6.5: Execution time and expected amount of file blocks to be read for each test query.

In the first query, the proposed implementation has a higher execution time - *50.114s* versus *48.576s* on the original version, even though the amount of blocks to be read is relatively small. This is due the block statistical calculation being executed for the first time, since there's no information of column minimum and maximum values. After this query, block and file statistics are cached, which impacts the subsequent queries.

In the second query, there are 36 blocks to be read. While the original implementation takes approximately *48s*, the version with statistic calculation takes *20.726s*, as irrelevant files are pruned, and only the needed file blocks are processed. The next query has a higher selectivity, as 76 blocks are to be read. Therefore, execution time increases in the proposed implementation version, as more blocks are processed.

The greatest improvements are observed in the last two queries, which possess high selectivity while block statistics are present. They involve reading 1 and 4 blocks, respectively, and so execution times are decreased when they are executed with the proposed implementation. In opposition, the original version still takes around 48 seconds to execute the queries.

6.2.3 Random and highly-repetitive file data

The last two tests executed queries on directories possessing a certain logical distribution: one or more partitioning keys were defined, and files were divided in directories that encompassed a given sub-range.

However, there are datasets in which no logical division exists. In this sub-section, the proposed modifications will be tested against files that possess highly repetitive data (e.g., column with the gender of a person) and random distributed values. A test dataset which included these kinds of columns was generated. Particularly:

- A column *"gender"*, which can assume two values only.
- A column *"birth_date"*, an integer representing one day of the year
- A column *"name"*, which is a string with length 4 formed by the concatenation of letters at random.

A hundred files containing the columns above and 1000 rows each were generated and placed in a folder. Values for the aforementioned columns were randomly generated following a discrete uniform probability distribution function. Also, some additional payload columns which will not be used in the test were introduced, as to increase the file sizes. Each one of them had around 2MB, so the total dataset size was 210MB. The block size was not modified, meaning that files will be processed at once. The cluster used to perform queries was the same one as in Section 6.2.2.

A set of queries was executed upon the directory. Table 6.2 shows the query predicates and their respective response times after averaging 10 executions. Once again, the proposed implementation was compared with the original importer version, so the processing performance can be compared: the variation is shown in the *Performance variation* column.

Table 6.2: Performance variation compared with execution times for the test queries

Predicate	# of files read	Execution time (in s)	Performance variation
`birth_date = 1`	100	1.862	+2.873%
`name like "a\%"`	100	1.812	+0.165%
`gender = "F"`	100	1.795	-0.187%
`name >= "taaa"`	100	1.805	-0.102%
`birth_date >= 300`	100	1.817	+0.025%

Even though the query predicates select a reduced portion of the dataset, each query results in all files being read. This is due to the nature of file columns involved in the predicate: since they present highly-repetitive (such as the column *gender*) and randomly distributed values (such as the column *name*), the calculated file statistics will have similar minimum-maximum ranges, which will not be able to be leveraged by the importer to prune files from the query execution. For instance, when executing the first query, all files present the value *"1"* in their *birth_date* column. Therefore, the importer will have to read every file to process the query.

In the third query, rows with the value `"F"` are selected. Since the *gender* column can assume one of two values, and given that an uniform probability distribution was used to generate them, the expected amount of data to be read was roughly 100MB - half of the

dataset. However, even with the proposed modifications, the importer reads 210MB of data (all files), since the value `"F"` is repeated in every file in the directory.

Therefore, this test shows that performance improvements obtained with the file statistics are directly impacted by the file data organization. Better results are obtained when values are partitioned by range or by key, and put into separate files. However, compared to the original version, the proposed modification will impact query processing time only when statistics need to be calculated - as observed in the fist query, in which execution time is 2.873% higher. When statistics are already present for the file, no additional calculation is performed, so the reading performance becomes very similar to the original version: this is observed in the variation column for the remaining queries.

6.3 Predicate pushdown and block pruning

Block statistics allow the engine to concentrate on portions of the file which have relevant information for a given query. Therefore, the amount of data read can be reduced, impacting the execution time - specially the predicate is highly selective.

To assess the performance of queries when block statistics are present, a public CSV dataset containing records for taxi rides in New York City, for the month of December 2016 (TLC Trip Record Data, 2020). It consists in a 898MB file having 17 columns which characterize a taxi journey.

Different queries targeting columns *"day"* and *"toll"* were executed. Their value ranges by block are shown in Figure 6.6. During the experiment, block size was set to 100MB, so the file was divided in 9 blocks. As seen in the picture, column *"ride_start_day"* has its values distributed in a more segmented fashion, as ranges are smaller. For instance, only the last 4 blocks have values greater than 20. Column *"toll"*, in the other hand, has a larger minimum and maximum ranges in each block.

Table 6.3: Query predicates and measurements for the test instance.

Predicate	% of rows selected	# blocks read	Speedup
`13 < day < 19`	21.93	5	1.831
`day > 20`	32.48	5	1.765
`day = 15`	3.86	5	1.827
`day = 31`	2.71	1	8.840
`600 < toll < 610`	4.78×10^{-5}	5	1.866
`toll < 0`	0.047	9	1.008
`toll > 910`	1.53×10^{-4}	1	8.748

Table 6.3 shows the predicates of test queries, along with the percentage of rows in the dataset selected, number of file blocks read and the observed speedup, which was calculated by dividing the query time prior the introduced changes by the time obtained after the statistical calculation was introduced. In many occasions, the observed speedup is approximately 1.8, as the predicate results in 4 blocks being pruned by the engine. When comparing queries that obtained such speedup, there's a discrepancy between the selectivity of queries: for instance, predicate *toll < 0* selects only five rows in the dataset, whereas the second query selects approximately 32% of file records. However, they still get roughly the same speedup. That is due to the data layout in the file: although there are few values smaller than zero in the *tolls* column, they are located in different blocks - so,

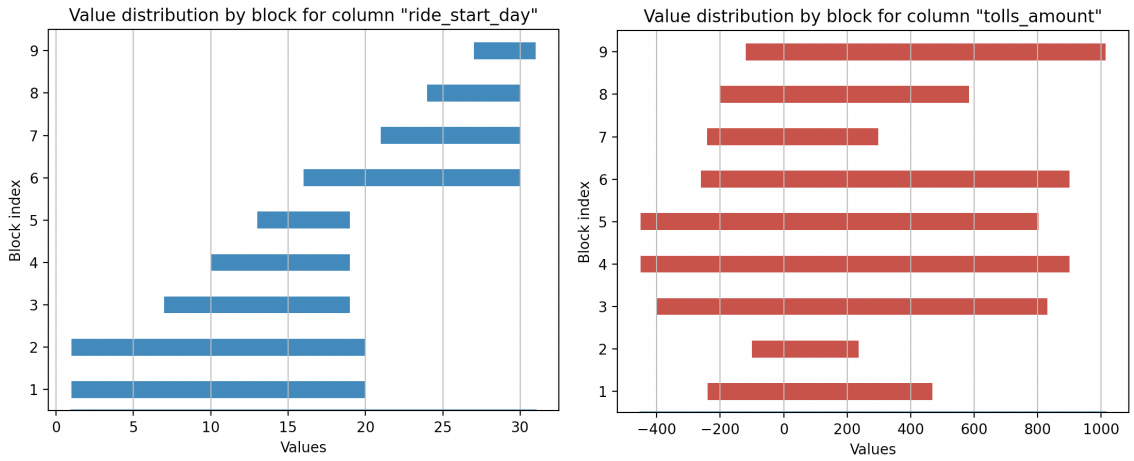


Figure 6.6: By using block statistics, less blocks are read, resulting in a smaller overall execution time.

these will have to be read and processed by the importer.

On the other hand, there are predicates with high selectivity which are able to obtain a considerable speedup: for instance, predicates *day = 31* and *toll > 910* select a reduced portion of the file while obtaining a speedup close to 9. This result is once again explained by the column value distribution in the file: all values *31* in the column *"day"* are located in the last file block, and only block 9 possesses values greater than 900 in the column *"toll"*.

Therefore, these results show that improvements on query execution time are closely related to the file data layout. Datasets that possess a logical column organization - such as ordered columns or well distributed domain ranges - will benefit the most from the proposed changes. However, even for files that do not possess such characteristic, performance improvements can be observed, if the query predicate happens to select values exclusive to a single block statistical range (such as observed in last query in table 6.3).

6.4 CSV & Parquet performances

Considering that CSV is a text format, it does not inherently possess the information needed to apply predicate pushdown. Contrarily, Parquet files have embedded statistics and a columnar data layout, characteristics that can be used to enhance the reading of data.

The block statistical calculation for CSV files aims to improve performance of queries over CSV in analytical contexts. Therefore, this section aims to measure how the CSV format fare against Parquet, a format which is tailored for swift processing and data retrieval..

The test was composed of 4 selective queries upon two directories containing the same datasets, but in different formats: one with CSV files, and the other one with Parquet files. These folders have shipment information for the years of 1992 to 1998, where a given subdirectory encloses data for a single year. Furthermore, for each of those sub-folders, 7 different files were created, using a hash partitioning on the integer column representing the identifier of a shipment order (`ORDERKEY`). In total, 49 files are contained in this dataset, for each format. The CSV folder has 7.8GB in size, and it was generated by a conversion tool based on the Parquet folder, which has 1.5GB.

The first query select records for a specific order key. The second one requests data for the year of 1993, and the third for the years of 1992 or 1998. Finally, the last query selects data for a specific date in 1995. By analyzing the results shown in Figure 6.7, it can be observed that the proposed CSV reading method with block statistics calculation is still far away from the performance presented by Parquet file version. Nevertheless, it outperforms considerably the original implementation, as only the appropriate data will be read in each query. Thus, the proposed modification can lower the performance gap between CSV and Parquet formats in analytical queries, such as the ones observed in the test.

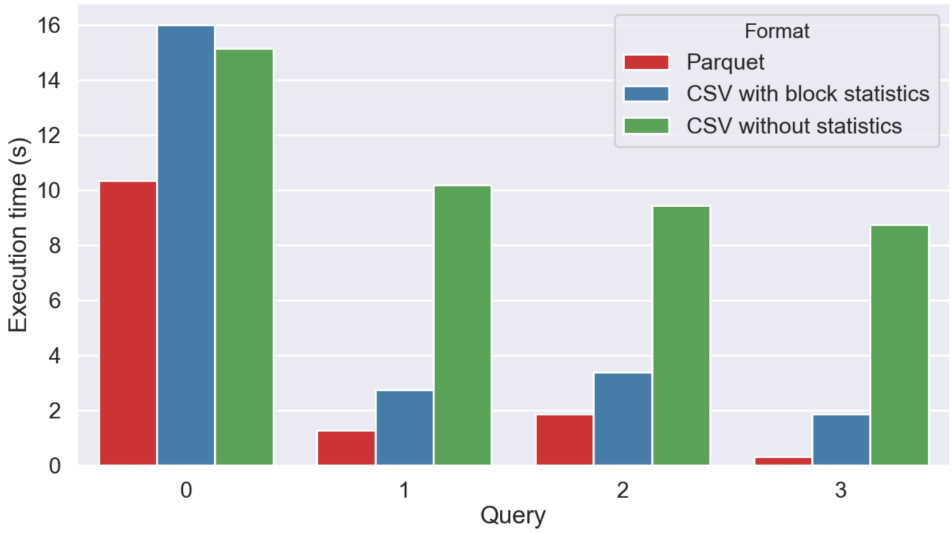


Figure 6.7: Block statistics for CSV make query execution more effective for this format.

6.5 Additional factors which impact performance

In this chapter, performance variations were measured from the relational engine's perspective, considering the ingestion and processing of files made by the importer and query predicates. However, there are some additional factors that may impact the performance of analytical workloads made upon CSV files in data lakes. They are:

a) **Data lake latency**: as file blocks are fetched from an external sources over the network, latency plays an important role in query performance. Different data lake providers present parameters and characteristics which may impact the access to remote files. For instance, users are able to choose the physical location in which their data will be stored - Amazon S3, Google Cloud Storage and Microsoft Azure are all examples of data lake systems which support multiple regions for hosting data. Depending on the user configuration, data access may be impacted by a high latency when downloading blocks, resulting in a decreased query performance.

   Additional configuration options, such as resource quotas, download limits and type of storage media can also play a role in the execution time of an analytical workload, as they may increase the time required to transfer data over the network.

b) **Data mutability**: files can encompass information that are regularly modified, or records can be frequently added. As discussed in Subsection 5.3.2, a certain file

modification will cause the block statistics to be recalculated. In this occasion, the total file processing time will be slightly increased (around 3%), as observed in tests made in subsection 6.1. Therefore, the proposed statistical calculation is suitable for files which are not updated with frequency.

c) **Hot vs. Cold data**: datasets can be categorized based on their access frequency. *Hot Data* is the term employed to categorize collections of information that need to be constantly accessed - for example, a company's online sale catalog, which is often updated. On the other end of the spectrum, there is *cold data*, which are bodies of records that are not frequently accessed. They are suitable for storing historical or legacy datasets, for instance.

When storing files in a data lake, one has to keep in mind that accesses incur a considerable cost, as information needs to be transferred over the network. Therefore, Vora's *SQL on files* tool is suitable for cold data, as queries will result in file blocks being downloaded from the data lake. If files storing hot data are employed with this feature, the response time of the analytical workload will be increased.

d) **Data Quality**: refers to the information correctness and consistency of records observed in a given dataset. An absence of such traits would mean that file data contains tainted information (e.g, ill-formatted characters, empty rows) and incorrect values (e.g., in CSV files, a certain column storing values which represent different data types). In these scenarios, some pre-processing operations are required to clean the data prior it can be used in analytical workloads, extending the overall execution time.

Moreover, wrong values may impact the statistical calculation and processing of CSV files, depending on the user configuration. *SQL on files* allows setting default values for columns, which are used if a given record cannot be processed by the relational engine (e.g., user has declared that a column stores integers, but a value corresponding to a string was found). Block statistics are affected in these situations, as minimum and maximum column ranges can be absent: this would represent statistical intervals of $(-\infty, max\_value]$ and $[min\_value, \infty)$, respectively.

7 CONCLUSION

SAP Vora is an in-memory distributed relational database management system that supports data ingestion from multiple sources. It represents a base for the company's big data solution stack, as it offers an alternative to orchestrate and consolidate heterogeneous types of information.

The project was centered around *SQL on Files*, a functionality that allows querying information stored in files located on data lakes. In analytical workloads, queries are made upon these file sources to extract important information, providing insights that can be either moved to secondary storage (e.g., relational database) or used as input to subsequent processing steps. In this context, binary file formats such as Parquet and ORC largely outperform text formats, such as CSV, because their inner structure allows the application of techniques to enhance file processing. Furthermore, an intrinsic challenge of supporting text files in analytical processing is parsing and converting text into appropriate data types. Nevertheless, CSV is a widespread format and continues to be used in many application areas. Therefore, the project goal was to enhance the performance of analytical queries involving this format.

The proposed solution was based on a technical and architectural analysis of Vora's relational engine and its data importer module to uncover what techniques were used to speed up the execution time of queries targeting "*big data*" file formats. From such investigation, predicate pushdown and file pruning were methods extended for CSV files. Moreover, the technique of block pruning was proposed for the format. These solutions relied on the parallel computation of statistical column minimum and maximum values when reading CSV files, which were persisted in a distributed in-memory cache.

Results show that although a 3% reading time overhead can be observed when processing the CSV file for the first time, the performance of analytical queries targeting the format can be considerably improved with selective predicates. The main advantage of the proposed modifications was that additional costs of fetching, parsing, and processing irrelevant blocks are removed. In addition, tests show that performance can be further enhanced if a logical organization is present in the file or directory storing the targeted dataset. These outcomes conform to the use cases envisioned by the SQL on Files functionality.

When executing analytical workloads in data lakes, not only the file format has to be taken into account. Other factors and configuration details inherently present in data lake systems, such as data locality, mutability, and quality, must be considered because they influence the global performance of such applications.

Future work can be centered around the proposed modifications. In order to minimize the impact of accessing external file sources and parsing text data, raw CSV file blocks can be stored in the distributed cache based on access frequency. Alternatively,

information about block offsets and row group locations can be cached. Block size is a variable that can be experimented with: by lowering the total size of the file block, the importer can produce finer-grained statistics, and possibly increase the pruning of blocks on analytical queries. Also, the solution implementation can be extended to include more statistics, such as column average, sum and value count. These can be useful in queries that select these specific measurements. Bloom filters are data structures that can efficiently determine if a given value is present in a data collection, so they can be used when equality-based query predicates are employed, and avoid having to access a specific file block. Moreover, one aspect that was not considered during the project was the amount of space needed for storing CSV datasets. Future work can be based on compression techniques to lower the total size of files transferred over the network. An important trade-off to be considered in that scenario is the compression and decompression overhead *versus* the download time improvement.

REFERENCES

ALAGIANNIS, I. et al. NoDB: efficient query execution on raw data files. In: New York, NY, USA. **Anais...** Association for Computing Machinery, 2012. p.241–252. (SIGMOD '12).

Amazon Athena. https://aws.amazon.com/athena, acessed on 13/05/2021, Amazon.

Amazon Redshift Spectrum. https://docs.aws.amazon.com/en_us/redshift/latest/dg/c-using-spectrum.html, acessed on 13/05/2021, Amazon.

Apache ORC Specification. https://orc.apache.org//, acessed on 22/05/2021, Apache ORC.

Apache Spark Official Website. https://spark.apache.org/g, acessed on 08/07/2020, Apache Foundation.

CHEN, H.; CHIANG, R. H.; STOREY, V. C. Business intelligence and analytics: from big data to big impact. **MIS quarterly**, [S.l.], p.1165–1188, 2012.

Dremio Query Engine. https://www.dremio.com/, acessed on 13/05/2021, Dremio.

Drought monitoring in Romania. https://www.copernicus.eu/en/use-cases/drought-monitoring-romania, acessed on 08/07/2020, Copernicus.

Facebook Inc. **Facebook code - Scaling the Facebook data warehouse to 300 PB**. 2019.

Fang, H. Managing data lakes in big data era: what's a data lake and why has it became popular in data management ecosystem. In: IEEE INTERNATIONAL CONFERENCE ON CYBER TECHNOLOGY IN AUTOMATION, CONTROL, AND INTELLIGENT SYSTEMS (CYBER), 2015. **Anais...** [S.l.: s.n.], 2015. p.820–824.

Farm Sustainability Tool (FaST). https://www.copernicus.eu/en/use-cases/farm-sustainability-tool-fast-space-data-sustainable-farming, acessed on 08/07/2020, Copernicus.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, [S.l.], v.C-21, n.9, p.948–960, 1972.

Garber, L. Using In-Memory Analytics to Quickly Crunch Big Data. **Computer**, [S.l.], v.45, n.10, p.16–18, 2012.

GE, C. et al. Speculative Distributed CSV Data Parsing for Big Data Analytics. In: 2015 . **Anais...** Association for Computing Machinery, 2019. p.883–899. (SIGMOD '19).

IDREOS, S. et al. Here are my data files. here are my queries. where are my results? In: BIENNIAL CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH, 5. **Proceedings...** [S.l.: s.n.], 2011. n.CONF.

IVANOV, T.; PERGOLESI, M. The impact of columnar file formats on SQL-on-hadoop engine performance: a study on orc and parquet. **Concurrency and Computation: Practice and Experience**, [S.l.], v.32, 09 2019.

JAIN, A.; DOAN, A.; GRAVANO, L. Optimizing SQL Queries over Text Databases. In: IEEE 24TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 2008. **Anais...** [S.l.: s.n.], 2008. p.636–645.

KAMBATLA, K. et al. Trends in big data analytics. **Journal of Parallel and Distributed Computing**, [S.l.], v.74, n.7, p.2561–2573, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

KARPATHIOTAKIS, M.; ALAGIANNIS, I.; AILAMAKI, A. Fast Queries over Heterogeneous Data through Engine Customization. **Proc. VLDB Endow.**, [S.l.], v.9, n.12, p.972–983, Aug. 2016.

LANEY, D. 3D data management: controlling data volume, velocity and variety. **META group research note**, [S.l.], v.6, n.70, p.1, 2001.

LAVALLE, S. et al. Big data, analytics and the path from insights to value. **MIT sloan management review**, [S.l.], v.52, n.2, p.21–32, 2011.

LEE, I. Big data: dimensions, evolution, impacts, and challenges. **Business Horizons**, [S.l.], v.60, n.3, p.293–303, 2017.

MITLöHNER, J. et al. Characteristics of Open Data CSV Files. In: INTERNATIONAL CONFERENCE ON OPEN AND BIG DATA (OBD), 2016. **Anais...** [S.l.: s.n.], 2016. p.72–79.

ORC Adopters. https://orc.apache.org/docs/adopters.html, acessed on 21/03/2019, Apache Software Foundation.

PALKAR, S. et al. Filter before You Parse: faster analytics on raw data with sparser. **Proc. VLDB Endow.**, [S.l.], v.11, n.11, p.1576–1589, July 2018.

Parquet Adopters. https://parquet.apache.org/adopters, acessed on 21/03/2019, Apache Software Foundation.

PENG, S.; WANG, G.; XIE, D. Social influence analysis in social networking big data: opportunities and challenges. **IEEE network**, [S.l.], v.31, n.1, p.11–17, 2016.

SAP Vora. https://help.sap.com/viewer/0991e2320f5940d988ed32b995d28a44/2.1/en-US, acessed on 13/05/2021, SAP Vora.

TAPSAI, C. Information Processing and Retrieval from CSV File by Natural Language. In: IEEE 3RD INTERNATIONAL CONFERENCE ON COMMUNICATION AND INFORMATION SYSTEMS (ICCIS), 2018. **Anais...** [S.l.: s.n.], 2018. p.212–216.

Task-Based Programming, Intel TBB Documentation. https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Task-Based_Programming.html, acessed on 01/04/2021, Task-Based Programming.

TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, acessed on 08/07/2020, NYC Taxi and Limousine Commission.

VAN RENESSE, R.; SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In: OSDI. **Anais...** [S.l.: s.n.], 2004. v.4, n.91–104.

WATSON, H. J. All about analytics. **International Journal of Business Intelligence Research (IJBIR)**, [S.l.], v.4, n.1, p.13–28, 2013.

ZAHARIA, M. et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: PRESENTED AS PART OF THE 9TH {USENIX} SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION ({NSDI} 12). **Anais...** [S.l.: s.n.], 2012. p.15–28.