





Storage Format Selection and Optimization for Materialized Intermediate Results in Data-Intensive Flows

Rana Faisal Munir

UNIVERSITAT POLITÈCNICA DE CATALUNYA
Department of Service and Information System Engineering
Barcelona, 2019







Storage Format Selection and Optimization for Materialized Intermediate Results in Data-Intensive Flows

Rana Faisal Munir

supervised by
Prof. Alberto Abelló
Prof. Oscar Romero
Prof. Wolfgang Lehner

Thesis submitted for the degree of Doctor of Philosophy at Universitat Politècnica de Catalunya and Technische Universität Dresden, in partial fulfillment of the requirements within the scope of the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College".

December, 2019

Storage Format Selection and Optimization for Materialized Intermediate Results in Data-Intensive Flows. December 2019.

Rana Faisal Munir
fmunir@essi.upc.edu

Database Technologies and Information Management Group
Universitat Politècnica de Catalunya
Jordi Girona, 1-3
08034 - Barcelona, Spain

UPC Main Ph.D. Supervisors:	Prof. Alberto Abelló
	Prof. Oscar Romero
	Universitat Politècnica de Catalunya, BarcelonaTech, Spain
TUD Ph.D. Supervisor:	Prof. Wolfgang Lehner
	Technische Universität Dresden, Germany
Ph.D. Committee:	Prof. Torben Bach Pedersen
	Aalborg University, Denmark
	Prof. Antonio Corral
	Universidad de Almería, Spain
	Prof. Norbert Ritter
	Universität Hamburg, Germany
Ph.D. Series:	Barcelona School of Informatics, Universitat Politècnica de Catalunya, BarcelonaTech

This dissertation is available on-line at the Thesis and Dissertations On-line (TDX) repository, which is coordinated by the Consortium of Academic Libraries of Catalonia (CBUC) and the Supercomputing Centre of Catalonia Consortium (CESCA), by the Catalan Ministry of Universities, Research and the Information Society.

© Copyright by Rana Faisal Munir. The author has obtained the right to include the published and accepted articles in the thesis, with a condition that they are cited, DOI pointers and/or copyright/credits are placed prominently in the references.

Acknowledgments

PhD is a long journey with many challenges, which is not possible to finish without the support of other people. I have always been blessed to be surrounded with supportive people, who were always there to support me in every aspect of my life. Specifically, they helped me a lot during the entire duration of my PhD. I would like to express my sincere gratitude to all of them for their trust and encouragement.

First of all, I wish to express my sincerest gratitude to my supervisors. Their support on each and every step made it possible to finish my PhD. Specifically, I would like to thank Dr. Alberto Abelló, Dr. Oscar Romero, Dr. Wolfgang Lehner and Dr. Maik Thiele for being always there to help and encourage.

I am also really grateful to Besim Bilalli and Mennan Selimi, for being always there in the difficult times of my PhD. Their advises and support made it possible to reach the finish line. I would also like to thank the whole DTIM group (past and present members) for the great memories.

I would like to thank my parents and sisters for their support during the entire PhD period. I would like to thank my beloved wife Atika and daughter Hiba, who always allowed me to spend extra time on PhD.

Finally, let me close with an Arabic word that expresses in a perfect way all the possible thanks to the Almighty Creator, Alhamdulillah.

Rana Faisal Munir
December, 2019
Barcelona, Spain

* * *

This work has been funded by the European Commission (EACEA) through the Erasmus Mundus doctoral fellowship, via the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College (IT4BI-DC)".

Abstract

Modern organizations produce and collect large volumes of data, that need to be processed repeatedly and quickly for gaining business insights. For such processing, typically, *Data-intensive Flows (DIFs)* are deployed on distributed processing frameworks. The DIFs of different users have many computation overlaps (i.e., parts of the processing are duplicated), thus wasting computational resources and increasing the overall cost. The output of these computation overlaps (known as intermediate results) can be materialized for reuse, which helps in reducing the cost and saves computational resources if properly done. Furthermore, the way such outputs are materialized must be considered, as different storage layouts (i.e., horizontal, vertical, and hybrid) can be used to reduce the I/O cost.

In this PhD work, we first propose a novel approach for automatically materializing the intermediate results of DIFs through a multi-objective optimization method, which can tackle multiple and conflicting quality metrics. Next, we study the behavior of different operators of DIFs that are the first to process the loaded materialized results. Based on this study, we devise a rule-based approach, that decides the storage layout for materialized results based on the subsequent operation types. Despite improving the cost in general, the heuristic rules do not consider the amount of data read while making the choice, which could lead to a wrong decision. Thus, we design a cost model that is capable of finding the right storage layout for every scenario. The cost model uses data and workload characteristics to estimate the I/O cost of a materialized intermediate results with different storage layouts and chooses the one which has minimum cost. The results show that storage layouts help to reduce the loading time of materialized results and overall, they improve the performance of DIFs.

The thesis also focuses on the optimization of the configurable parameters of hybrid layouts. We propose ATUN-HL (Auto TUNing Hybrid Layouts),

which based on the same cost model and given the workload and characteristics of data, finds the optimal values for configurable parameters in hybrid layouts (i.e., Parquet).

Finally, the thesis also studies the impact of parallelism in DIFs and hybrid layouts. Our proposed cost model helps to devise an approach for fine-tuning the parallelism by deciding the number of tasks and machines to process the data. Thus, the cost model proposed in this thesis, enables in choosing the best possible storage layout for materialized intermediate results, tuning the configurable parameters of hybrid layouts, and estimating the number of tasks and machines for the execution of DIFs.

Keywords

data-intensive flows; storage layouts; big data; parallelism; data management

Resumen

Las organizaciones producen y recopilan grandes volúmenes de datos, que deben procesarse de forma repetitiva y rápida para obtener información relevante para la empresa. Para tal procesamiento, por lo general, se emplean flujos intensivos de datos (DIFs por sussiglas en inglés) en entornos de procesamiento distribuido. Los DIFs de diferentes usuarios tienen elementos comunes (es decir, se duplican partes del procesamiento, lo que desperdicia recursos computacionales y aumenta el coste en general). Los resultados intermedios de varios DIFs pueden pues coincidir y se pueden por tanto materializar para facilitar su reutilización, lo que ayuda a reducir el coste y ahorrar recursos si se realiza correctamente. Además, la forma en qué se materializan dichos resultados debe ser considerada. Por ejemplo, diferentes tipos de diseño lógico de los datos (es decir, horizontal, vertical o híbrido) se pueden utilizar para reducir el coste de E/S.

En esta tesis doctoral, primero proponemos un enfoque novedoso para materializar automáticamente los resultados intermedios de los DIFs a través de un método de optimización multi-objetivo, que puede considerar múltiples y contradictorias métricas de calidad. A continuación, estudiamos el comportamiento de diferentes operadores de DIF que acceden directamente a los resultados materializados. Sobre la base de este estudio, ideamos un enfoque basado en reglas, que decide el diseño del almacenamiento para los resultados materializados en función de los tipos de operaciones que los utilizan directamente. A pesar de mejorar el coste en general, las reglas heurísticas no consideran estadísticas sobre la cantidad de datos leídos al hacer la elección, lo que podría llevar a una decisión errónea. Consecuentemente, diseñamos un modelo de costos que es capaz de encontrar el diseño de almacenamiento adecuado para cada escenario dependiendo de las características de los datos almacenados. El modelo de costes usa estadísticas y características de acceso para estimar el coste de E/S de un resultado inter-

medio materializado con diferentes diseños de almacenamiento y elige el de menor coste. Los resultados muestran que los diseños de almacenamiento ayudan a reducir el tiempo de carga de los resultados materializados y, en general, mejoran el rendimiento de los DIF.

La tesis también presta atención a la optimización de los parámetros configurables de diseños híbridos. Proponemos así ATUN-HL (Auto TUNing Hybrid Layouts), que, basado en el mismo modelo de costes, las características de los datos y el tipo de acceso que se está haciendo, encuentra los valores óptimos para los parámetros de configuración en disponibles Parquet (una implementación de diseños híbridos para Hadoop Distributed File System).

Finalmente, esta tesis estudia el impacto del paralelismo en DIF y diseños híbridos. El modelo de coste propuesto ayuda a idear un enfoque para ajustar el paralelismo al decidir la cantidad de tareas y máquinas para procesar los datos. En resumen, el modelo de costes propuesto permite elegir el mejor diseño de almacenamiento posible para los resultados intermedios materializados, ajustar los parámetros configurables de diseños híbridos y estimar el número de tareas y máquinas para la ejecución de DIF.

Palabras Clave

flujos intensivos de datos; diseños de almacenamiento; grandes datos; paralelismo; gestión de datos

Abstrakt

Moderne Unternehmen produzieren und sammeln große Datenmengen, die wiederholt und schnell verarbeitet werden müssen, um geschäftliche Erkenntnisse zu gewinnen. Für die Verarbeitung dieser Daten werden typischerweise Datenintensive Prozesse (DIFs) auf verteilten Systemen wie z.B. MapReduce bereitgestellt. Dabei ist festzustellen, dass die DIFs verschiedener Nutzer sich in großen Teilen überschneiden, wodurch viel Arbeit mehrfach geleistet, Ressourcen verschwendet und damit die Gesamtkosten erhöht werden. Um diesen Effekt entgegenzuwirken, können die Zwischenergebnisse der DIFs für spätere Wiederverwendungen materialisiert werden. Hierbei müssen vor allem die unterschiedlichen Speicherlayouts (horizontal, vertikal und hybrid) berücksichtigt werden.

In dieser Doktorarbeit wird ein neuartiger Ansatz zur automatischen Materialisierung der Zwischenergebnisse von DIFs durch eine mehrkriterielle Optimierungsmethode vorgeschlagen, der in der Lage ist widersprüchliche Qualitätsmetriken zu behandeln. Des Weiteren wird untersucht die Wechselwirkung zwischen verschiedenen Operatortypen und unterschiedlichen Speicherlayouts untersucht. Basierend auf dieser Untersuchung wird ein regelbasierter Ansatz vorgeschlagen, der das Speicherlayout für materialisierte Ergebnisse, basierend auf den nachfolgenden Operationstypen, festlegt. Obwohl sich die Gesamtkosten für die Ausführung der DIFs im Allgemeinen verbessern, ist der heuristische Ansatz nicht in der Lage die gelesene Datenmenge bei der Auswahl des Speicherlayouts zu berücksichtigen. Dies kann in einigen Fällen zu falschen Entscheidung führen. Aus diesem Grund wird ein Kostenmodell entwickelt, mit dem für jedes Szenario das richtige Speicherlayout gefunden werden kann. Das Kostenmodell schätzt anhand von Daten und Auslastungsmerkmalen die E/A-Kosten eines materialisierten Zwischenergebnisses mit unterschiedlichen Speicherlayouts und wählt das kostenminimale aus. Die Ergebnisse zeigen, dass Speicherlayouts

die Ladezeit materialisierter Ergebnisse verkürzen und insgesamt die Leistung von DIFs verbessern.

Die Arbeit befasst sich auch mit der Optimierung der konfigurierbaren Parameter von hybriden Layouts. Konkret wird der sogenannte ATUN-HL-Ansatz (Auto TUNing Hybrid Layouts) entwickelt, der auf der Grundlage des gleichen Kostenmodells und unter Berücksichtigung der Auslastung und der Merkmale der Daten die optimalen Werte für konfigurierbare Parameter in Parquet, d.h. eine Implementierung von hybrider Layouts.

Schließlich werden in dieser Arbeit auch die Auswirkungen von Parallelität in DIFs und hybriden Layouts untersucht. Dazu wird ein Ansatz entwickelt, der in der Lage ist die Anzahl der Aufgaben und dafür notwendigen Maschinen automatisch zu bestimmen. Zusammengefasst lässt sich festhalten, dass das in dieser Arbeit vorgeschlagene Kostenmodell es ermöglicht, das bestmögliche Speicherlayout für materialisierte Zwischenergebnisse zu ermitteln, die konfigurierbaren Parameter hybrider Layouts festzulegen und die Anzahl der Aufgaben und Maschinen für die Ausführung von DIFs zu schätzen.

Schlüsselwörter

datenintensive Flüsse; Speicherlayouts; Große Daten; Parallelität; Datenmanagement

Contents

Abstract **v**

1 Introduction **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Research Problems and Challenges . . . 5
- 1.3 Running Example . . . 6
- 1.4 Contributions . . . 9
- 1.5 Thesis Overview . . . 10
  - 1.5.1 Chapter 3: Intermediate Results Materialization Selection 11
  - 1.5.2 Chapter 4: Storage Format Selection for Materialized Intermediate Results . . . 12
  - 1.5.3 Chapter 5: Auto Tuning of Hybrid Layouts Using Workload and Data Characteristics . . . 12
  - 1.5.4 Chapter 6: Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization . . . 13

2 Related Work **15**
- 2.1 Intermediate Results Materialization . . . 15
  - 2.1.1 Relational Databases . . . 15
  - 2.1.2 Distributed Processing Frameworks . . . 16
- 2.2 Storage Layouts for Materializing Intermediate Results . . . 18
- 2.3 Configuring Parameters of Hybrid Layouts . . . 21
- 2.4 Configuring Parallelism for Hybrid Layout . . . 22

3 Intermediate Results Materialization Selection **25**
- 3.1 Introduction . . . 26
  - 3.1.1 Motivational Example . . . 27
- 3.2 Formal Building Blocks and Problem Statement . . . 30

3.2.1 Multiquery AND/OR DAGs and Data-Intensive Flows . 30
3.2.2 Components 31
3.2.3 Problem Statement 32
3.3 Cost Model 32
3.3.1 Data-Intensive Flow Statistics 33
3.3.2 Metrics 34
3.3.3 Cost Functions 35
3.4 State Space Search Algorithm 38
3.4.1 Actions 38
3.4.2 Initial State 40
3.4.3 Heuristic 41
3.4.4 Searching The Solution Space 42
3.5 Experiments 42
3.5.1 Intermediate Results Selection Evaluation 43
3.6 Conclusions 49

4 Storage Format Selection for Materialized Intermediate Results **51**
4.1 Introduction 52
4.2 Background and Motivation 54
4.2.1 Storage layouts 54
4.2.2 Layout performance comparison 56
4.3 Our Approach in a Nutshell 57
4.3.1 Storage layout selection 57
4.4 Heuristic Rules 58
4.4.1 Comparison of Data Formats 58
4.4.2 Selecting the Appropriate Format 59
4.5 Cost-Based Model 60
4.5.1 Write cost 62
4.5.2 Read cost 64
4.6 Instantiating the Cost Model 69
4.6.1 SequenceFile (SeqFile) format 69
4.6.2 Avro format 72
4.6.3 Parquet format 73
4.7 Experiments 74
4.7.1 Experimental setup 76
4.7.2 Validation of file size estimations 76
4.7.3 Validation of file format choice 78

4.8 Conclusion 84

5 Auto Tuning of Hybrid Layouts Using Workload and Data Characteristics **85**
5.1 Introduction 86
5.2 Cost Model 88
5.2.1 Estimating the selection cost 89
5.2.2 Estimating the size of the dictionary 92
5.3 ATUN-HL 93
5.3.1 Collecting workload and data characteristics 94
5.3.2 Finding the best configuration parameters 94
5.4 Experimental Results 96
5.4.1 Setup 96
5.4.2 Dataset 97
5.4.3 Results 97
5.5 Conclusions 101

6 Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization **103**
6.1 Introduction 104
6.2 Cost Model for Hybrid Layouts 106
6.2.1 Parameters of the Cost Model 106
6.2.2 Physical Format of Hybrid Layouts 107
6.2.3 Estimating Number of Tasks 107
6.2.4 Estimating MakeSpan 107
6.3 Our Approach 112
6.3.1 Query Parser 113
6.3.2 Query Profiling 113
6.3.3 Data Profiling 113
6.3.4 Cost Model 114
6.4 Multi-Objective Optimization 114
6.5 Experimental Results 116
6.5.1 Setup 116
6.5.2 Results 117
6.6 Conclusions 121

7 Conclusions and Future Directions **123**
7.1 Conclusions 123
7.2 Future Directions 125

Bibliography **127**

1

Introduction

1.1 Background and Motivation

We are living in an era, where data is a valuable asset. Its size is exponentially growing from petabytes to zettabytes [71]. For instance, data globally generated in two days is larger than what we have generated from the dawn of civilization up until 2003[1]. Such large volume of data cannot be handled in a single machine, due to the limitation in computing power. Thus, the availability of data has imposed a shift in the hardware, from single machines to large scale computer clusters. Researchers have proposed many distributed processing systems (such as Hadoop[2], Spark[3], etc.) to facilitate storing and processing of such large volume of data.

The Hadoop ecosystem is a pioneer large-scale distributed system, that consists of a storage layer namely, Hadoop Distributed File System (HDFS)[4] and a processing layer namely, MapReduce [18]. The former allows to keep data in raw format without any normalization or pre-processing. The latter allows parallel processing of the data. Hadoop follows a master-slave

[1]https://techcrunch.com/2010/08/04/schmidt-data
[2]https://hadoop.apache.org
[3]https://spark.apache.org
[4]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

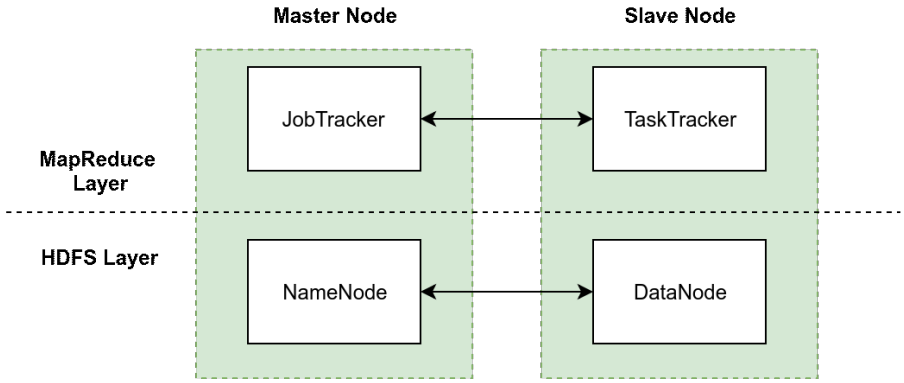


Fig. 1.1: High-level architecture of Hadoop

architecture, where one node is a master and all the others act as slaves. HDFS master node (also known as NameNode) is responsible for storing meta-data about data stored on each slave machine (also known as DataNode). Whereas, MapReduce has a master node (known as JobTracker), which is responsible of executing jobs on different slave machines (known as TaskTracker). Figure 1.1 shows the high-level architecture of a Hadoop cluster.

Many enterprises already have their own Hadoop clusters and motivate their employees to utilize the cluster for their analysis. However, it is difficult to write an analytical job in pure MapReduce programming model. Researchers have proposed many high-level languages (Pig [5], Hive [6], Drill [7], etc.) to facilitate writing analytical jobs. These languages hide the complexity from end-user and facilitate exploring the data on ad-hoc basis. They translate the code into a Directed Acyclic Graph (DAG) of multiple MapReduce jobs, which is also known as Data-Intensive Flows (DIFs). Each node of a DAG takes an input data and produces an output after performing certain processing. Figure 1.2a shows an example of two DIFs.

These DIFs process a large volume of data and take a lot of time to produce the desired outputs. The competition in businesses also demands quick business-insights, which make it always desirable to optimize the execution of DIFs. The goal of optimization is to reduce the execution time and also save computational resources, which indirectly save energy and money. This thesis focuses specifically on optimizing the execution of DIFs.

[5] https://pig.apache.org
[6] https://hive.apache.org
[7] https://drill.apache.org

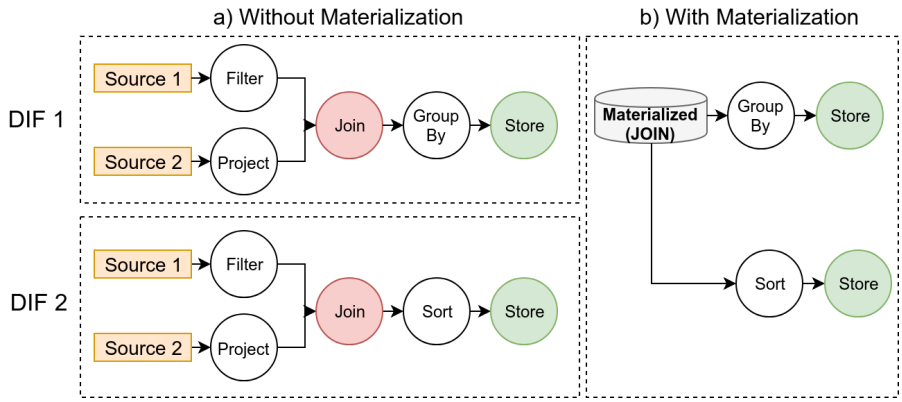


Fig. 1.2: Benefits of Materialization

As mentioned previously, modern organizations have their own Hadoop cluster, which is shared and used by multiple users of the same organization. They deploy their DIFs on the cluster to store and process the data. They work for the same objectives and thus, their DIFs share many common/redundant parts (also known as Intermediate Results - IRs), which is already acknowledged in an in-depth study of seven enterprises [14]. This study showed that 80% of DIFs had redundant/common parts. Similarly, recent studies [43, 44] from Microsoft have shown that 65% of their DIFs have redundant parts.

These studies highlight the opportunity of optimizing the DIFs' execution. They imply that a proper management of IRs could provide benefits in terms of computational resources and execution time. For instance, if the IRs are materialized then they can be used in future executions without recomputing them as shown in Figure 1.2. This would help to improve the execution time and save computational resources.

The execution time of DIFs can be further reduced by choosing the storage layouts for the chosen materialized IRs. Typically, the materialized IRs are stored on HDFS, where I/O operations are expensive [10]. Hence, unnecessary reads and writes performed, increase the execution cost. Researchers have come up with different storage layouts that help in reducing the amount of read and write operations. These layouts are built on top of HDFS and are designed for fast loading, fast query processing and efficient storage utiliza-

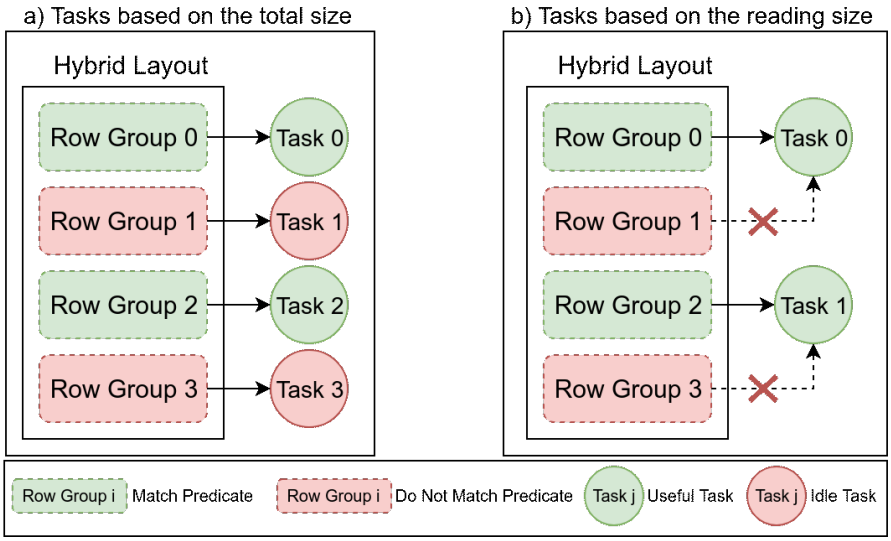


Fig. 1.3: Degree of Parallelism

tion. They are categorized into horizontal layouts (i.e., SequenceFile[8], Avro[9], etc.), vertical layouts (i.e., Zebra[10], etc.), and hybrid layouts (i.e., Parquet[11], ORC[12], etc.). None of them is the universal best choice; different workloads require different layouts to achieve optimal performance [5]. Indeed, the I/O operations of loading materialized IRs can be reduced by using different storage layouts for materialization based on the workload [59].

Additionally, researchers have proposed new methods to further improve the execution of DIFs by storing data as a very wide table without applying any normalization [10, 54] and using hybrid layouts to store it, due to their built-in support for many basic operations (i.e., selectivity, projection, aggregation, etc.) allowing direct ad-hoc analysis, without the need of moving the data to other storage engines (i.e., relational, document store, etc.). Hybrid layouts have many configuration parameters, which need to be configured according to the running workload. Otherwise, their default values can significantly impact the execution of DIFs.

As mentioned previously, hybrid layouts allow to read less data from the disk for certain operations, which is not thoroughly exploited by distributed

[8]https://wiki.apache.org/hadoop/SequenceFile
[9]https://avro.apache.org
[10]https://wiki.apache.org/pig/zebra
[11]https://parquet.apache.org
[12]https://orc.apache.org

frameworks when deciding the degree of parallelism (which is also the number of tasks). Indeed, they always decide the number of tasks based on the total table size and not on the portion of the table being read, which leads to the over-provisioning of tasks and degrade the performance of individual DIFs, because many tasks remain idle — without any data to process, but still present extra overhead (e.g., initialization time, garbage collection) as shown in Figure 1.3a. In an ideal scenario, it should create the number of tasks based on the amount of data read from the disk as shown in Figure 1.3b, which would help to reduce the execution time and also save the computational resources.

1.2 Research Problems and Challenges

Data analysis is performed by deploying DIFs on a distributed cluster to process the stored data and getting useful information for the business. The competition in markets demands data to be processed quickly and efficiently. Quick analysis can be done by reducing the execution time of DIFs and efficiency can be improved by reducing the resource usage, which indirectly leads to save money. This thesis focuses on providing solutions for both.

Nowadays, organizations have a central distributed system for analysis installed within premises or on a public cloud. This cluster is shared and used by multiple users of the same organizations simultaneously. These users work for the same business objectives, hence they perform many redundant tasks, which can be avoided by materializing their outputs (also known as IRs). Thus, the materialization is stored on the disk, where I/O operations are expensive. These I/O operations can be reduced by storing the materialized IRs using the best possible storage layout based on the running workload.

The first research problem is related to the selection of best IRs under the given Service Level Agreements (SLAs). There have been already many solutions proposed for choosing the IRs for materialization, but they do not consider multiple conflicting SLAs while choosing IRs for materialization. Our goal is to propose a generic approach, which can take any SLA that is quantifiable (i.e., it has associated metrics to compute its cost) and utilize the given SLA's metrics in a multi-objective approach for selecting the best IRs, that help in improving the execution of DIFs.

The second research problem focuses on storing the chosen IRs in the best possible way, so that their loading time can be reduced. There are many

available storage layouts, that are best suited for different workloads. It is very challenging to choose a storage layout for a selected IR, which can improve its loading time because different storage layouts are good for different types of operations. Thus, it is very important to analyze the type of operation, which is going to read IRs and based on that decide the best option. Our objective is to propose heuristic-based rules and statistical-based cost model for choosing the storage layout. The heuristic-based rules help for cold-start when there is no statistical information are available. Whereas, we record the statistical information during the first execution and utilize them in our cost model for future executions.

Nowadays, storage is very cheap but time is always a constraint for getting business insights. Thus, researchers have proposed denormalized tables (aka wide tables) to store the data. Wide tables allow to analyze data without any need of using JOIN operations, which is expensive in distributed systems. Typically, these very wide tables are stored using hybrid layouts that support projection and selection operations. This support helps to read less data from the disk and improve the overall execution. However, hybrid layouts have parameters that need to be configured according the running workload. Otherwise, it can degrade the execution of DIFs.

The third research problem focuses on tuning the configuration parameters for hybrid layouts. Thus, this thesis proposes a cost-based approach for finding the best possible values according to the current workload.

As mentioned earlier, hybrid layouts help to read less data for certain operations. However, distributed systems do not consider this important factor when deciding the number of tasks to process the data. This leads to over-provision of tasks, that impact the execution of DIFs and waste the computational resources.

Thus, **the fourth research problem** focuses on choosing the number of tasks and number of machines for the efficient execution of a given DIF. The thesis utilizes the proposed cost model to decide the number of tasks and computational resources.

1.3 Running Example

In this section, we present a running example to show the complete end-to-end pipeline based on the above mentioned research questions. Let us assume that there are two users (user A and user B), who are running their

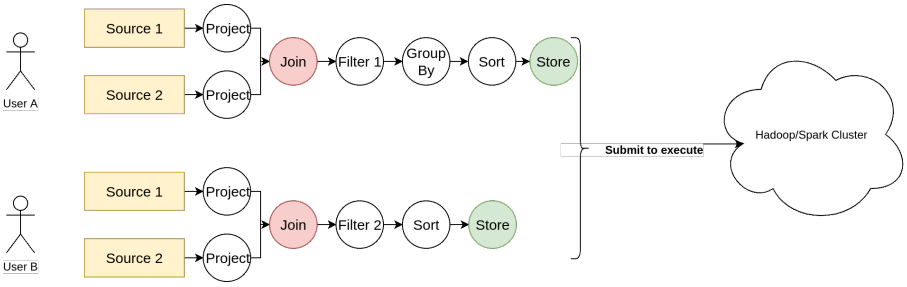


Fig. 1.4: Two DIFs with common parts

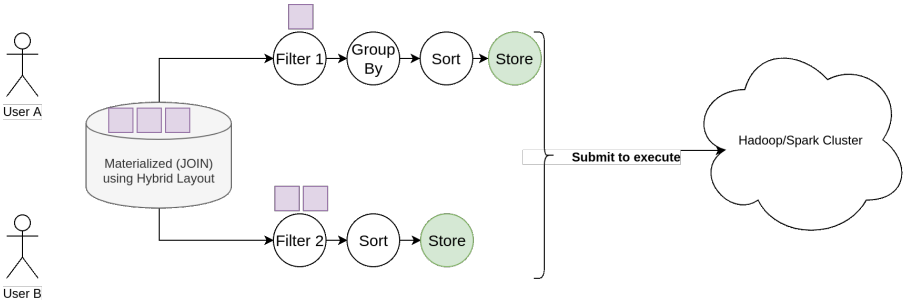


Fig. 1.5: JOIN materialized node with its selected storage format

DIFs on a shared distributed cluster as shown in Figure 1.4. As it can be seen in the figure, the DIFs of both users have a common node (i.e., join), which can be materialized for re-use. The problem of identifying the right node to materialize relates to our first research question.

After choosing JOIN for materialization, we would need to store it on the disk to re-use in future executions. There are many storage layouts available, however each one of them is good for specific types of workloads. In our second research question, we focus on this problem and we propose a solution that helps in deciding the right storage layout according to the access pattern of the chosen materialized node. The proposed solution analyzes the first operations, which are going to read the data from the materialized node. In the above scenario, the first operation in both DIFs is filter and thus, our approach chooses hybrid layout (as shown in Figure 1.5) due to its built-in support for filters. It can be seen in Figure 1.5, that hybrid layouts help these DIFs to read less blocks (i.e., 1 out of 3 for user A and 2 out of 3 for user B) from the disk by using predicate push-down.

Let us assume that the default block size of the hybrid layout is 128MB.

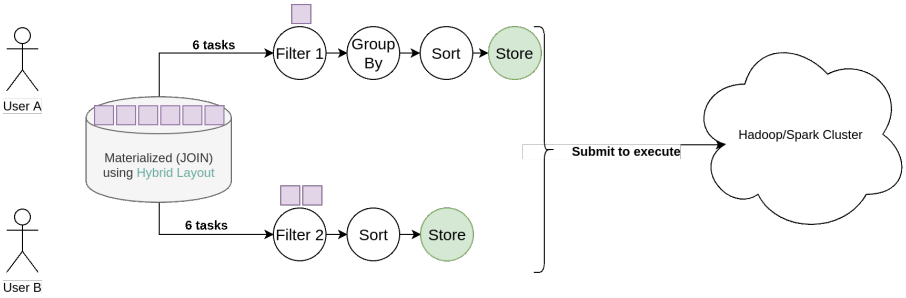


Fig. 1.6: JOIN materialized node with its selected configured storage format

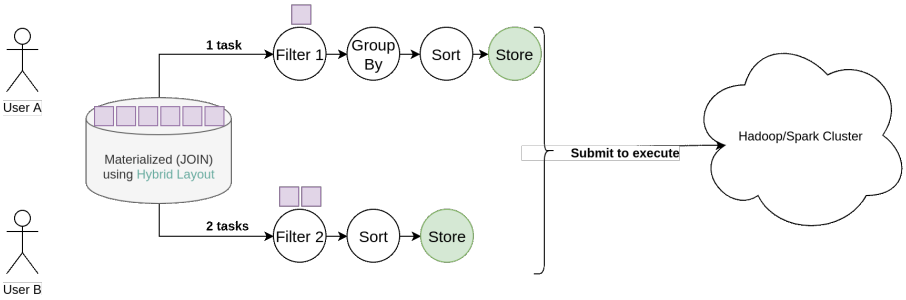


Fig. 1.7: JOIN materialized node with its selected configured storage format and number of optimal tasks

It means that the DIF of user A is reading 128MB from the disk (1 block), whereas the DIF of user B is reading 256MB (2 blocks). The amount of data read can be reduced by configuring the default block size of hybrid layouts. This problem relates to our third research question. Let us suppose that our proposed approach configures 64MB for the block size and now, we have in total 6 blocks rather than 3 as shown in Figure 1.6. The number of blocks read is the same but the total amount of data read has been reduced (64MB for user A and 128MB for user B). This reduces 50% of the I/O operations and helps in improving the execution time of each DIF.

It should be noted that the distributed processing frameworks always create a task for each block on the disk, rather than based on the actual number of blocks read. Thus, it always creates 6 tasks (if we execute one task per machine, then we need 6 machines to process these tasks) for each DIF, however both DIFs are reading fewer blocks. It means many tasks are going to be idle without processing any data and also wasting computational resources.

Our fourth research question focuses on configuring the number of tasks and machines based on the real number of blocks read. As shown in Fig-

ure 1.7, it configures 1 tasks for the DIF of user A and 2 tasks for the DIF of user B, based on their number of blocks read. If we want to execute all of these tasks in parallel, it means we just need 1 machine for the DIF of user A and 2 machines for the DIF of user B, which would help to improve the execution of both DIFs and also save computational resources.

1.4 Contributions

The first research problem focuses on selecting materialized IRs for the given SLAs. We proposed a generic framework that can take any type of quantifiable SLA. These SLAs can be conflicting and can serve different objectives. We utilized an existing multi-objective optimization approach (i.e., local minima with hill-climbing) to consider all the given SLA's metrics and according to them, selecting the best IRs for materialization. We prototyped our approach for four example SLA's metrics (i.e., loading time, query time, storage space, and freshness). The loading time and freshness were considered due to their conflicting nature. Our proposed framework is also compared and tested against the state of the art solutions to confirm its usefulness.

The second research problem focuses on deciding the storage layout for chosen IRs. We did extensive experimental study and found that only the first operation impacts the reading directly from the disk. Thus, deciding the storage layout based on the first operation can help in deciding the best possible storage layout. Based on this evaluation, first we propose heuristic-based rules to choose the best storage layout. These rules are derived based on the features provided by different storage layouts and their implementations. The improvements that we got through our heuristic-based rules motivated us to go for statistical-based cost model to do the same job. The cost model provides better accuracy compared to the rule based approach. However, the cost model requires statistical information, which is not available in the first execution of DIF. To handle this scenario, we proposed a hybrid approach that utilizes both heuristic-based rule for cold-start and statistical-based cost model for future executions. We also proposed an approach to record statistical information during the execution of DIFs.

The third research problem focuses on configuring the parameters of hybrid layouts according to the running workload. We already proposed the cost model for the second research problem, that has a cost model for hybrid layout to estimate its read and write cost. We have used the same cost model

in estimating the reading cost in hybrid layouts for current running workload, by putting different values for all configurable parameters. This cost model helped us to find the best possible values, which improve the overall execution of the running workload.

Finally, the fourth research problem focuses on configuring the number of tasks and number of machines for executing a given query. We have extended the same cost model for hybrid layouts to decide the number of tasks and machines. The extended cost model considered the reading size and used it further in estimating the tasks and machines, that are optimal to process a given DIF. Our detailed experimental evaluation showed significant improvements.

1.5 Thesis Overview

The thesis focuses on solving four research problems. The solution of each problem is presented in a separate chapter. The main body of this thesis is also presented in the following publications:

P1. Rana Faisal Munir, Oscar Romero, Alberto Abelló, Besim Bilalli, Maik Thiele, and Wolfgang Lehner. ResilientStore: A Heuristic-Based Data Format Selector for Intermediate Results[13] In: *International Conference on Model and Data Engineering, (MEDI 2016)*. pp. 42-56. DOI: `https://doi.org/10.1007/978-3-319-45547-1_4`

P2. Rana Faisal Munir, Sergi Nadal, Oscar Romero, Alberto Abelló, Petar Jovanovic, Maik Thiele, and Wolfgang Lehner. Intermediate Results Materialization Selection and Format for Data-Intensive Flows[14]. In: *Fundamenta Informaticae, (Fundam. Inform. 2018)*. pp. 111-138. DOI: `https://doi.org/10.3233/FI-2018-1734`

P3. Rana Faisal Munir, Alberto Abelló, Oscar Romero, Maik Thiele, and Wolfgang Lehner. A Cost-based Storage Format Selector for Materi-

[13]This work has been carried out in collaboration with Besim Billali, who has helped in reviewing the paper and setting up the experiment's environment. Whereas, Rana Faisal Munir proposed the core idea, did its implementation and executed the experiments on the cluster.

[14]This work has been done together with Sergi Nadal and Petar Jovanovic. Specifically, Petar Jovanovic helped in formalizing the approach and devising the algorithm. Whereas, Sergi Nadal did the implementation and experiments part for the algorithm. Rana Faisal Munir extended Sergi's algorithm to include characteristics vector and did the comparison with an existing state of the art solution.

alization in Big Data Frameworks. In: *Distributed and Parallel Databases (DAPD 2019)*. DOI: https://doi.org/10.1007/s10619-019-07271-0

P4. Rana Faisal Munir, Alberto Abelló, Oscar Romero, Maik Thiele, and Wolfgang Lehner. ATUN-HL: Auto Tuning of Hybrid Layouts Using Workload and Data Characteristics. In: *European Conference of Advances in Databases and Information Systems, (ADBIS 2018)*. pp. 200-215. DOI: https://doi.org/10.1007/978-3-319-98398-1_14

P5. Rana Faisal Munir, Alberto Abelló, Oscar Romero, Maik Thiele, and Wolfgang Lehner. Automatically Configuring Parallelism for Hybrid Layouts. In: *European Conference of Advances in Databases and Information Systems, (ADBIS 2019)*. pp. 120-125. DOI: https://doi.org/10.1007/978-3-030-30278-8_15 (Short Paper)

P6. Rana Faisal Munir, Alberto Abelló, Oscar Romero, Maik Thiele, and Wolfgang Lehner. Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization. Submitted to Big Data (Under review)

The related work for all research problems is presented in Chapter 2. Remaining chapters present a solution for each research problem. Chapter 3 presents the solution for selecting IRs for materialization, Chapter 4 solves the problem of choosing storage layouts for chosen materialized IRs, Chapter 5 provides solution to fine-tune the configuration parameters of hybrid layouts, and Chapter 6 configures the degree of parallelism. In Chapter 7, we summarize this thesis and discuss the future work directions.

1.5.1 Chapter 3: Intermediate Results Materialization Selection

In this chapter, we proposed a generic framework, which can take any quantifiable SLA and based on these, find the best possible intermediate results for materialization using multi-objective optimization. We consider four SLA's metrics (i.e., loading time, storage cost, query cost, and freshness) for this study to show the importance and benefits of our approach. Freshness is considered to allow stale data in some materialized intermediate results, which help to reduce loading time for frequent update input sources. We evaluate our approach with an existing state of the art solution to show its accuracy and benefits.

1.5.2 Chapter 4: Storage Format Selection for Materialized Intermediate Results

This chapter focuses particularly on the storage of materialized intermediate results. We found that the first operation has direct impact on reading from the disk. Whereas, subsequent operations read all the data produced by the first operation. Thus, only the first operation can decide to read less data from disk. Consequently, we propose heuristic-rules based on the type of first operation and features provided by different storage layouts. We utilize the heuristic-rules in a hybrid approach to decide the storage layouts for the chosen intermediate results for materialization. The heuristic-rules are used for cold-start, when there is no statistical information available. When statistical information is available, the cost model is used to estimate the I/O cost of different storage layouts and choosing the one, which has overall minimum cost. Our detailed experiments on open source benchmarks show the significant reduction in the loading time of materialized intermediate results and overall, it improves the execution of DIFs.

1.5.3 Chapter 5: Auto Tuning of Hybrid Layouts Using Workload and Data Characteristics

Hybrid layouts are becoming a standard to store very wide tables, due to their built-in support for projection and selection operations. They also have support for dictionary encoding and compression on the stored data. These features help to read less data from the disk and further speedup the execution of DIFs. These built-in features of hybrid layout require many configuration parameters, which have default values. The default values can impact negatively on the execution of DIFs and degrade the execution performance. These parameters should be configured based on the characteristics of data and running workload.

In this chapter, we propose a cost-based approach to configure these parameters and find their best possible values based on the characteristics of data and workload. The cost model requires statistical information about data and workload. We have utilized single-column profiling technique to profile the data and job history files are used to extract the characteristics of the workload. Based on these characteristics, the best possible values are configured for hybrid layouts. Our detailed experiments show the improvements in the execution of DIFs.

1.5.4 Chapter 6: Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization

Resource provisioning is always a challenge for the execution of DIFs. There is no straight way to decide how many tasks and machines are optimal for executing a given DIF. Specifically, when DIFs read data from hybrid layouts, that help to read less data for certain operations (i.e., projection and selection). However, modern distributed processing frameworks do not consider this factor, when deciding the number of tasks to execute a DIF. They always decide the number of tasks based on the total table size, which leads to the tasks over-provision. This happens due to idle tasks, which do not have any data to process but still add extra overhead and require computational resources for execution. This can be tackled if the number of tasks and computational resources are decided based on the amount of actual data read from the disk.

In this chapter, we utilize our cost model to estimate the reading size for hybrid layouts and use it further in a multi-objective optimization method to decide the number of tasks and computational resources. Our approach helps to avoid over-provisioning and also helps in saving money and energy. Our experimental results also show the improvements in the execution of DIFs.

2

Related Work

In this chapter, we present the related work for all research problems, categorized into different sections. Each section presents the related work of an individual research problem.

2.1 Intermediate Results Materialization

The related work of intermediate results materialization are categorized into two main research domains, namely, *relational databases* and *distributed processing frameworks*. These both research domains have proposed different solutions to improve the execution of queries by materializing the common parts.

2.1.1 Relational Databases

There have been extensive research carried out in relational databases to improve the query performance by utilizing materialized views. These existing works can be categorized broadly into views selection, multi-query optimization, and sub-expressions elimination.

Views Selection. [55] presents a detailed survey on view selection methodologies in relational databases. These methodologies expect a workload and schema as an input and try to pick a set of views for materialization by finding the trade-off between query processing and view maintenance costs. These works expect that the workload is redundant and iterative.

Additionally, the research community of data warehouses [31] has also studied the view selection problem in detail for improving the execution of ETLs by materializing and reusing the redundant parts. Modern database systems also have a built-in physical adviser (e.g., [4]) that analyzes the workload history and based on that, proposes a set of possible views for materialization to improve the execution of queries.

The main focus of these research works is to improve the query execution time by utilizing materialized views. They only consider query processing and view maintenance cost as possible metrics for SLAs. There is no possibility to add more SLAs according to the business requirements. For instance, in some cases, it might be a good idea to use stale data (i.e., freshness of a materialized view) to fetch results quickly. However, this is not possible in these existing solutions.

Multi-query Optimization. The main focus of multi-query optimization is to optimize the execution of concurrent running queries by reusing the redundant parts. The materialization is temporary and the materialized intermediate results are deleted as soon as queries finished their executions. There have been extensive works for relational databases [51, 67, 70], which focus on concurrent workload and temporary materialization without any type of SLAs.

Sub-expressions Elimination. These works [73, 86] try to find the common sub-expression within a query or among a set of concurrent running queries. They also keep the materialized output in memory and discard it after finishing the execution.

2.1.2 Distributed Processing Frameworks

Similar to relational databases, distributed processing frameworks also have work on materialized view selection (i.e., materialized intermediate results) and multi-query optimization (i.e., sharing computations). These works explicitly focus on distributed processing frameworks and their technologies,

which are different from relational databases and also have different challenges.

Materialized Intermediate Results. There are research works [21, 50] explicitly focusing on distributed processing frameworks to choose a set of nodes for materialization and reusing them in future executions. These works are based on heuristic rules, which they use while selecting intermediate results for materialization. They are also tightly coupled with technologies, because the rules are designed based on the technology used for DIFs. Additionally, they do not consider multiple SLAs based on the business requirements.

The closest approach to ours is [64], however it focuses on performance aimed to cloud environments. Their goal is to find a set of materialized views and a storage platform (i.e., relational databases or Hadoop cluster) to improve the execution of DIFs. They accomplish this by using a cost model, which estimates the performance improvements using materialized views and different storage platforms. Based on these costs, they decide a set of materialized views and a platform to store them. However, they consider only a fixed SLA metric (i.e., performance gain using materialization and different storage engine). There is no way to introduce a new metric in their framework.

Sharing Computations. The distributed processing frameworks have also utilized the idea of multi-query optimization for optimizing the concurrent running DIFs. Existing research works [11, 62, 84] focus on concurrent running DIFs, whereas our goal is to improve the execution of recurrent DIFs.

	Materialization	Workload	SLAs
Views selection	Yes	Iterative	Partial (query cost and view maintenance cost)
Multi-query optimization	No	Concurrent	Partial (query cost)
Sub-expressions selection	No	Concurrent	Fixed
Materialized intermediate results	Yes	Iterative	Fixed
Sharing computations	No	Concurrent	Fixed
Our approach	Yes	Iterative	Any quantifiable SLAs

Table 2.1: Summary of related work for intermediate results materialization

Summary. Table 2.1 provides a summary of different research lines on intermediate results materialization. View selection and materialized intermediate results have the same objective to ours. However, the difference is in that they consider only two SLA metrics, one for improving the query execution and second for reducing the view maintenance cost. These SLAs, in the way they are proposed, they are tightly coupled and as such, it is not possible to introduce a new quantifiable SLA (e.g., Freshness). On the other hand, all other research lines focus on the concurrent executions of DIFs rather than on iterative execution. Thus, they use temporary materialization and they do not consider any maintenance cost.

In summary, most of these works do not support multiple SLAs, whereas our approach can take any type of quantifiable SLAs along with their associated characteristics vector. Moreover, our approach focuses on iterative DIFs and would materialize intermediate results for future re-usage based on all the given SLA's metrics.

2.2 Storage Layouts for Materializing Intermediate Results

Typically, the selected materialized intermediate results are stored on a distributed file system, where I/O operations are expensive. Thus, the writing and reading of materialized IRs can impact the execution of DIFs. There are many storage layouts proposed for reducing the I/Os cost, however these storage layouts are optimized for different types of workload.

These storage layouts can be categorized into *horizontal*, *vertical*, and *hybrid*. *Horizontal layouts* store their data row-wise and are good for scan-based workloads. Whereas, *vertical layouts* divide data into vertical partitions (known as column groups) and are good for projection-based workloads. On the other hand, *hybrid layouts* divide data into horizontal partitions and inside each partition, they store data column-wise. These layouts provide support for both projection and selection operations. However, they are not good for scan-based workload, because they have an extra cost of row reconstruction.

As mentioned earlier, there is no layout that can be used for all types of workload. Thus, researchers have proposed the use of multiple storage layouts in a single system. For example, DB2 [65] utilizes both horizontal and vertical layouts in the same table-space and the in-memory DBMS SAP HANA [23] also uses horizontal and vertical layouts for On-line Transaction

Processing (OLTP) and On-line Analytical Processing (OLAP) workloads, respectively. However, these layouts are pre-determined and non-modifiable at run-time.

The use of multiple layouts also pushes the organizations to go for multi-database environments for supporting different types of workload. For instance, Polybase [20] uses both a Hadoop cluster and a relational database for storage. It dynamically decides the system for execution based on the running workload and based on its decision, it may also move data from one system to another. Similarly, [39] keeps multiple copies of the same data in different storage layouts and based on the workload, it chooses the most appropriate. In addition, there are systems [22, 42, 69], that choose a storage engine by considering the data access patterns. In [22], the system requires training in order to take the right decision in choosing the best storage engine for queries. Furthermore, this training runs every query in all available systems to see which system is good for most of the queries.

There are research works [5, 41] that have vertical layouts as their base storage layout and create different column groups based on the running workload. These systems also have a cost model for estimating the reading cost from different column groups. However, the drawback of these approaches is that the creation of column groups is NP-hard problem and it is not feasible for a very wide table which has thousands of columns.

There are few works done for very wide tables [9, 10] and nested data [7]. [9, 10] help in reducing the seek cost in a wide table by storing the columns in the appropriate order based on the access patterns. This approach helps to reduce the disk cost and, overall, it reduces the execution cost of different queries. However, it considers only hybrid layouts in their study and it provides a cost model only for estimating the seek cost. [7] proposes a caching approach for nested data (i.e., JSON), which helps to keep more frequently used data in the cache by storing them in appropriate layout, according to the running workload. This work also supports our hypothesis to use different layouts for different type of workloads. However, it is limited to only nested data and not applicable to other scenarios.

[8] has proposed a cost model for Spark, which helps to estimate the cost of different query plans and decides the best one based on its cost. They assume the number of tasks and executors are fixed. Their main goal is to evaluate different query plans and choose the best one. However, this work can be complementary to our approach and would optimize the overall query

plan from planning to execution. [37] has proposed a cost model to estimate the reading and writing costs for different schemas presented in a multi-schema database. This work focuses on finding the best schema based on a given query, rather than focusing on their physical storage layouts.

	System modification	Cost Model	Workload
Horizontal layouts	No	No	Scan
Vertical layouts	No	No	Projection
Hybrid layouts	No	No	Projection and selection
Custom vertical layouts	Yes	Yes	Scan and projection
Our approach	No	Yes	Scan, projection, and selection

Table 2.2: Summary of related work for choosing storage layouts for materializing IRs

Summary. Table 2.2 summarizes the related work of choosing storage layout for materializing IRs. We summarize only those works, which directly use or propose physical storage layout for storing the data. To the best of our knowledge, there is no related work focusing explicitly on materializing IRs. Moreover, the use of multiple systems is also out of our scope.

There are already many storage layouts, which are supported by default in the existing distributed processing frameworks. Thus, we do not have to modify anything in these systems to use them. However, the customization proposed on top of vertical layouts is not possible without any modification. It requires an organization to change their current running infrastructure. Typically, organizations hesitate for any new modification. Specifically, when they require to move their data form old storage layout to a newly proposed storage layout. Thus, we propose a framework, which does not require any modification in the current running system but still supports all types of workload. Additionally, we also propose I/O cost model for horizontal, vertical, and hybrid layouts for estimating reading and writing costs. The cost model helps in choosing a storage layout, configuring its parameters, and controlling the degree of parallelism.

2.3 Configuring Parameters of Hybrid Layouts

There are many techniques proposed for optimizing hybrid layouts. Typically, these techniques store extra metadata to improve the performance of selective queries. For instance, SmartFetch [24] proposes an indexing technique for hybrid layouts, which stores extra metadata (i.e., indexing information) per row group and also per data node. This metadata are used in selective queries to filter row groups better, compared to the default metadata of hybrid layouts. However, this approach uses default size of row group (i.e., 128MB).

Researchers have also proposed few partitioning techniques [77, 78] to improve the execution of selective queries. These techniques compute extra metadata based on the predicates presented in different queries. These predicates are referred as features, where a bit is stored for each tuple matching a feature. This eventually computes a feature-vector for every tuple and this is stored as extra metadata per horizontal partition. The feature-vector metadata helps in skipping horizontal partitions more efficiently.

There are few research works [9, 10] proposing to optimize a table layout by storing columns in different orders. They focus on reducing the seek cost in a table by re-ordering the columns. The columns are re-ordered based on the access patterns. Thus, the columns that are accessed together are stored together. This approach also proposes the duplication of some columns.

Modern distributed processing systems (e.g., Spark and Hadoop) have many configuration parameters, which should be tuned to improve the execution of different workloads. Researchers have proposed many techniques [15, 27, 34, 36, 63] to find the optimal values of different parameters. [63] proposes a trial and error approach to tune the configuration parameters of Spark. Similarly, [27] proposes a methodology to profile the impact of different parameter pairs on benchmarking applications, by applying a graph algorithm to create complex candidate configurations. These configurations are checked in parallel and then, the best performing one is chosen.

Additionally, there is a research work [15], that profiles the bottlenecks (i.e., JVM, GC, serialization, etc.) of TPC-H queries and parameters are manually configured to avoid these bottlenecks, which significantly increase the query performance. These existing works focus on improving the execution of the overall system rather than specifically focusing on tuning the parameter of hybrid layouts. Our approach can take benefit from these techniques

to improve the overall execution of DIFs.

	System modification	Extra Metadata	Default configuration	Cost model
SmartFetch	Yes	Yes	Yes	No
Partitioning techniques	Yes	Yes	Yes	No
Table layout optimization	Yes	Yes	Yes	Yes[1]
Tuning distributed processing systems	No	No	No[2]	No
Our approach	No	Yes	No	Yes

[1] They provide a cost model to estimate seek cost
[2] They do not focus on configuring Hybrid Layouts

Table 2.3: Summary of related work for configuring parameters of hybrid layouts

Summary. Table 2.3 summarizes the related work for configuring parameters of hybrid layouts. These research works require extra metadata to work properly, which leads to modify the existing infrastructure. Additionally, they always use default configuration of hybrid layouts. There are also few works, which have proposed a cost model but these cost models are not usable for configuring the parameters of hybrid layouts. On the contrary, our approach does not require extra metadata except statistical information about data. Moreover, it proposes a cost model which explicitly focus on fine-tuning the configuration parameters of hybrid layouts.

2.4 Configuring Parallelism for Hybrid Layout

The degree of parallelism is crucial in distributed processing frameworks, because it helps to decide how much computational power is required to run a given DIF. There are few solutions [60, 83] from Hadoop community to control the number of mappers and reducers. In [60], the elbow curve technique is used to find the trade-off between number of tasks and execution time. This helps to find the right number of tasks where execution time is minimized. Similarly, [83] utilizes a multi-objective approach for estimating the number of tasks by considering a deadline constraint. These approaches do not consider the amount of data read, while estimating the number of

tasks, but only estimate the tasks based on the available number of machines and some objectives (such as deadline). As previously argued, the amount of data read is an important factor in deciding the number of tasks.

Similarly, the cloud community [35, 40, 72, 79] also proposes many solutions to control the resource provisioning by controlling the number of tasks execute in parallel. Additionally, [85] surveyed related works on energy-efficient techniques for big data analytics and categorized them into five. One of them (i.e., energy-aware resource allocation) focuses on deciding the number of machines to execute a given query with the aim to save energy. These works from both cloud computing and energy-efficient big data analytics focus more on deciding the number of machines to process an application. They aim at saving energy and computational resources, which indirectly leads to cost savings. However, they make these decisions without considering the reading size. Our approach could help them to decide resource provisioning in more granular level and overall, it can help these works to achieve their goals more efficiently.

The shuffle phase is always a bottleneck in distributed processing frameworks and [17] explicitly focuses on improving the shuffle performance in Spark by controlling the total number of shuffle files. This approach consolidates multiple shuffle files into one based on the available cores. This helps in improving the execution time of shuffle phase.

Nevertheless, these existing works do not explicitly consider the degree of parallelism. Their main aim is to decide number of machines for improving query execution time, which might lead to allocate unnecessary computing resources, because they do not consider data read.

	Makespan[1]	Resource Consumption	Cost Model
Estimating Number of Tasks	Yes	No	Yes[2]
Resource Provisioning in Cloud	No	Yes	No
Tuning Shuffling Phase	Yes	No	No
Our approach	Yes	Yes	Yes

[1] Makespan is the total time taken to execute a given DIF
[2] Yes, to estimate number of tasks for given constraints

Table 2.4: Summary of related work for configuring parallelism for hybrid layouts

Summary. Table 2.4 summarizes the related works for configuring parallelism. The research works for estimating number of tasks and tuning shuffle phase only focus on improving execution time. On the other hand, the cloud community only tries to reduce resource consumption without considering makespan. Additionally, all these works do not consider amount of data being processed, which is an important factor when deciding the number of tasks. Thus, our approach takes into consideration the amount of data read and provide better estimation for number of tasks and machines.

3

Intermediate Results Materialization Selection

Data-intensive flows deploy a variety of complex data transformations to build information pipelines from data sources to different end users. As data are processed, these workflows generate large intermediate results, typically pipelined from one operator to the following ones. Materializing intermediate results, shared among multiple flows, brings benefits not only in terms of performance but also in resource usage and consistency. Similar ideas have been proposed in the context of data warehouses, which are studied under the materialized view selection problem. With the rise of Big Data systems, new challenges emerge due to new quality metrics captured by service level agreements which must be taken into account. In this chapter, we propose a novel multi-objective approach for automatic selection of intermediate results for materialization in data-intensive flows, which can tackle multiple and conflicting quality objectives. The experimental results show that our approach provides 40% speedup on average with respect to the current state-of-the-art solutions.

Co-authoring declaration. This work has been done together with Sergi Nadal. Precisely, the introduction (Section 3.1), problem formulation (Section 3.2) and definition of cost model for intermediate result materialization selection (Section 3.3) were done jointly with equal contribution. The state space search algorithm (Section 3.4) was mainly developed by Sergi Nadal, which is extended to include the characteristic vector. The experimental evaluation (Section 3.5) was done by Rana Faisal Munir.

3.1 Introduction

Nowadays, many organizations are shifting their business strategy towards data analytics in order to guarantee their success. In the past, the vast majority of analyzed data was transactional, however the emergence of Big Data systems allows a new range of data analytics, by replacing traditional extract-transform-load (ETL) process with much richer data-intensive flows (DIFs) [45]. This new range of data analytics is supported by the Hadoop[1] ecosystem which has a distributed storage system (Hadoop Distributed File System - HDFS[2]) to store large scale data and a processing engine (i.e., MapReduce [18]) to execute DIFs. It works on a distributed cluster of commodity hardware which provides competitive advantage to organizations by reducing their hardware costs. In addition, many modern cloud providers offer pay-per-use services to organizations by implementing the big data systems under service level agreements (SLAs).

An in-depth study of analytical workloads, in Big Data systems across seven enterprises, shows that user workloads have high temporal locality, as 80% of them will be reused by different stakeholders on the range of minutes to hours [14]. Thus, providing partial materialization of results in shared flows can clearly bring benefits by saving computational resources. The materialization boils down to the traditional data management problem of *materialized view selection* [33], which is well-known to be NP-hard [29]. There are some works[21, 62, 84], which have tackled the problem of finding the optimal partial materialization in DIFs, however all of them are specific to the MapReduce framework and only aim at optimizing the system performance-wise by ignoring other relevant SLA's metrics (such as freshness, reliability, scalability, etc.[75]).

[1]https://hadoop.apache.org
[2]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Similarly, the existing materialized view solutions in relational databases [31] focus only on improving query execution time without considering multiple generic SLAs. Moreover, the aforementioned solutions do not consider different characteristics associated with different SLAs. For instance, in some organizations, they allow to get results from a stale materialized node (i.e., to allow low freshness) for a certain time period to reduce the loading cost. These characteristics can be expressed separately and the optimal value should be chosen for each materialized node. These shortcomings of existing solutions are addressed by our proposed approach, which is a technology independent materialization solution and can take into consideration generic quantifiable SLAs with their associated characteristics.

3.1.1 Motivational Example

To motivate our work, we present a DIF, shown in Figure 3.1, which depicts a high-level representation containing relational operations and User Defined Functions (UDFs). It uses five input sources and serves three queries. Each data source and data operator is labeled by its estimated processing cost (i.e., consumed resources, in seconds) and storage cost (in GB). Note that data processing entails extracting and loading data from the sources into the data processing system.

For the sake of this example, let us suppose that all the sources update once per day, except *Source 1* and *Source 3* that have a update frequency of 6 and 4 times per day, respectively. *Query 1*, *Query 2* and *Query 3* have a frequency of 2, 20, and 10 times per day, respectively. In addition, let us assume that we allow stale materialized results (i.e., we do not update materialized nodes with each update in their associated input sources) and it is provided as a characteristic vector (given as number of updates per time unit $[1, 2, ..., n]$).

In this example, we focus on optimizing four SLA's metrics (i.e., time to load, time to query, space needed to store intermediate results, and freshness). *Loading time* is measured by the sum of processing cost from the sources to the partial materializations, *query time* is measured by the sum of execution cost from the partial materializations to the user's output, *storage space* is measured by the sum of storage cost for the selected partial materializations, and *freshness* is measured using the cost function presented in Section 3.3.3.

Several of the existing intermediate result materialization approaches from

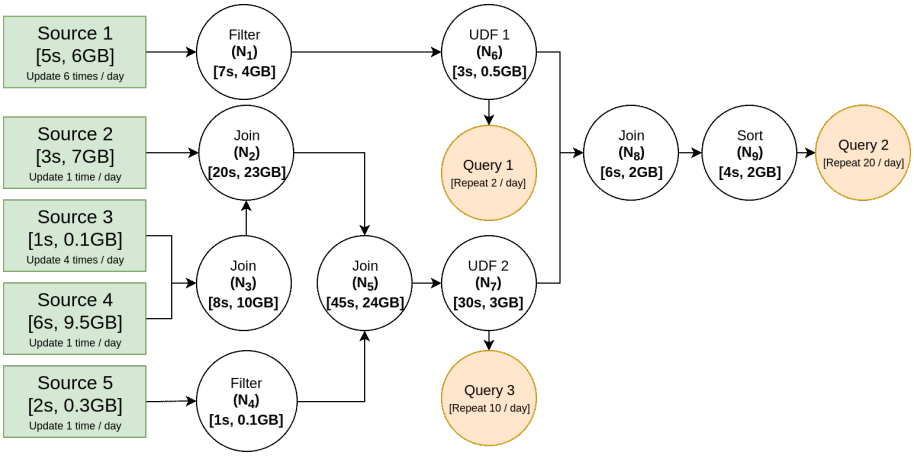


Fig. 3.1: An Example of a DIF

Big Data Systems can be used, as discussed in Section 2.1. Let us focus on one of them, namely ReStore [21], which uses two kinds of heuristics (i.e., conservative and aggressive) in order to choose DIF nodes for materialization. Conservative heuristics materialize the output of those operators that reduce the input size (i.e., project and filter). The aggressive heuristics materialize the output of those operators which produce large outputs and those known to be computationally expensive (i.e., join, group and cogroup). Table 3.1 shows the selected nodes and a quadruple with the costs (i.e., loading time, query time, storage space, and freshness). It should be noted that *query time* represents the sum of all three queries' time. The first two columns show both ReStore's heuristics, while the rightmost shows a pareto-optimal solution (i.e., a solution that cannot be improved further in the presence of multiple conflicting SLAs) in boldface.

	ReStore Cons.	ReStore Agg.	Pareto-optimal
Nodes	N_1, N_4	N_2, N_3, N_5, N_8	$\mathbf{N_7}$
Cost	$\langle$75 s, 257 s, 4.1 GB, 1$\rangle$	$\langle$336 s, 115 s, 59 GB, 1$\rangle$	$\langle$**101 s, 100 s, 3 GB, 0.78**$\rangle$

Table 3.1: Selected intermediate nodes and cost for the four SLA's metrics (load, query, store, freshness)

As shown in Table 3.1, ReStore conservative heuristics choose N_1 and N_4. They take more time in loading, due to *Source 1*, which updates very frequently and effects the loading cost of N_1. We calculate the total load time of a node by multiplying its load time with the update frequency of its input

sources. Furthermore, both N_1 and N_4 do not provide good speedup because they have high query time. ReStore aggressive heuristics choose N_2, N_3, N_5, and N_8. These nodes help to reduce the query time, but require more space to store and more time to load. It should be noted that ReStore does not support freshness. If the input sources change, it deletes all their dependent materialized nodes. This pull-strategy provides a fixed degree of freshness and thus, we set it to 1 in our quadruple.

Finally, the pareto-optimal solution considers four SLA's metrics (i.e., loading time, query time, storage space, and freshness) together by assigning them the same weight, and based on them, it chooses only node N_7, which provides better speedup compared to ReStore's heuristics. Even though, N_7 depends on *Source 3* , which has a high loading cost due to its high update frequency, it is still worth to materialize because it is reused more often by repetitive queries (i.e., Query 2, Query 3). Moreover, the loading cost can be improved by choosing the optimal value of refresh frequency for N_7. The possible values are $[1, 2, 3, 4]$, where 4 will provide the maximum freshness. The pareto-optimal solution chooses 3 as the refresh frequency for N_7, which helps to improve the load time and it also provides a good degree of freshness.

The above given example shows that the state-of-the-art solutions produce suboptimal results in the case of different SLAs. To address this problem, we revisit the traditional frameworks for materialized view selection [80] and analyze their applicability and extensions in the context of DIFs for Big Data systems. As a result, we present an approach to automatically select the optimal materialization of intermediate results, driven by multiple quality objectives represented as quantifiable SLAs with their associated characteristics. This is achieved by implementing a multi-objective optimization technique (discuss in Section 3.4) which efficiently tackles multiple and conflicting objectives to select materialized nodes.

Contributions. The main contributions of this chapter are as follows:

- We propose a novel cost model for multi-objective selection of optimal partial data materialization for DIFs.
- We present a local search algorithm that, driven by a set of SLAs, probabilistically selects a set of near-optimal intermediate results to materialize.
- We assess our method and show its performance gain by using the

TPC-H benchmarking suit.

Outline. The rest of the chapter is structured as follows. Section 3.2 presents the theoretical building blocks and formalizes our approach. Sections 3.3 and 3.4 present the cost model for intermediate result selection and the algorithm to explore the search space. In Section 3.5, we present the experimental results, while Section 3.6 concludes the chapter.

3.2 Formal Building Blocks and Problem Statement

3.2.1 Multiquery AND/OR DAGs and Data-Intensive Flows

The general framework for materialized view selection [80] relies on multiquery AND/OR Directed Acyclic Graphs (DAGs). As defined in [82], a query DAG is a bipartite graph $\mathcal{G}$, where AND nodes (or *operational nodes*) are labeled by a relational algebra operator, and OR nodes (or *view nodes*) are labeled by a relational algebra expression. Moreover, given a set of queries Q defined over a set of source relations R, a multiquery DAG $\mathcal{G}$ is a query DAG, which may have multiple sink (query) nodes. Roughly speaking, the materialized view selection problem can be expressed as a search space based problem over the multiquery DAG $\mathcal{G}$. Additionally, [81] formalizes the output of such problem as a data warehouse (DW) configuration $C = \langle V, Q^V \rangle$, where Q^V is the set of queries in the query set Q rewritten over the view set V. Note that there exist two special DW configurations: $\langle Q, Q^Q \rangle$ which represents a materialization of the query set Q and $\langle R, Q \rangle$ which represents a complete materialization of the source data stores R.

However, the multiquery AND/OR DAGs fail to capture the complex semantics present in DIFs operators, as they solely rely on relational operators. To this end, in this chapter we build upon the ideas from the aforementioned frameworks and adapt them for the case of DIFs, which consider more complex data transformations [46]. It is straightforward to see that any multiquery DAG $\mathcal{G}$ can be represented as a DIF, however the opposite does not hold due to the fact that AND/OR DAGs are solely based on relational operators, while DIFs are extended with more complex operations. Thus, in this chapter, we extend the notion of DW configuration to **Big Data system (BDS) configuration** for the case of DIFs. Hereinafter, we will depict a BDS configuration as a set of nodes from the DIF to materialize $B = \{b_1, \ldots, b_n\}$. In the following sections, we describe the specific components for the prob-

lem in-hand, and reformulate the materialized view selection problem in the context of BDS.

3.2.2 Components

Data-Intensive Flow. In this chapter, we adopt the notation from [49], hence we define a DIF D as a DAG (V, E) where its nodes (V) are the flows' operational nodes, and its edges (E) represent the directed data flow. Operational nodes are defined as $o = \langle \mathbb{I}, \mathbb{O}, \mathbb{S}, V_{pre} \rangle$, where $\mathbb{I}$ and $\mathbb{O}$ are sets of respectively input and output schemata (a DIF can have multiple inputs and outputs), where each schema is defined as a finite set of attributes, $\mathbb{S}$ expresses operator's semantics, and V_{pre} a subset of attributes of the input schemata ($V_{pre} \subseteq \bigcup_{I \in \mathbb{I}} I$) whose values are used by o.

Design Goal (DG). $\mathbb{DG}$ represents a set of design goals, where each (DG_i) characterizes an SLA. It can be specified as either a minimization or a maximization of an objective cost function, or alternatively as a boundary that must not be surpassed in such cost function. Formally:

- *Min/Max*: From a set of BDS configurations $\mathbb{B}$, it returns the minimal B by means of evaluating the cost function CF, defined as $DG_{min}(\mathbb{B}) = \min_{B \in \mathbb{B}} (CF(B))$. Note that maximizing the cost function is equivalent to the negation of minimization.
- *Constraints*: For a BDS configuration, it checks whether the evaluation of the cost function CF fulfills the constraint, formally $DG(B) = [CF(B) \leqslant K]$, where K is a variable to enforce an upper or lower limit. Note that the constraint can express an arbitrary logical predicate (e.g., $<, >, \geqslant$). It is important to note that DG(C), where C is constraints, in this case is binary true/false and it differs from the above which gets the value obtained from the cost function.

Cost Function (CF). Given a BDS configuration, $\mathbb{CF}$ represents a set of cost functions where each (CF_i) is the estimation of an SLA for B. Hence, we define $CF(B) = \sum_{b \in B} E(b)$ (where $E(b)$ is the cost estimation of an element $b \in B$).

Characteristics Vector (CV). Some costs are determined once a node is chosen, but for SLAs, we can select arbitrary values for the features that impact

them. For instance, in some organizations, they allow stale data for some period of time, which can be defined as a refresh frequency for every materialized node. Thus, the refresh frequency should be chosen to maximize the overall benefit. We keep a vector of such choices. Each position of the vector represents a characteristic affecting some SLAs. These characteristics will impact on the associated cost function CF.

Utility Function. In the context of multi-objective optimization (MOO) [56], it is common to aggregate all objectives $DG_1, \ldots, DG_n$ into a single value to obtain the global utility. Such function, known as the *utility function* U as it measures benefit, is formally defined as $U(\mathbb{DG}) = U(CF_1(B), \ldots, CF_n(B))$. Each $CF_i(B)$ provides a quantitative evaluation of B (it can be seen as individual utility functions for each cost function) for its associated DG_i, in the case of min/max design goals, or 0, and $+\infty$ for satisfied and non-satisfied constraints, respectively. Generally in MOO high utilities are preferred. However, in our context there are some CF where we aim for minimal utilities (e.g., query time).

3.2.3 Problem Statement

We state the problem of intermediate results materialization selection and format in DIFs as: given a data-intensive flow D, a set of design goals $\mathbb{DG}$, a set of cost functions $\mathbb{CF}$, a characteristic vector $\mathbb{CV}$, a utility function $U(\mathbb{DG})$, and a cost model represented by a set of estimators over D calculated by means of statistical information from sources, return a BDS configuration B', such that, $U(\mathbb{DG})$ is minimal or maximal based on the given CF, each $b \in B'$ with its optimal characteristic values chosen from $\mathbb{CV}$.

3.3 Cost Model

In this section, firstly we present our approach to estimate statistics for each operation of a DIF. We assume that the statistics of each input source are available. Secondly, we discuss the metrics (i.e., execution and storage) that we consider in this chapter. It should be noted that we choose to ignore the CPU cost, and focus only on the I/O operations. Also, regardless of being executed in parallel, the overall execution cost of the flow will remain the same (only time span would be reduced). Finally, we use the proposed

metrics to estimate the cost of SLAs. In this chapter, we present the cost functions for four SLA's metrics (loading, query, storage, and freshness).

3.3.1 Data-Intensive Flow Statistics

As previously mentioned, cost functions are computed from estimators. Every operational node in a data-intensive flow D might have several estimators, each assessing a single SLA metric (e.g., an execution cost), where they perform a cost based estimation according to the operator's semantics. In order to devise more accurate metrics, some essential statistics must be obtained from the input data stores and propagated across D. By topologically traversing D, we can propagate such statistics at each node, based on the specific semantics of operators. In [30], the authors describe a complete set of statistics which are necessary to perform cost based estimations for DIFs. Here we focus on the following subset: selectivity factor $sel_P(R)$, number of distinct values per attribute $V(R.a)$ and cardinality $T(R)$. R denotes an input data store, while $R.a$ is an attribute of R. Note that statistics only consider logical properties of the flow, hence they are independent of the underlying engine where the flow is executed.

Example 3.3.1
Let us assume a *JOIN* operator $R' = R \bowtie S$ (e.g., N_6 in Figure 3.1), with input schemata $R(a, b)$, $S(c, d)$ and semantics $P_{R.a=S.c}$. Inspired by the work in [26], we propose measures for the above-mentioned statistics for this *JOIN* operator (we have done likewise for the rest of operators) as:

$$sel'_P = \begin{cases} \dfrac{1}{V(R.a)}, & \text{if } domain(S.c) \subseteq domain(R.a) \\ \dfrac{V(R.a \cap S.c)}{V(R.a) \cdot V(S.c)}, & \text{otherwise} \end{cases}$$

$$V(R'.att_i) = V(R.att_i) \cdot (1 - sel_P)^{\frac{T(R)}{V(R.att_i)}} \qquad T(R') = sel'_P(R \bowtie S) \cdot T(R) \cdot T(S)$$

The selectivity factor is obtained as the fraction of the number of shared values in the *JOIN* attributes, when the domain of the right-hand side is contained in the domain of the left-hand side (i.e., analogously to Primary Key(PK)-Foreign Key(FK) relations), otherwise an estimation is made as a fraction of shared values and its Cartesian product. Regarding the number of distinct values for an attribute, it is estimated as the input number of

distinct values, multiplied by the probability that no repetitions of a value are selected. Finally, the cardinality is measured likewise the relational case.

3.3.2 Metrics

Once statistics for D have been calculated, they can act as building blocks for metrics. Here, we focus on estimating both performance-wise ($Execution_{estimator}$) and space-wise ($Space_{estimator}$) metrics. Performance metrics are measured by means of estimated disk I/O (in blocks) and space metrics by the number of disk blocks occupied by the intermediate results materialization. It is worth noting that in terms of execution, the CPU cost is negligible as opposed to I/O cost [3], hence we ignore CPU cost and focus on the I/O cost of operators. Therefore, non-blocking operational nodes (acting as pipelines) will not incur any cost for such $Execution_{estimator}$.

To devise metrics, certain characteristics of the underlying engine are required. We focus on the following subset: the size of a disk block B, the number of main memory buffers available M, and the size in bytes that each attribute occupies $sizeOf(att_i)$. For instance, in the Oracle relational database the block size is approximately of 8KB, while in Hadoop's HDFS it is 64MB or 128MB. The incurred space of intermediate results is measured by means of the estimated number of blocks generated. However, this will vary according to the underlying schema that such results have and therefore, we need to make this calculation based on the record length, that is $sizeOf(att_i)$ (including the corresponding control information). Thus, the specific number of blocks for an input R is measured as:

$$B(R) = \left\lceil \frac{T(R)}{\left\lfloor \frac{B}{\sum sizeOf(att_i)} \right\rfloor} \right\rceil$$

Example 3.3.2
Given the *JOIN* operation from example 3.3.1, one implementation of such operator is based on the block-nested loop algorithm, which scans S for every block of R using $M-2$ memory buffers (as the remaining two are used to perform tuple comparisons and output results), thus the estimation for execution and space costs is as follows:

$$Execution_{estimator} = B(S) + B(R) \cdot \left\lceil \frac{B(S)}{M-2} \right\rceil \qquad Space_{estimator} = B(R')$$

However, in a MapReduce environment, execution cost is dominated by data transfers (i.e., communication cost over the network) that occurs during the data shuffling phase between mapper and reducer nodes [2]. In such case, the natural implementation of a $JOIN$ is using the hash join algorithm, where the hash function maps keys to k buckets and data is shipped to k reducers. Assuming no data skewness, each reducer receives a fraction of $\frac{T(R)}{k}$ and $\frac{T(S)}{k}$. Having c as a constant representing the incurred network overhead per transferred HDFS block, the cost estimations of the *JOIN* are:

$$Execution_{estimator} = \frac{B(R) + B(S)}{k} \cdot c \qquad Space_{estimator} = B(R')$$

Note that the presented metrics can be highly variable within D. For instance, not surprisingly, *JOIN* nodes are the operations that consume the most time and space in order to generate intermediate results. Additionally, modern DIFs make heavy usage of User Defined Functions (UDFs) which consists of ad-hoc operations, difficulting the estimation of their I/O cost. An approach to solve this problem is to rely on static analysis of code to estimate the I/O cost for UDFs [38]. Finally, it is worth noting that other approaches exist to measure the presented metrics, for instance [74] proposes a method based on micro-benchmarking. On the contrary, our approach does not require any execution of the flow which however, impacts the quality of the estimation.

3.3.3 Cost Functions

In this section, we present a set of cost functions to evaluate a BDS configuration, based on metrics from the materialized operational nodes of D. In our experiments, we focus on traditional metrics used in multi-query optimization namely loading cost, query cost, storage cost and freshness. However, note that our approach is extensible to other types of metrics such as monetary aspects [61], energy consumption [66], etc. For instance to estimate monetary cost, the pay-per-use cloud services charge based on the resources

used, which can be estimated by our cost model. Further, the estimated resource utilization can be used to calculate the cost of renting machines on different cloud providers. Regarding storage, here we are not concerned with the layout to be used as this is assessed once the selection of nodes to be materialized has been found.

First, we must present some auxiliary methods over BDS configurations in which cost functions are based on. Let $Pre(b)$ and $Suc(b)$ be respectively the input and output subgraphs for a node b, recursively defined as $Pre(b) = b \cup \forall_{b_i \in predecessors(b)} Pre(b_i)$ and $Suc(b) = b \cup \forall_{b_i \in successors(b)} Suc(b_i)$, and respectively ending when $deg^-(b) = 0$ and $deg^+(b) = 0$. Hence, we can define the input and output subgraphs of a BDS configuration B as $I(B) = \bigcup_{b \in B} Pre(v)$ and $O(C) = \bigcup_{v \in C} Suc(v)$. Specifically, the former is a subgraph where its source nodes are the sources in a D and sink nodes are all the elements $b \in B$. The latter is a subgraph where its source nodes are all elements $b \in B$ and sink nodes are the final nodes in D. Additionally, $sources(b)$ gives the input sources of a node b and $sinks(b)$ provides the queries over a node b.

Loading Cost. The cost of loading a set of intermediate results CF_{LT} is the sum of processing source data and propagating them to the intermediate results in B. Our approach is valid for both maintenance and update of intermediate results, as long as source statistics are properly updated. From a BDS configuration B, the estimated loading cost is intuitively the cost of executing the operations of D, loading the intermediate results for each node $b \in B$ (i.e., $I(B)$), and the cost of writing such results to the disk. Thus, we define $CF_{LT} = \sum_{b \in B} [\sum_{b_i \in I(b)} Execution_{estimator}(b_i) * RF(b_i)] + \sum_{b \in B} Space_{estimator}(b)$. Here, RF represents the refresh frequency of materialized nodes which is fixed in the characteristics vector CV of each node. The unit of refresh frequency is *total number of updates per time unit*. It should be noted that we are talking about sequential files that do not provide random access, so only full update is possible (no incremental).

Query Cost. The query cost CF_{QT} is the sum of querying the intermediate results, transform the data and deliver results to the user. From a BDS configuration B, the estimated query cost is computed as the sum of execution costs of successor nodes for each node $b \in B$ (i.e., $O(C)$). However, note that the cost

of processing the operations of the nodes in B should not be taken into account as it is already evaluated in CF_{LT}. Therefore, it is necessary to consider only nodes in the set $O(B)\backslash B$, denoted $O^{+}(B)$. Finally, it is necessary to consider the cost of reading such intermediate results from the disk. Hence, we define $CF_{QT} = \sum_{b\in B}[\sum_{b_i\in O^{+}(b)}(Execution_{estimator}(b_i) * (\sum_{s_i\in sinks(bi)} QF(s_i))] + \sum_{b\in B} Space_{estimator}(b)$. Here, QF represents the frequency of queries. The query frequency can be expressed per day, hour or minute. QF helps to select the most reused node. Queries with high frequencies benefit more from the re-usage. Hence, a node which is used in highly repetitive queries will be given more weight during selection.

Storage Cost. The storage cost function CF_S concerns the storage space needed to store intermediate results. It is computed as the sum of estimated space for storing the results of each node in B, and it can be seen as the estimated space require to accommodate the deployed BDS configuration. It is defined as $CF_S = \sum_{b\in B} Space_{estimator}(b)$. Notice that $Space_{estimator}$ can be used for estimating the costs of reading and loading intermediate results, showed in CF_{LT} and CF_{QT}, as well as to estimate the occupied space for the case of minimizing or constraining its value.

Freshness. The freshness cost function estimates the freshness of the results of a query, which are obtained using materialized nodes, denoted as B'. For the freshness function, we build on the formula from [68]. The variable *age* tells how old data are in a materialized result with regard to the current data in the input sources. In our case, age cannot be known as it is not possible to foresee when materialized results are going to be used. It should be noted that update frequency of a node is calculated based on its input sources. Whereas, the refresh frequency is given in the characteristics vector CV of each node. Moreover, we assume that refresh frequencies in CV are synchronized with the input sources. We calculate the update frequency of a materialized node b as an average of the input frequencies of its input sources $UF(b) = Average_{s_i\in sources(b)} UF(s_i)$. Then, we can approximate *age* of b as the mid point between two refreshments $Age(b) = RF(b)^{-1}/2$. With such, we can measure the freshness of a node b as $Freshness(b) = (1 + UF(b) * Age(b))^{-1}$. Furthermore, $Freshness(b)$ is used to calculate the freshness of the results of a query Q as $CF_{Freshness}(Q_{results}) = Average_{b'\in B'} Freshness(b')$. This cost function helps to choose a node for materialization which provides up-to-

date results to the queries.

3.4 State Space Search Algorithm

As previously mentioned, the problem of intermediate results materialization in DIFs can be reduced to the general materialized view selection problem, known to be NP-hard. Hence, we must avoid exhaustive algorithms and rely on informed search algorithms. Furthermore, in this particular case, purely greedy algorithms will not provide near-optimal results as the proposed cost functions are not monotonic. In classic artificial intelligence, a state space search problem is usually represented with the following components: *(i) initial state* where to start the search; *(ii) set of actions* available from a particular state; *(iii) transition model* describing what each action does and what are the derived results from it; *(iv) goal test* which determines whether the evaluated state is the goal state (i.e., the optimal state); and *(v) path cost* function to assign cost to the actions path.

In our context, we see a state as any BDS configuration B over which action functions are applied. It is noteworthy to mention that in such problem we are not interested in the set of actions that have led to a solution, but in the solution itself, which is initially unknown. Additionally, as any state B is a valid solution, we drop the component of *goal state*. Furthermore, the path cost is substituted by the definition of a *heuristic function*, which will guide the search. In the following subsections, we present the particularities of our specific problem for the remaining components.

3.4.1 Actions

For a BDS configuration B, we can compute actions (navigations over the graph), yielding new BDSs B'. First, we define the generic navigation operation $B' = Nav(b_{origin}, b_{destination})$, with $b_{origin}, b_{destination} \in D$ and semantics $Nav(b_{origin}, b_{destination}) = \big(B \backslash \{b_{origin}\}\big) \cup \{b_{destination}\}$. We then define three specific actions applied over nodes in B:

1. *Forward* ($F(b, b') = Nav(b, b')$): characterizing a forward movement from b to b' in D, applicable when $b' \in successors(v)$.

2. *Backward* ($B(b, b') = Nav(b', b)$): characterizing a backward movement from b' to b in D, applicable when $b' \in predecessors(b)$.

3. *Stay* ($N(b) = \varnothing$): always applicable, as it does not perform any movement. Such operator is only useful when other nodes b' are combined with M or U, so that a new state is generated where b remains selected.

4. *Increment* ($Inc(b, i)$): Increases the value of a characteristic (identified by position i_{th} of the vector CV) for a node b.

5. *Decrement* ($Dec(b, i)$): On the contrary, it helps to decrease the value of a characteristic for node b at i_{th} position of the vector CV.

From the previous definitions, for each node b, we define the set $Actions(b)$ as:

$$\bigcup_{b_i \in successors(b)} \{F(b, b_i)\} \cup \bigcup_{b_i \in predecessors(b)} \{B(b, b_i)\} \cup \{N(b)\} \cup \{Inc(b, i)\} \cup \{Dec(b, i)\}$$

Finally, we obtain all possible actions from a BDS configuration B by computing the Cartesian product of the power set of each $Actions(b_i)$ (note, empty sets are removed from each power set but this is not depicted for readability) as $\mathcal{P}(Actions(b_1)) \times \ldots \times \mathcal{P}(Actions(b_n))$. The rationale behind this operation is to generate, for each node b, all combinations of movements. The usage of a power set is relevant for the cases when the input or output schemata of a node is not unary (e.g., a $JOIN$). Then, such different combinations are furtherer combined with the rest of nodes via a Cartesian product. Note that such set can be extremely large for complex Ds, however it is easy to see that many combinations generate invalid BDS configurations. To this end, we define the two essential conditions that a BDS configuration must fulfill in order to be valid, namely answerability and non-dominance.

Answerability of all queries. Ensuring that all queries (sink nodes) can be answered from materialized results. It can be checked by guaranteeing there is at least one materialized node for each path in D. Formally, $\forall b \in sources(D) \forall p_i \in Paths_{b,sinks(D)} \exists node \in p_i : node \in B$. For instance, in Figure 3.2a, we can see that the green-colored BDS configuration does not satisfy answerability as the path from N_2 to N_9 does not contain any materialized node.

Non-dominance of nodes. The purpose of our approach is to minimize the number of nodes to materialize by avoiding unnecessary materializations. For instance, if it is decided to materialize all sink nodes then it is unneces-

sary to materialize any intermediate node. In graph theory, a node *m dominates* *n* if all paths from the source node to *n* must pass through *m*. We extend this definition for the case of multiple nodes, and thus we test non-dominance of a set of nodes by checking that, for each node *b* there is at least one path from *b* to sink nodes where *b* is the only selected node. This is formally defined as $\forall b \in B \exists p_i \in Paths_{b,sinks(D)} : |\{\forall node \in p_i : node \in B\}| = 1$. For instance, Figure 3.2b, shows that the green-colored BDS configuration does not satisfy non-dominance, as nodes N_6 and N_7 dominate N_5.

Besides the two essential conditions, it is necessary to maintain a set of visited nodes to check whether a state *B* has not been already visited, and thus avoid unnecessary expansions in the search space. Figure 3.2c, depicts the valid BDS configurations obtained by applying actions to the BDS configuration {3,4}. Experimenting with the DIF in Figure 3.1, it has been observed that on average eliminating states that do not fulfill such conditions makes a reduction on the search space by 88%. Next, we generate increment and decrement actions for the current node to move vertically by using different index positions of *CV*. This helps to find the best possible values for given characteristics vector *CV* for each to-be-materialized node.

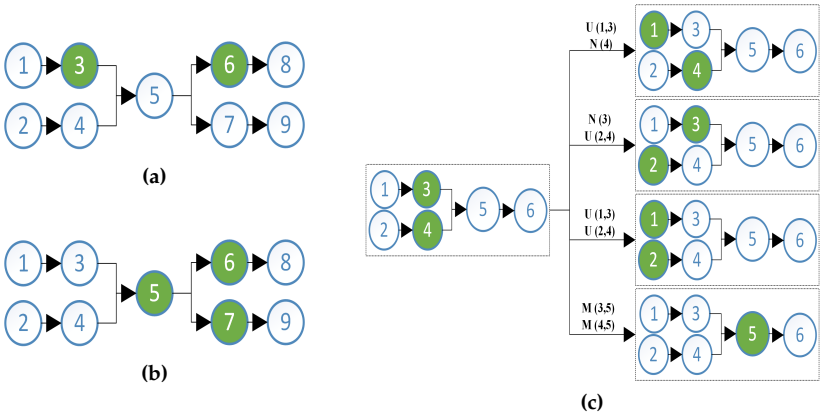


Fig. 3.2: (a) depicts a BDS configuration where answerability is not satisfied, (b) depicts a configuration where non-dominance is not satisfied, and (c) depicts the valid actions for configuration {3,4}

3.4.2 Initial State

As previously mentioned, the search space contains many local optimum points due to the non-monotonicity of cost functions, therefore the obtained

solution might vary depending on the initial state. Three possible initial states have been devised aiming to cover all search space varieties:

- Materialize all source nodes, representing the BDS configuration $B = sources(D)$.
- Materialize all sink nodes, representing the BDS configuration $B = sinks(D)$.
- Random selection of nodes, guaranteeing a valid initial state where answerability and non-dominance are satisfied. Further, for the random selected nodes, we also randomly choose values in all positions of the characteristics vector CV.

Note that the two former are special cases of the third, thus this is the strategy that has been chosen to generate initial states (we provide a more thorough discussion on this in Section 3.4.4).

3.4.3 Heuristic

Provided that values of the different objectives lay in very different ranges, and in order to provide a consistent comparison, it is necessary to make use of a non-dimensional utility function normalizing all objectives. There exist a vast number of different normalization strategies [28]. For our purpose, and given the nature of the problem, we make use of the *normalized weighted sum* as utility function, defined as:

$$h(B) = U(\{CF_1, \ldots, CF_n\}, B) = \sum_{i=1}^{n} w_i \cdot CF_i^{trans}(B)$$

$CF_i^{trans}(B)$ stands for the evaluation of the transformed cost function for B, being $CF_i(B)$ is evaluation CF_i (see Section 3.2.2), CF_i^o the *utopia* point (i.e., minimal BDS for CF_i), and CF_i^{max} the maximal BDS:

$$CF_i^{trans}(B) = \frac{CF_i(B) - CF_i^o}{CF_i^{max} - CF_i^o}$$

Such approach yields values between zero and one, depending on the accuracy of both CF_i^o and CF_i^{max} computation. However, it is mostly unattainable to get their exact values, and for that we have to rely on estimations. To achieve this, we compute estimations of utopian BDSs for all cost functions as the union of all minimum nodes for each path from source to sink nodes. Maximum points are obtained by following the similar approach, in

this case obtaining maximum nodes for each path from source to sink nodes. Note that, if design goals with constraints are presented, then it is possible to use such constraint value K as maximum point by dismissing the need of estimations.

3.4.4 Searching The Solution Space

Local search algorithms consist of the systematic modification of a given state, by means of *Action* functions, in order to derive an improved state. Many complex techniques do exist for such approach (e.g., simulated annealing or genetic algorithms). The intricacy of these algorithms consists of their parametrization, which is also their key performance aspect at the same time. In this chapter, we focus on *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search by always following the path that yields higher heuristic values. Since the used cost functions are highly variable due to their non-monotonicity, hill-climbing might provide different outputs depending on the initial state. In order to overcome such problem, we adopt a variant named *Shotgun hill-climbing* which consists of a hill-climbing with restarts (see Algorithm 1). After certain number of iterations, we can keep the best solution. Such approach of hill-climbing with restarts is surprisingly effective, specially when considering random initial states.

3.5 Experiments

In this section, we report our experimental findings on the evaluation of our approach for intermediate results selection. Our experiments are performed on an 8-machine cluster. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory and 1TB SATA-3 of hard disk. Each machine runs Hadoop 2.6.0 and Pig 0.15.0[3] on Ubuntu 14.04 (64 bit). We have dedicated one machine for the name node and the remaining seven machines for data nodes. We use TPC-H[4] benchmarking tool to generate datasets and queries. These queries have been converted to Apache Pig which is a procedural language of the big data systems. In order to create a complex DIF, we use CoAl [48], which in this case, combines six TPC-H queries into one integrated DIF as shown in Figure 3.3. The DIF size is chosen with the goal

[3]https://pig.apache.org
[4]http://www.tpc.org/tpch/

Algorithm 1 Shotgun Hill-Climbing

Input D, i: ▷ DIF, number of iterations
Output *solution* ▷ Solution BDS configuration

```
1: solution = null
2: do
3:     B = randomInitialState(D); finished = false
4:     while !finished do
5:         neighbors = ResultsFromActions(B)
6:         B' = stateWithSmallestHeuristic(neighbors)
7:         if h(B') < h(B) then
8:             B = B'
9:         else
10:            finished = true
11:        end if
12:    end while
13:    if h(B) < h(solution) then
14:        solution = B
15:    end if
16:    − − i
17: while i > 0
18: return solution
```

of representing a realistic data pipeline, however being still tractable for validation with an exhaustive search.

3.5.1 Intermediate Results Selection Evaluation

In this section, we evaluate our approach to validate its two properties: one is convergence and second is the quality of the obtained solutions. We also compare our approach with an existing state of the art solution to show its effectiveness.

Evaluation of Convergence and Quality of the Obtained Solutions

The goal of this experiment is to evaluate convergence and quality of the obtained solutions in Algorithm 1. For the sake of experiments, we assign update frequency to each table of TPC-H as shown in Table 3.2. We assume that *supplier* and *nation* tables never update and hence, they have 0 update frequency. Further, *part* and *customer* tables do not update very often and their changes can be applied every 6 hours. That is why, we assign them 4 per day update frequency. Finally, *orders* and *lineitem* tables are frequently updated and they update together whenever there is a new order. We assume

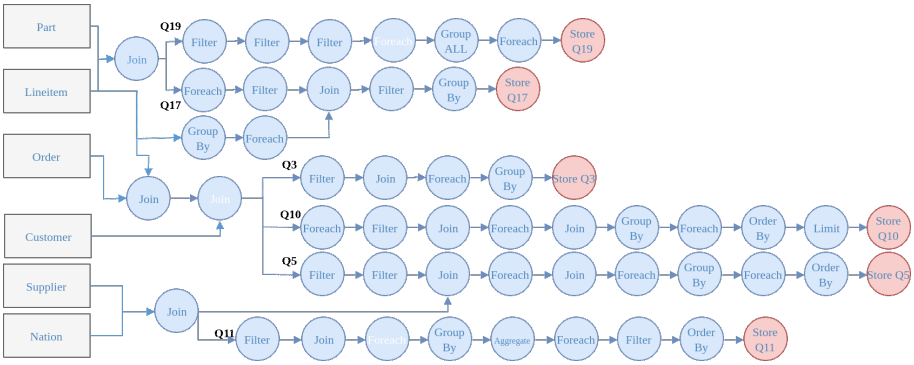


Fig. 3.3: DIF of six TPC-H queries

Table Name	UF
Part	4 / day
Lineitem	24 / day
Orders	24 / day
Customer	4 / day
Supplier	0 / day
Nation	0 / day

Table 3.2: Update Frequency (UF) of TPC-H tables

that their changes are synchronized every 1 hour and thus, their update frequencies are 24 per day.

To evaluate the convergence of solutions, we systematically executed single shots of our approach (i.e., one iteration) until the number of obtained solutions converged and no new solutions were obtained. With such information, and using the different frequencies, we can provide an estimation of the probability to obtain a solution B_K, formally defined as:

$$P(B_K) = \frac{freq(B_K)}{\sum_{j=1}^{n} freq(B_j)} \tag{3.1}$$

We aim to provide an estimation of the probability of the running Algorithm 1 with i iterations, to find a solution B_K. To this end, we introduce Equation (3.2) measuring the probability to obtain such solution at position K ($1 \leqslant K \leqslant n$) by running i iterations. It should be noted that B_1 has been confirmed to be the optimal after performing a breadth first search.

$$P(B_K, i) = P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1) + P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1) \tag{3.2}$$

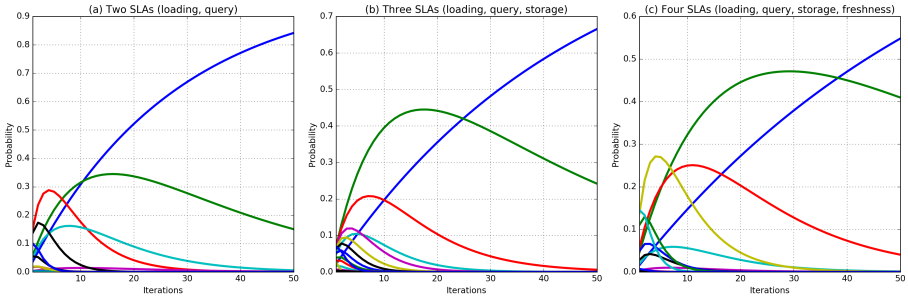


Fig. 3.4: Evolution of probabilities per number of iterations (each line corresponds to a potential solution)

The above mentioned formula is a recursive formula where the base case (i.e., $P(B_K, 1)$) corresponds to the previously defined $P(B_K)$ (i.e., the probability to find solution B_K in one iteration). The rationale behind the recursive case is that after each iteration the one with the lowest heuristic value is kept. Thus, we measure the probability that the solution at position K (i.e., B_K) remain chosen after i iterations. This is achieved by adding (a) the probability that in the previous $i-1$ iterations, B_K is chosen and in the ith iteration an equal or worst solution is chosen (i.e., $P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1)$); and (b) the probability that in the previous $i-1$ iterations a worst solution was chosen and in the ith iteration B_K is chosen (i.e., $P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1)$). Intuitively, increasing the number of iterations, those with smallest heuristics will have a higher probability to be found regardless of the initial probability being low. With such basis, we can provide an estimation of the evolution of the probability to find a solution B_K by applying the aforementioned formula.

Based on the above mentioned formula, we experiment with the trade-off between different SLAs. We perform evaluation with the following settings: (1) two SLA's metrics (i.e., load time, query time), equally weighted to 50%, (2) three SLA's metrics (i.e., load time, query time, storage space), equally weighted to 33%, and (3) four SLA's metrics (i.e., load time, query time, storage space, freshness), equally weighted to 25%. Our experiments show that the number of iterations to *converge* gradually increases with the number of considered SLAs. As shown in Figure 3.4, our approach takes 11 iterations,

Query Name	Start Time	Repeated	When to Repeat
Q3	0	Yes	6 / hour
Q5	1	No	-
Q10	2	Yes	2 / hour
Q11	3	Yes	1 / hour
Q17	0	Yes	14 / hour
Q19	2	No	-

Table 3.3: Sample workload based on TPC-H

26 iterations, and 39 iterations to converge (i.e., to be certain with a probability of 80%, that the obtained solutions will be one in the top three) for two, three, and four considered SLA's metrics, respectively. In addition, we measure the average execution time of an iteration in different settings. Our approach takes 55.45 seconds, 58.68 seconds, and 183.34 seconds for two, three and four SLA's metrics, respectively. For four SLA's metrics, it takes more time because it has larger search space, due to the conflicting SLAs and the characteristics vector (i.e., refresh frequency). As all considered scenarios follow the same convergence trend as shown in Figure 3.4, let us focus on the most complex scenario involving the trade-off of four SLAs. For four SLA's metrics, we obtained $n = 22$ different solutions across 90 executions. It can be seen that after 39 iterations, it is almost certain (i.e., >90% probability) that the obtained solutions will be one in the top three.

From such results, we conclude that the problem of finding optimal solutions by using hill-climbing indicates the issues with local optimums, known for greedy multi-objective optimization algorithms, and opens the challenge of applying more complex (i.e., parametrized solutions). However, the approach of shotgun hill-climbing, quickly yields near-optimal results after few iterations with high probability.

Comparison with an Existing Solution

Several intermediate materialization approaches for Big Data systems can be found in the literature, as discussed in Section 2.1. However, all of them focus on improving the query execution time without considering others SLAs (such as freshness). In order to show the effectiveness of our approach, we compare against the best representative solution (i.e., ReStore).

As mentioned in [14], 80% of queries are repeated in the range of minutes to hours. Thus, we created a sample workload by utilizing six TPC-H

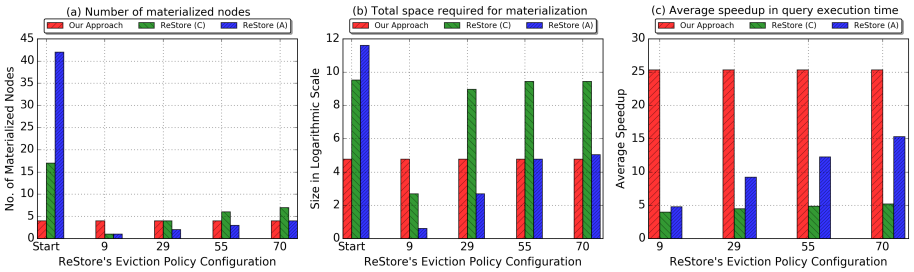


Fig. 3.5: Comparison of our approach, ReStore (C) conservative heuristics and ReStore (A) aggressive heuristics

queries, based on the aforementioned work. We set four out of six queries as repetitive and two out of six as non-repetitive queries. In addition, we set their query frequencies in the range of minutes to hours as shown in Table 3.3. Moreover, ReStore has a configuration parameter for applying its eviction policies (to delete unused materialized nodes). For experiments, we chose different configuration values such as 9, 29, 55, and 70 in minutes to compare with all the possible behaviors of ReStore.

Figure 3.5 depicts three charts to show different metrics for comparison. In Figure 3.5a, we compare *the total number of materialized nodes*, in Figure 3.5b, we show the total space required (presented using base 10 logarithm) , and in Figure 3.5c, we show the average speedup gain in the repetitive queries. When executing the queries for the first time as shown in Figure 3.5a and Figure 3.5b, ReStore materializes each operator matching the heuristics and thus, it materializes more nodes and takes more space. Whereas, our approach uses the cost model to materialize only those nodes which satisfy all the four objectives (i.e., loading time, query time, storage space, and freshness).

When we configured *9 minutes* for applying ReStore's eviction policies, it perceives only Q17 as a repetitive query because it is repeated before applying the eviction policies. Hence, it deletes all the materialized nodes except those which are used in Q17. Similarly, when we chose *29 minutes*, now it assumes that Q3 and Q17 are repetitive queries and keeps only their materialized nodes. This decision helps to reduce the occupied space but it also decreases the average speedup as shown in Figure 3.5c. Likewise, when we configured *55 minutes*, ReStore notices three queries (i.e., Q3, Q10, and Q17) as repetitive and keeps only the associated materialized nodes. As a consequence, it deletes all other materialized nodes which also reduces the average

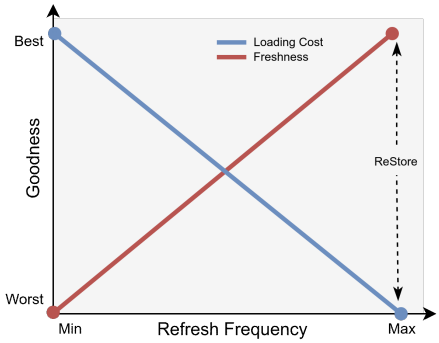


Fig. 3.6: Effect of Refresh Frequency on Loading Cost and Freshness

speedup. Finally, when we configured *70 minutes*, now it detects all possible repetitive queries and manages to keep all the required materialized nodes. However, still ReStore (C) keeps more nodes and takes more space compared to our approach as shown in Figure 3.5a and Figure 3.5b. On the other hand, ReStore (A) keeps a similar number of materialized nodes to our approach, but provides less average speedup. In general, our approach considers query frequency which helps to choose only the required materialized nodes from the start and provides better speedup than ReStore's heuristics.

In our experiments, we also evaluated our approach based on the characteristics vector (i.e. refresh frequency) to find the trade-off between loading cost and freshness. As shown in Figure 3.6, ReStore does not have support to balance them. It always deletes a materialized node as soon as any of its input source is updated. Thus, it always provides maximum freshness (only affected by the time to materialize new nodes). Consequently, it worsens the loading cost for materialized nodes, which may have highly variable input sources. Oppositely, our approach takes refresh frequency as an input and based on this, it tries to balance loading cost and freshness, by choosing the optimal value for each to-be-materialized node.

From these experiments, we conclude that our approach provides better solutions for materialization than ReStore. In addition, it can consider different SLAs as discussed in Section 3.3.3, which are not an option in the existing materialization solutions. Moreover, our example shows that we can also accept refresh frequency as a characteristic to find the balance between freshness and loading cost which is not possible in the existing materialized solutions.

3.6 Conclusions

In this chapter, we have presented an approach for the selection of intermediate results from data-intensive flows. We have built upon the general framework for materialized view selection by giving it additionally a multi-objective perspective. Moreover, we have provided a set of three cost functions with its building blocks (i.e., engine-independent statistics and engine-dependent metrics), and a representation of the approach as a state space search problem. Experimental results have showed that our approach is highly efficient in terms of performance, while providing near-optimal results.

4

Storage Format Selection for Materialized Intermediate Results

Modern big data frameworks (such as Hadoop and Spark) allow multiple users to do large-scale analysis simultaneously, by deploying Data-Intensive Flows (DIFs). These DIFs of different users share many common tasks (i.e, 50-80%), which can be materialized and reused in future executions. Materializing the output of such common tasks improves the overall processing time of DIFs and also saves computational resources. Current solutions for materialization store data on Distributed File Systems by using a fixed storage format. However, a fixed choice is not the optimal one for every situation. Specifically, different layouts (i.e., horizontal, vertical or hybrid) have a huge impact on execution, according to the access patterns of the subsequent operations. In this chapter, we present a cost-based approach that helps deciding the most appropriate storage format in every situation. A generic cost-based framework that selects the best format by considering the three main layouts is presented. Then, we use our framework to instantiate cost models for specific Hadoop storage formats (namely SequenceFile, Avro and Parquet), and test it with two standard benchmark suits. Our solution gives on average 1.33x speedup over fixed SequenceFile, 1.11x speedup over fixed Avro, 1.32x speedup over fixed Parquet, and overall, it provides 1.25x speedup.

4.1 Introduction

Data analysis plays a decisive role in today's data-oriented organizations, which produce and store large volumes of data (i.e., in the order of petabytes to zettabytes [71]). To store and process such data, organizations typically rely on the use of distributed frameworks, such as Apache Hadoop[1] and Apache Spark[2]. These frameworks are used by multiple users and the data is processed by deploying complex analytical workflows that orchestrate multiple tasks. Each task produces an output that is used as input for the subsequent tasks. The workflows may have many redundant tasks, whose output, if materialized, can be reused to improve the overall execution time. In this chapter, we refer to the materialization of the output of redundant tasks as *Intermediate Results (IRs)*.

Figure 1.2a shows two analytical workflows, orchestrating multiple tasks. They both have a common task (i.e., JOIN), which can be materialized for reuse. That is, if these workflows were to be submitted again as shown in Figure 1.2b, they would not require to recompute the JOIN, because that would have already been materialized and could be reused by both. This *IR* would help on saving computational resources and reducing the execution time. Yet, note that *IRs* are different from intermediate results, produced by different tasks in the same workflow. For instance, in Spark, the intermediate results are always stored in memory and discarded afterwards. Whereas, a *IR* is stored in the disk and its purpose is to be reused not only by the same workflow but also by different workflows.

The importance of *IRs* has been acknowledged in an in-depth study of seven enterprises [14], where it was shown that 80% of their different analytical workflows had redundant/common tasks. Similarly, recent studies [43, 44] from Microsoft have shown that 65% of their workflows have redundant parts. These studies imply that a proper management of *IRs* could provide benefits in terms of computational resources and execution time. Yet, a solution to this problem means answering the following questions: "*(1) which IR should be chosen*?" and "*(2) which layout should be used for its storage*?".

This thesis have already proposed an approach in Chapter 3, which helps on choosing the *IRs* that minimize the overall analytical workflow execution times. However, *IRs* are typically stored in a *Distributed File System (DFS)*,

[1] http://hadoop.apache.org
[2] https://spark.apache.org

using a single fixed layout, thus, ignoring the second question of "*which layouts should be used when persisting IRs*?". The importance of this question lies on the fact that, since DFS I/O operations are expensive, the analytical workflow execution times can be further reduced by choosing the physical storage layouts based on the operators. Obviously, a fixed storage layout can not be optimal for all types of workloads. Indeed, [5] shows the importance of storing data according to their access pattern and that single fixed layouts are not good for all types of workloads. Similarly, [25, 41] also focus on the importance of storing data according to their access patterns and highlight the effect of different storage layouts on different workloads[3]. Nevertheless, no current solution is able to choose the layout of *IRs* automatically.

In this chapter, we present a cost-based approach to address the second question and find the most appropriate storage layout for *IRs*. However, since a cost model requires statistical information about the data and the analytical workflows in order to make a decision, we also propose the use of a rule-based approach for cold-start. Therefore, we first apply rules for choosing storage layouts, while collecting the statistical information. Once the required statistical information has been gathered, we can apply the proposed cost-model.

Our contributions are as follows:

- We present a generic I/O model for the three main layouts (i.e., horizontal, vertical, and hybrid) in big data frameworks, for estimating their read and write costs.
- We instantiate the cost model on *Hadoop Distributed File System (HDFS)*, for SequenceFile, Avro, and Parquet.
- We propose and implement a generic framework for big data systems, to store the selected *IRs* in the appropriate storage format.
- We conduct comprehensive experiments on two de-facto standard industry benchmarks for Decision Support Systems (DSSs). The results show that our approach reduces the overall workflow execution times when compared to using single fixed layouts, by providing a 1.25x average speedup.

The remainder of this chapter is organized as follows: In Section 4.2, we discuss the storage layouts and our motivation. In Section 4.3, we present

[3]http://www.svds.com/how-to-choose-a-data-format

our overall approach. In Sections 4.4, we discuss heuristic rules. In Section 4.5 and Section 4.6, we present the generic cost model and its instantiation in detail. In Section 4.7, we report on our experimental results. Finally, in Section 4.8, we conclude the chapter.

4.2 Background and Motivation

In this section, we discuss the different storage layouts available and exemplify them with their corresponding instantiation for HDFS. Moreover, we discuss existing materialized solutions and also motivate our work by illustrating the fixed layout limitations.

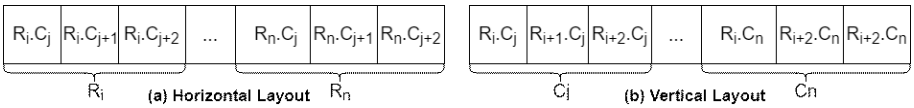


Fig. 4.1: Horizontal and vertical layouts

4.2.1 Storage layouts

There are many layouts, used in different processing frameworks, that can be divided into three categories based on how they fragment data: horizontal, vertical or hybrid. Each concrete layout has its own physical storage structure that is beneficial for a specific kind of workloads.

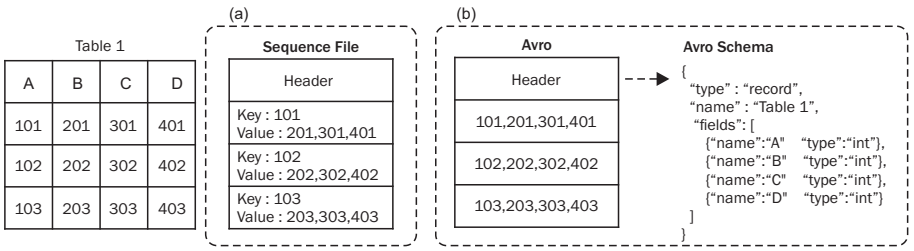


Fig. 4.2: Examples of SequenceFile and Avro layouts

Horizontal layouts

They are organized row-wise, and the attributes of each row are stored together, as shown in Figure 4.1a (where *R* represents the row and *C* represents the column of a row). For this reason, a horizontal layout especially

suits scan-based workloads (i.e., reading all rows and columns). However, if a query is just referring to a small subset of columns, this layout results in a low effective read ratio, since non-required columns will be fetched anyway. In HDFS, the horizontal layout is implemented by SequenceFile[4] and Avro[5]. SequenceFile is a special type of horizontal layout storing simple key-value data, whereas Avro explicitly splits data into columns inside every row. In other words, it embeds schema information. Figure 4.2 shows an example of a table and its corresponding format in SequenceFile (i.e., Figure 4.2a) and Avro (i.e., Figure 4.2b).

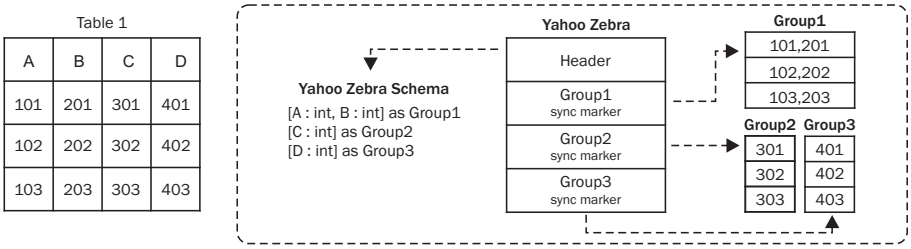


Fig. 4.3: Example of Zebra layout

Vertical layouts

They divide each row into columns, and store each column separately, which is beneficial for workloads reading just few columns. Thus, these layouts excel in projection-based workloads. Figure 4.1b sketches the physical structure of vertical layouts. Zebra[6], illustrated in Figure 4.3, is an implementation of this kind for HDFS. Zebra also allows to group columns together, but without any horizontal partition.

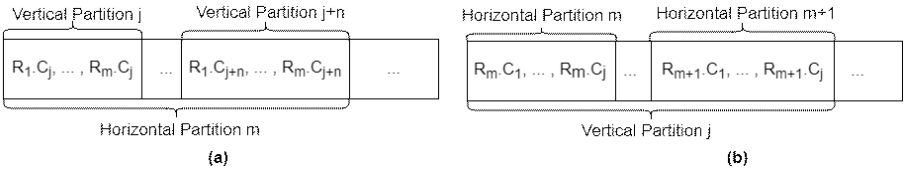


Fig. 4.4: Hybrid layouts

[4]https://wiki.apache.org/hadoop/SequenceFile
[5]https://avro.apache.org
[6]https://wiki.apache.org/pig/zebra

Hybrid layouts

They are a combination of horizontal and vertical layouts, having two alternative implementations: Either the data is divided horizontally and then vertically, like in Figure 4.4a, or vice versa, like in Figure 4.4b. Both cases are especially helpful for combinations of projection and selection operations. There are many implementations of this kind, but the most popular ones in HDFS are Optimized Row Columnar (ORC)[7] and Parquet[8], both primarily fragmenting data horizontally. Figure 4.5 exemplifies Parquet.

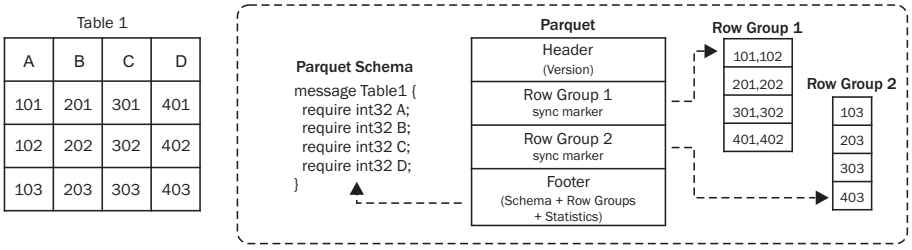


Fig. 4.5: Example of Parquet layout

4.2.2 Layout performance comparison

Ad-hoc and exploratory analysis are very popular among data analysts, helping them to understand different aspects of their business. However, it is very difficult to tune a system for such scenarios since the workload is very dynamic, and *current solutions are not considering layouts depending on workflow operations, and ignore this fact when storing IRs in the disk.*

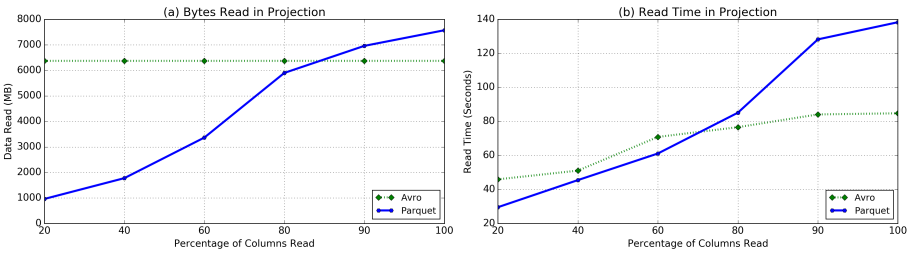


Fig. 4.6: The effect of the number of retrieved columns on different layouts

[7]https://orc.apache.org
[8]http://parquet.apache.org

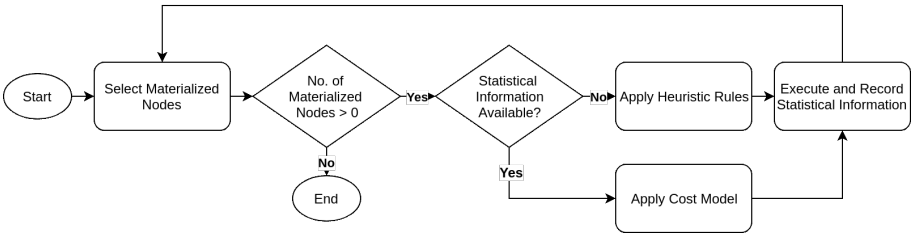


Fig. 4.7: Flowchart of our approach

To illustrate the drawback of the current static approaches let us assume the following example from TPC-H. Lets assume the join between *Lineitem* and *Part* tables is chosen as a *IR*. Figure 4.6 shows the execution time of a simple projection-based query for horizontal and hybrid layouts. It can be seen that Parquet (i.e., a hybrid layout) performs well when the total amount of data read from disk is below 75%, whereas Avro (i.e., a horizontal layout) performs better as soon as we read more than 75% of data. This is happening due to the added cost of row reconstruction for hybrid layouts. Thus, this shows that the characteristics of the query/workflow help to determine the optimal layout.

4.3 Our Approach in a Nutshell

From here on, a *Data-Intensive Flow (DIF)* is represented as a directed-acyclic graph of operations (an example can be seen in Figure 1.2). Nodes represent operations and directed edges show the dependencies between the nodes. The starting node of an edge produces the data to be consumed by the ending node (note that a node output can be consumed by several nodes). Different DIFs can have multiple common nodes, whose output when materialized is referred to as *Intermediate Results (IRs)*.

Given a DIF, Figure 4.7 illustrates the flowchart of our approach. Following the two questions introduced in Section 4.1, first, (i) it selects *IRs* using Chapter 3 and, then, (ii) for each of them, it chooses the best storage layout.

4.3.1 Storage layout selection

Since existing materialized solutions use a fixed layout for *IRs*, our approach helps them to decide the best storage layout for each chosen *IR*. If statistical information about *IRs* is available, we use the cost-based model to decide the

storage layout. If for any reason we have not enough statistical information (see the variables required by our cost model in Section 4.5) to make a decision on a given *IR*, we can still apply heuristic rules (in Section 4.4.2) to determine the storage layout. The heuristic rules choose a storage layout based only on the operation type. Obviously, it might happen that the heuristic rules do not choose the best storage layouts since they do not consider essential information to estimate the total volume of data to be read from the disk. For example, projections/selections can perform differently based on the percentage of columns/rows read. Thus, factors such as the number of columns and selectivity factors may drastically impact the operation performance depending on the storage layouts, as illustrated in Figure 4.6. These factors cannot be considered in heuristic rules, because they heavily depend on the concrete operation and data characteristics. Still, heuristic rules provide a fair first-approach to the problem with small computational requirements in scenarios where there is lack of information. Oppositely, if the required statistics are available, the cost-based approach, like the one in Section 4.5, is more accurate. Finally, the DIF is executed and the chosen *IRs* are stored with their chosen storage formats. Our approach also records/updates the needed statistical information to be used in the future.

4.4 Heuristic Rules

In this section, we present heuristic rules for choosing the storage format for a materialized node. We compare the existing data formats based on their features. In this chapter, we focus on the following set of data formats, which vary from each other in term of the implemented layout: *horizontal*: SequenceFile and Avro; *vertical*: Zebra; and *hybrid*: Parquet. Next, we present our heuristic rules based on the devised features of such data formats.

4.4.1 Comparison of Data Formats

In Table 4.1, a comparison of all the layouts and their representative formats is given. This allows to look at their features side by side. As it can be noted from the table, all formats except SequenceFile, store the schemas of data. The schema information helps during the data serialization and de-serialization phases by avoiding the need to cast the data at the application level - which is a costly operation. Moreover, the table shows that both vertical and hybrid layouts provide support for column pruning. It

means, they only read the required columns and do not perform unnecessary reads. Hybrid layouts can also push down the selection predicates into the storage layer because they store indexing information that helps in filtering the records while reading. Furthermore, since hybrid layouts store statistical information for each column, they enable easier computation of aggregates. Additionally, vertical and hybrid layouts have support for nested records which helps in storing bag, map and custom user data types. It can also be noted that hybrid layouts have additional features but they also have a significant overhead when writing and therefore when reading the same amount of data (due to the amount of metadata stored with data). Moreover, nowadays vertical layouts have been subsumed into hybrid, as they support all their features.

Features	**Horizontal**		**Vertical**	**Hybrid**	
	SequenceFile	**Avro**	**Zebra**	**ORC**	**Parquet**
Schema	No	Yes	Yes	Yes	Yes
Column Pruning	No	No	Yes	Yes	Yes
Predicate Pushdown	No	No	No	Yes	Yes
Metadata	No	No	No	Yes	Yes
Nested Records	No	No	Yes	Yes	Yes

Table 4.1: Comparison of data formats

In summary, each format provides a different set of features that will affect the overall performance when retrieving the intermediate results from the disk. Generally, hybrid layouts perform well if a subset of data is read. Alternatively, horizontal layouts perform well if all, or most of the data is read.

4.4.2 Selecting the Appropriate Format

In this section, we introduce the set of heuristic rules that choose a data format for a given IR, which derive from well-known properties of horizontal, vertical and hybrid layouts and their features presented in Table 4.1. Note that more than one rule may apply when deciding for a given to-be-materialized node. In case two contradictory rules apply (e.g., selecting Avro and Parquet), we prioritize the selection based on the data format's features. Hence, we give the highest priority to Parquet owing to the fact that Parquet has more features and a more flexible behavior. The second highest priority is assigned to Avro because it stores schema information about the data which

speeds up the reading. Finally, the lowest belongs to SequenceFile, which is only chosen for key-value data.

`rule1` : $v \rightarrow$ SequenceFile, `IF` $|v.\mathbf{O}| = 2$

`rule2` : $v \rightarrow$ Parquet, `IF` $\exists x \in successors(v)$, `WHERE` $x.\mathbf{S} \in \{AggregationOps\}$

`rule3` : $v \rightarrow$ Parquet, `IF` $\exists x \in successors(v)$, `WHERE` $x.v_{pre} \subsetneq v.\mathbf{O}$

`rule4` : $v \rightarrow$ Avro, `IF` $\forall x \in successors(v)$, `WHERE` $x.v_{pre} = v.\mathbf{O}$

`rule5` : $v \rightarrow$ Avro, `IF` $\exists x \in successors(v)$, `WHERE` $x.\mathbf{S} \in \{Join, CartesianProduct, GroupALL, Distinct\}$

`Rule1` chooses SequenceFile for the materialization of nodes that have exactly two attributes. This is an immediate application of the SequenceFile format (which stores data as key-value pairs). Otherwise, several columns would need to be combined (e.g., with a separator marker such as "-" or ";") either in the key or the value and parsed at the application level. `Rule2` chooses Parquet when performing aggregations on data. Since Parquet stores statistical information for each column, it is the most efficient when computing aggregates. Also, Parquet's hybrid layout is also the best choice when it comes to read subsets of data, or when operators apply on subsets of columns (see Table 4.1). This rationale is behind `rule3`. Oppositely, Avro is chosen when all the data is read or when the operator does not apply on a certain subset of columns. This is a consequence of Avro implementing a horizontal layout. Hence, both `rule4` (the operator affects all columns) and `rule5` (the operator requires to read the whole data without filtering) recommend Avro. It is noteworthy to mention that our rules do not consider vertical layouts as they are subsumed by hybrid layouts. Furthermore, leveraging on the presented formalization other formats can be easily added.

4.5 Cost-Based Model

The cost-based model relies on a wide range of statistical information that is summarized in Table 4.2, containing system constants, data statistics, workload statistics as well as layout variables. The system constants are generally based on [41]. We only extended the list with specific variables related to the selectivity factor and storage layouts.We assume the constants depending on the configuration of the environment (e.g., BW_{Disk}, BW_{Net}) are given and

Variable	Description
	System Constants
R	Replication factor
p	Probability of accessed replica being local
$Chunk_{Size}$	Block size in the DFS
BW_{Disk}	Disk bandwidth
BW_{Net}	Network bandwidth
$Time_{Seek}$	Disk seek time
$Time_{Disk}$	$\frac{Chunk_{Size}}{BW_{Disk}}$
$Time_{Net}$	$\frac{Chunk_{Size}}{BW_{Net}}$
	Data Statistics
$\|IR\|$	Number of Rows in IR
Row_{Size}	Average Row Size of IR
$ColValue_{Size}$	Average Column Size[1] of IR
$\#Cols$	Columns of IR
	Workload Statistics
Ref_{Cols}	Number of columns used in an operation
SF	Selectivity factor of an operation
	Layout Variables
RG_{Size}	Row group size of hybrid layouts
$Meta_{Size_{Layout}}$	Metadata size for a given layout
$Body_{Size_{Layout}}$	Size of the body of a layout
$Header_{Size_{Layout}}$	Size of the header of a layout
$Footer_{Size_{Layout}}$	Size of the footer of a layout
$Used_{Chunks_{Layout}}$	Number of chunks of a layout
$Used_{RowGroups_{Layout}}$	Number of row group of hybrid layouts
$\|RG\|$	Number of rows of a row group
$Total_{Seeks_{Layout}}$	Total number of seeks for a given layout

[1] Extra 4 bytes are considered for variable length columns

Table 4.2: Parameters of the Cost Model

the statistics are collected during the DIW execution. Moreover, it should be noted that we consider only I/O cost in our cost model, because it is the dominant factor in DIWs.

$$Total_{Size_{Layout}} = Header_{Size_{Layout}} + Body_{Size_{Layout}} + Footer_{Size_{Layout}} \tag{4.1}$$

$$Used_{Chunks_{Layout}} = \frac{Total_{Size_{Layout}}}{Chunk_{Size}} \tag{4.2}$$

$$Total_{Seeks_{Layout}} = \lceil Used_{Chunks_{Layout}} \rceil \tag{4.3}$$

Independently of the kind of layout, the driving factor of our cost model is the file size. The body, together with the header and footer compose it (Equation 4.1). From that, we can obtain the number of chunks used (Equation 4.2) and the number of disk seeks we need to reach them (Equation 4.3). The number of seeks is equal to the total number of chunks rounded up, because one seek is required for every chunk, even if it is not full. Note that modern Solid State Disks (SSDs) also have seek time (i.e., time required to turn on the right circuit), however their seek time is much less (i.e., 0.1ms) compared to hard disks (i.e., 4ms) [52, 53]. Thus, our cost model still applies and we would only need to update the system constants accordingly.

In the next subsections, we analyze the cost of data writes and reads, because they are the dominant factors in the overall execution time of DIWs. The write cost model estimates the data volume footprint of each layout as well as the cost incurred in writing it, while the read cost model estimates the cost of an operation depending on the access pattern. Regarding the latter, given the simplicity of a file system (far from that of a DBMS) only three operations are possible (namely full scan, projection, and selection).

$$W_{WriteTransfer} = \frac{Time_{Disk} + (R-1) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (R-1) \cdot Time_{Net}} \tag{4.4}$$

$$\begin{aligned} Cost_{Write_{Layout}} &= Used_{Chunks_{Layout}} \cdot W_{WriteTransfer} \\ &\quad + Total_{Seeks_{Layout}} \cdot (1 - W_{WriteTransfer}) \end{aligned} \tag{4.5}$$

4.5.1 Write cost

First of all, we have to take into consideration that distributed processing frameworks are using DFS to store data into multiple chunks. Thus, the number of chunks of a file is used to estimate the overall writing costs. Given that a chunk consists of multiple contiguous disk blocks and inside it, sequential read is guaranteed, assuming that the chunk size is smaller than a disk cylinder, the write cost can be simply computed as the number of chunks plus the seek cost to locate the position of each. Nevertheless, since our cost model is thought for distributed processing frameworks, we further need to consider the replication factor R used for fault-tolerance, and therefore the network costs for writing R copies needs to be taken into account. We assume that the replication procedure is sequential (as it is in HDFS) and the multiple copies

are written one after another. Equation 4.4 gives the weight of transferring a chunk by considering the network and the disk write against the seek costs. Finally, Equation 4.5 shows the total write cost taking both seek and transfer weights into account.

In the following, we present the write cost for each horizontal, vertical and hybrid layouts.

Horizontal layouts

They store data row-wise into the body section. Oppositely, metadata containing information such as schema and version, is written into the header and footer sections. Nevertheless, in some implementations, additional metadata is also written in the body with every row, for example, metadata used to separate each row or each column (i.e., its size is not constant and depends on the number of columns).

$$Body_{Size_{Horizontal}} = Meta_{Size_{HBody}} + |IR| \cdot (Meta_{Size_{HRow}} + Row_{Size}) \tag{4.6}$$

Equation 4.6 estimates the size of the body by multiplying the average row size and metadata (i.e., $Size(Meta_{HRow})$) by the total number of rows, plus other metadata (i.e., $Size(Meta_{HBody})$) we may find in the body section.

$$OneCol_{Size} = ColValue_{Size} \cdot |IR| \tag{4.7}$$

$$Body_{Size_{Vertical}} = Meta_{Size_{VBody}} + \#Cols \cdot (Meta_{Size_{VCol}} + OneCol_{Size}) \tag{4.8}$$

Vertical layouts

They store each column independently (i.e., values of a column, which share the same data type, are stored consecutively) using a separator (i.e., $Meta_{Size_{VBody}}$) of fixed size between columns. Equation 4.7 provides the estimation of the individual column size, which is used in Equation 4.8 to determine the overall size of the body by multiplying the size of one column by the total number of columns.

$$Used_{RowGroups_{Hybrid}} = \frac{\#Cols \cdot (Meta_{Size_{YCol}} + |IR| \cdot ColValue_{Size})}{RG_{Size}} \tag{4.9}$$

$$Meta_{Size_{Hybrid}} = \lceil Used_{RowGroups_{Hybrid}} \rceil \cdot Meta_{Size_{YRowGroup}} \tag{4.10}$$

$$Body_{Size_{Hybrid}} = Used_{RowGroups_{Hybrid}} \cdot RG_{Size} + Meta_{Size_{Hybrid}} \tag{4.11}$$

Hybrid layouts

They are a combination of horizontal and vertical layouts. They divide rows into horizontal partitions known as row groups and each row in one row group is further divided into vertical partitions storing each column separately, and inserting metadata (i.e., $Meta_{Size_{YCol}}$) between them. Additionally, they also store metadata (i.e., $Meta_{Size_{YRowGroup}}$) for every row group. Thus, the total size of the body depends on the number of row groups being used, which can be estimated as in Equation 4.9 and the size of metadata of row groups is estimated in Equation 4.10. Notice that the metadata of the row group is stored irrespectively of it being completely full, so this must be rounded up. Furthermore, Equation 4.11 obtains the size of the body by multiplying the number of row groups by the size of a row group and by adding the total size of metadata.

4.5.2 Read cost

This section presents the read cost model for scan, projection and selection operations. All DIW operations in current massively distributed processing environments use a full scan access pattern on the DFS, except projection and selection operations that are specifically supported natively in some storage layouts. Thus, we consider them separately in the following.

$$Scan_{Size_{Layout}} = Total_{Size_{Layout}} + (Used_{Chunks_{Layout}} \cdot Meta_{Size_{Layout}}) \tag{4.12}$$

Scan

It reads all stored data from the disk, irrespective of the layout being used. Relatively, the metadata (such as schema, statistics, etc.) stored inside header

or footer sections, reads separately in each task. The reason is that the distributed processing engines (such as Hadoop and Spark) create a separate process for each task with its own memory. This memory is not accessible to other tasks and hence, forces to read all metadata in each task separately, and consequently, increases the reading size. The number of tasks is equal to the number of used chunks. Equation 4.12 estimates the scan size, which can be used further to estimate the scan cost.

The scan cost purely depends on the number of used chunks to be read. Assuming the block is the transfer unit between disk and memory, there are three factors impacting the cost: the average seek time needed to locate a disk block cylinder, the rotation time to move the disk head over the cylinder to reach the block, and the transfer time to bring data in the block from disk into memory. Nevertheless, despite every chunk consists of multiple blocks on disk, it should be noted that DFS typically guarantee that all disk blocks are contiguous within one disk cylinder, under the assumption that the chunk size does not go beyond the cylinder size. This is why we do not need to consider seek time for all the disk blocks. Instead, we only consider seek time once for every chunk. Also, as confirmed in our experiments, the rotation time is negligible, because modern hardware and operating systems implement very effective pre-fetching techniques. Furthermore, our cost model is also applicable to SSDs. Since SSDs have very small seek time and high I/O speed, the corresponding system constants would simply be replaced respectively. For the rest, since the basic unit of our cost model is defined in terms of bytes, all the estimations will remain the same.

$$W_{ReadTransfer} = \frac{Time_{Disk} + (1-p) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (1-p) \cdot Time_{Net}} \tag{4.13}$$

$$Used_{Chunks_{Scan_{Layout}}} = \frac{Scan_{size_{Layout}}}{Chunk_{Size}} \tag{4.14}$$

$$\begin{aligned} Cost_{Scan_{Layout}} &= Used_{Chunks_{Scan_{Layout}}} \cdot W_{ReadTransfer} \\ &\quad + Total_{Seeks_{Layout}} \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.15}$$

On the other hand, we have to take under consideration that in a distributed data processing framework data can be accessed remotely. Consequently, we introduce a probability p to indicate the likelihood of chunks being accessed locally (i.e., data shipping through the network is not needed

to reach the operation executor). This is used to estimate the weight of transferring the chunk data compared to the corresponding seek time using Equation 4.13. Then, Equation 4.14 estimated the total number of read chunks and Equation 4.15 provides the scan cost taking both the seek and the transfer cost into account with the corresponding weights.

Projection

It helps in fetching only some columns from disk (skipping others) to save some I/Os. Its cost depends on the support provided by each layout.

Horizontal layouts. They do not provide specific support for projection operation, but actually use a full scan to bring all the data into memory and only afterwards discard the unnecessary columns. Therefore, its cost is exactly the same as that of scan (i.e., Equation 4.15).

$$Project_{Size_{Vertical}} = Header_{Size_{Vertical}} + Footer_{Size_{Vertical}} + OneCol_{Size} \cdot Ref_{Cols} \quad (4.16)$$

$$\begin{aligned} Cost_{Project_{Vertical}} &= Used_{Chunks_{Project_{Vertical}}} \cdot W_{ReadTransfer} \\ &\quad + Ref_{Cols} \cdot Total_{Seeks_{OneCol}} \\ &\quad \cdot (1 - W_{ReadTransfer}) \end{aligned} \quad (4.17)$$

Vertical layouts. They do support projections. Their cost depends on the size retrieved data, which is exactly that of the referred columns and the metadata in the header and footer sections, as in Equation 4.16. The seek time depends on the number of retrieved columns (that might not be consecutively stored in disk), and their size. Equation 4.17 combines both components considering the weight of a read transfer as defined in Equation 4.13.

$$|RG| = \left\lfloor \frac{|IR|}{Used_{RowGroups_{Hybrid}}} \right\rfloor \tag{4.18}$$

$$RefCols_{Size} = Ref_{Cols} \cdot (Meta_{Size_{YCol}} + |RG| \cdot ColValue_{Size}) \tag{4.19}$$

$$\begin{aligned} Project_{Size_{Hybrid}} &= Header_{Size_{Hybrid}} + Footer_{Size_{Hybrid}} \\ &\quad + (RefCols_{Size} + Meta_{Size_{YRG}}) \\ &\quad \cdot Used_{RowGroups_{Hybrid}} + (Used_{Chunks_{Hybrid}} \cdot Meta_{Size_{Hybrid}}) \end{aligned} \tag{4.20}$$

$$\begin{aligned} Cost_{Project_{Hybrid}} &= Used_{Chunks_{Project_{Hybrid}}} \cdot W_{ReadTransfer} \\ &\quad + TotalSeek_{Hybrid} \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.21}$$

Hybrid layouts. They also natively support projection, and similarly to vertical layouts, we have to calculate its size to estimate the cost. However, hybrid layouts store data into multiple row groups. Therefore, we first need the row group size to estimate the projection size. As each row group contains a subset of rows, we estimate it as in Equation 4.18. Furthermore, Equation 4.19 gives the size of the columns used in the operation inside a group, which is then used in Equation 4.20 to estimate the overall projection size. Similar to the scan cost, hybrid layout also reads metadata separately for projection in each task, which we consider in the projection size. Hybrid layouts also have a seek cost to be considered, which depends on the number of row groups needed by the overall size of the file (not only of the result of the projection). Similar to previous cases, we can estimate the projection cost of hybrid layouts by appropriately weighting the transfer and seek times as in Equation 4.21.

Selection

It helps in fetching only some rows from disk (skipping others) to save some I/Os. As for projection, its cost depends on the support provided by each layout.

Horizontal and vertical layouts. They do not natively support this operation. They perform scan to bring all the data into memory and then filter them out based on the given predicate. Thus, their selection cost is the same as that of scan.

$$P_{RGSelected} = 1 - (1 - SF)^{|RG|} \tag{4.22}$$

$$RowsSelected_{Size} = \left\lceil \frac{SF \cdot |IR|}{|RG|} \right\rceil \cdot \#Cols \cdot (Meta_{Size_{YCol}} + |RG| \cdot ColValue_{Size}) \tag{4.23}$$

$$Used_{RG_{Select_{Hybrid}}} = \begin{cases} Used_{RG_{Hybrid}} \cdot P_{RGSelected} & \text{if Unsorted} \\ \left\lceil \frac{RowsSelected_{Size}}{RG_{Size}} \right\rceil & \text{if Sorted} \end{cases} \tag{4.24}$$

$$\begin{aligned} Select_{Size_{Hybrid}} &= Header_{Size_{Hybrid}} + Footer_{Size_{Hybrid}} \\ &+ (Used_{RG_{Select_{Hybrid}}} \cdot RG_{Size}) \\ &+ (Used_{Chunks_{Hybrid}} \cdot Meta_{Size_{Hybrid}}) \end{aligned} \tag{4.25}$$

$$\begin{aligned} Cost_{Select_{Hybrid}} &= Used_{Chunks_{Select_{Hybrid}}} \cdot W_{ReadTransfer} \\ &+ TotalSeeks_{Select_{Hybrid}} \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.26}$$

Hybrid layouts. They keep statistical information about data values in every column for every row group (typically, inside the header or footer sections). This helps in skipping some of the row groups that do not satisfy the predicate. Thus, the number of row groups to be read depends on the filtering condition and the sorting order of the column on which the selection is applied.

For unsorted columns, we can use the probability as in Equation 4.22 (borrowed from bitmap indexes [12]) to estimate the likelihood of any data in a row group satisfying the condition (i.e., a row group being fetched). In Equation 4.24, this probability is used to obtain the expected number of retrieved row groups. However, if a column is sorted, then we are using the *Selectivity Factor (SF)* to estimate how much data is going to be read using Equation 4.23, which is later used in Equation 4.24 to calculate the fetched row groups for sorted columns (notice that all data fulfilling the condition is stored together if they are sorted on that column). Having the number of selected row groups, Equation 4.25 determines the size of a selection by

adding up the total size of fetched row groups, metadata, header, and footer sections. As previously discussed about multiple reads of metadata in each task, we also consider this factor in the estimation of selection size.

Finally, this selection size can be used to estimate the total number of chunks and seeks as in Equations 4.2 and 4.3, which are then weighted as in Equation 4.26 to estimate the total selection cost.

4.6 Instantiating the Cost Model

This section shows the file sizes for the three considered HDFS file formats, together with the system variables with their values according to our testbed. Table 4.3 lists all the system variables. They are divided in three categories. First category has the variables related to disk which are important to calculate the reading and writing cost. Additionally, second category has variables for network to calculate the transfer cost, since Hadoop writes multiple copy of data for fault tolerance purpose and this involves writing to other nodes. For this writing, it needs to transfer data, and it is important in calculating the overall write cost. Final category lists the variables related to the configuration of our Hadoop cluster.

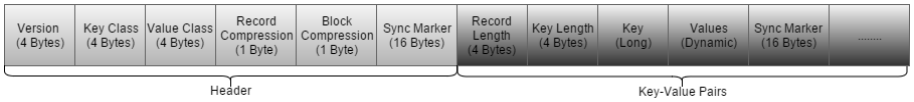


Fig. 4.8: Physical file format of SequenceFile

4.6.1 SequenceFile (SeqFile) format

SeqFile[9] is introduced in 2009 to improve the performance of MapReduce framework. It is used to store the temporary output of map phases as compressed to reduce I/Os. Moreover, it is also splittable which is ideal for processing in parallel. It considers a special type of horizontal layout, which stores data in the form of key-value pairs. Figure 4.8 shows its structure and Table 4.4 shows the specific variables of SeqFile with their values.

[9]https://wiki.apache.org/hadoop/SequenceFile

Variables for Disk		
BW_{Disk}	Disk bandwidth	1.30×10^8 bytes/second
$Block_{Size}$	Disk block size	8.00×10^3 bytes
$Time_{Seek}$	Disk random seek time	5.00×10^{-3} seconds
$Time_{Rotation}$	Disk rotation time	4.17×10^{-6} seconds
Variables for Network		
BW_{Net}	Network bandwidth	1.25×10^8 bytes/second
Variables for Hadoop		
$Chunk_{Size}$	HDFS block size	1.28×10^8 bytes
$Buffer_{Size}$	Buffer size	6.40×10^4 bytes
R	Replication factor	3
p	Probability of accessed replica being local [41]	0.97

Table 4.3: System variables with their values according to our testbed

$$Row_{Size_{SeqFile}} = RecordLength_{Size} + KeyLength_{Size} \tag{4.27}$$

$$+ ColValue_{Size} \cdot \#Cols \tag{4.28}$$

$$+ Meta_{Size_{SCol}} \cdot (\#Cols - 2)$$

$$TotalRows_{Size_{SeqFile}} = Row_{Size_{SeqFile}} \cdot |IR| \tag{4.29}$$

$$Meta_{Size_{SBody}} = \left\lceil \frac{TotalRows_{Size_{SeqFile}}}{SyncBlock_{Size}} \right\rceil \cdot SyncMarker_{Size} \tag{4.30}$$

$$Body_{Size_{SeqFile}} = TotalRows_{Size_{SeqFile}} + Meta_{Size_{SBody}} \tag{4.31}$$

To instantiate from our generic cost model, we need to estimate the sizes of header, body and footer sections. The header section of SeqFile has a fixed size, so we define it as a constant. To estimate body size, we need to calculate row and metadata sizes. SeqFile divides each row into a key-value pair and stores one column into the key, and the remaining columns into the value by using a user-defined separator. Thus, it has two types of metadata: one is

Variables for SeqFile		
$Header_{Size_{SeqFile}}$	Header size of SeqFile	30
$RecordLength_{Size}$	Fixed field	4
$KeyLength_{Size}$	Number of bytes for key	4
$Meta_{Size_{SCol}}$	Number of bytes for user-defined separator per column	1
$SyncMarker_{Size}$	Size of sync marker	16
$SyncBlock_{Size}$	Number of bytes between sync markers	2,000
$Footer_{Size_{SeqFile}}$	Footer size	0

Table 4.4: Sizes of SeqFile according to our testbed

used to separate values and another to make blocks for parallel processing. Then, the size of a row is compound of some fields of fixed size (i.e., record and key lengths) together with the corresponding key-value pair as shown in Figure 4.8, containing all user columns (notice that we need two less user-defined separators than columns, because the key is managed by the file format itself). Equation 4.27 is estimating this size (i.e., a row for SeqFile), which is later used in Equation 4.29 to estimate the size of all key-value pairs. Equation 4.30 calculates the overhead of block-related metadata (i.e., sync markers), which SeqFile introduces at fixed intervals. Finally, Equation 4.31 simply adds the size of key-value pairs and metadata, which allows in turn to obtain the total size of SeqFile using Equation 4.1 with an empty footer section.

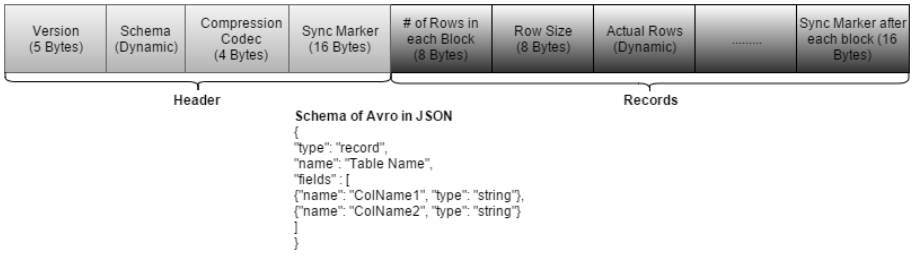


Fig. 4.9: Physical file format of Avro

Variables for Avro		
$Version_{Size}$	Version of Avro	5
$Codec_{Size}$	Compression codec	4
$SyncMarker_{Size}$	Size of sync marker	16
$ColSchema_{Size}$	Size of schema information per column	~30 bytes
$B_{Size_{Avro}}$	Block size of Avro	4,000
$Meta_{Size_{ARow}}$	Meta information for each row	8
$Meta_{Size_{ABlock}}$	Meta information for each block	8
$Footer_{Size_{Avro}}$	Footer size	0

Table 4.5: Sizes of Avro according to our testbed

4.6.2 Avro format

Apache Avro[10] is a language-neutral data serialization system. It means Avro can be written in one language and can be read in another language without changing the code. This support is provided by the schema information which Avro stores as a meta information. Moreover, it is also compressible and splitable. It is a horizontal layout and Figure 4.9 sketches its physical structure. Moreover, there are specific variables for Avro which are given in Table 4.5. The data schema is stored in a header section of variable length. Similarly, the size of body is also variable and it depends on the number of rows in an IR.

$$Header_{Size_{Avro}} = Version_{Size} + \#Cols \cdot ColSchema_{Size} + Codec_{Size} + SyncMarker_{Size} \tag{4.32}$$

$$TotalRows_{Size_{Avro}} = (Row_{Size} + Meta_{Size_{ARow}}) \cdot |IR| \tag{4.33}$$

$$Meta_{Size_{ABody}} = (Meta_{Size_{ABlock}} + SyncMarker_{Size}) \cdot \left\lceil \frac{TotalRows_{Size_{Avro}}}{B_{Size_{Avro}}} \right\rceil \tag{4.34}$$

$$Body_{Size_{Avro}} = TotalRows_{Size_{Avro}} + Meta_{Size_{ABody}} \tag{4.35}$$

[10] https://www.tutorialspoint.com/avro/avro_tutorial.pdf

Header section of Avro contains meta information corresponding to the schema of the data in the form a JSON. Given that the size of the schema is orders of magnitude smaller that data, we estimate it as a constant per column. Considering also the version and codec information, the overall header size is calculated by Equation 4.32. Following the horizontal layout, Avro adds metadata to each row, which is considered in Equation 4.33 to estimate the size of a row. Moreover, it also adds extra metadata in the body for every block. Thus, Equation 4.34 is calculating the total size of metadata by multiplying the number of blocks by the size of sync marker and that of counter for the number of rows in the block. Finally, Equation 4.35 is used to calculate the body size, which allows in turn to obtain the total size of Avro using Equation 4.1 with an empty footer section.

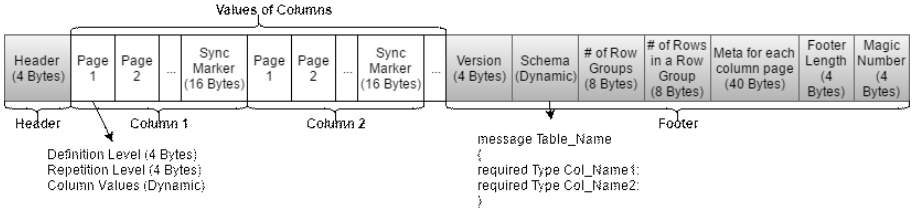


Fig. 4.10: Physical file format of Parquet

4.6.3 Parquet format

Apache Parquet[11] is introduced in 2013 to provide hybrid layout support for Hadoop echosystem. It divides data horizontally into row groups, whereas each row group is further divided vertically to store columns separately, as sketched in Figure 4.10. Additionally, it also divides each vertical partition into multiple pages. Moreover, it also stores the schema and statistical information about the data as meta information in the footer section. All variables specific to Parquet are listed in Table 4.6.

[11] http://parquet.apache.org

$$Used_{Pages_{RowGroup_{Parquet}}} = (ColValue_{Size} \cdot |RG| + SyncMarker_{Size}) \cdot \frac{\#Cols}{Page_{Size}} \tag{4.36}$$

$$Body_{Size_{Parquet}} = (((DefinitionLevel_{Size} + RepetitionLevel_{Size} + Page_{Size}) \cdot Used_{Pages_{RowGroup_{Parquet}}}) + RowCounter_{Size} + SyncMarker_{Size}) \cdot Used_{RowGroup_{Parquet}} \tag{4.37}$$

$$Footer_{Size_{Parquet}} = Version_{Size} + ColSchema_{Size} \cdot \#Cols + MagicNumber_{Size} + FooterLength_{Size} + Used_{RowGroup_{Parquet}} \cdot Meta_{Size_{PCol}} \cdot (1 + Used_{Pages_{RowGroup_{Parquet}}}) \tag{4.38}$$

The header section of Parquet has a fixed size, as stated in Table 4.6. To estimate the body size, we first need to estimate the total number of row groups (i.e., Equation 4.9) and the total rows per row group (i.e., Equation 4.18). Moreover, we need to be aware that Parquet stores every individual column by dividing it into multiple pages, whose number which is estimated by Equation 4.36 per row group. Next, we are calculating the body size of Parquet using Equation 4.37, by considering metadata for each page (namely definition level and repetition level), and for every row group (namely counter of rows per row group and sync marker).

Finally, we calculate the footer size by approximating the size the of the schema, sketched in Figure 4.10, by a constant amount of bytes per column. Moreover, Parquet also stores statistical information about columns in the Footer section for both row groups and data pages. Equation 4.38 uses all these values together to calculate overall size of footer. Then, total size of Parquet is obtained by adding the header, body and footer sections, as defined in Equation 4.1.

4.7 Experiments

In this section, we evaluate our approach and show the accuracy of our cost model for estimating the file sizes and the cost of scan, projection and selec-

Variables for Parquet		
$Header_{Size_{Parquet}}$	Size of header	4
$DefinitionLevel_{Size}$	Size of definition level	4
$RepetitionLevel_{Size}$	Size of repetition level	4
$RowCounter_{Size}$	Size of number of rows	8
$SyncMarker_{Size}$	Size of sync marker	16
$Version_{Size}$	Version in footer	4
$ColSchema_{Size}$	Size of schema information per column	~30 bytes
$Meta_{Size_{PCol}}$	Size of columns meta data for storing statistical information	40
$MagicNumber_{Size}$	Magic number in footer	4
$FooterLength_{Size}$	Footer length in footer	4
RG_{Size}	Layout row group size	128MB
$Page_{Size}$	Layout page size	1MB

Table 4.6: Sizes of Parquet according to our testbed

tion for different storage formats. We choose representative storage formats from Apache Hadoop, the most popular distributed processing framework, because it is used in 59% of the enterprises to process big data, as shown in a survey from Cloudera [6]. In order to generate realistic data-intensive workflows, we rely on standard industry benchmarks. We utilize TPC-H and TPC-DS for evaluating our approach. TPC-H provides OLAP-like queries that are typically characterized by a low selectivity factor. To properly assess our approach, a broader range of analytical queries (i.e., typical reporting and data mining queries) are required. For this reason, we also leverage on TPC-DS for a more representative set of experiments.

Prior to conduct our experiments, we first instantiate our cost-model for Apache Hadoop. In HDFS, we can find several storage formats that follow the storage layouts discussed. Among them, we choose the most representative ones to show the effectiveness of our approach: SequenceFile (SeqFile) and Avro for horizontal layouts and Parquet for hybrid layouts. Section 4.6 contains all the details about the instantiation of these formats, including the file format size calculation and the required system variables. Note that, despite being included in Section 4.5 for the sake of completeness, we did not

include any vertical layout, since those available for HDFS ended up being subsumed by hybrid ones and deprecated with time. Additionally, it is not possible to use hybrid layouts to mimic the behavior of vertical layouts, because the current implementations of hybrid layouts define a maximum limit on the size of row groups (i.e., 2GB), and do not allow creating different column groups (grouping different columns inside one vertical group and storing them row-wise). Finally, note also that for a fairer comparison, we are not considering encoding, which is available only in Parquet.

4.7.1 Experimental setup

Our experiments are performed on a 16-machines cluster[12]. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory and 1TB SATA-3 of hard disk and runs Hadoop 2.6.2 and Pig 0.16.0 on Ubuntu 14.04 (64 bit). We have dedicated one machine for the HDFS name node and the remaining 15 machines for data nodes. We are using Apache Parquet 1.9.0, Avro 1.7.0 and elephant-bird 4.9[13] for SeqFile.

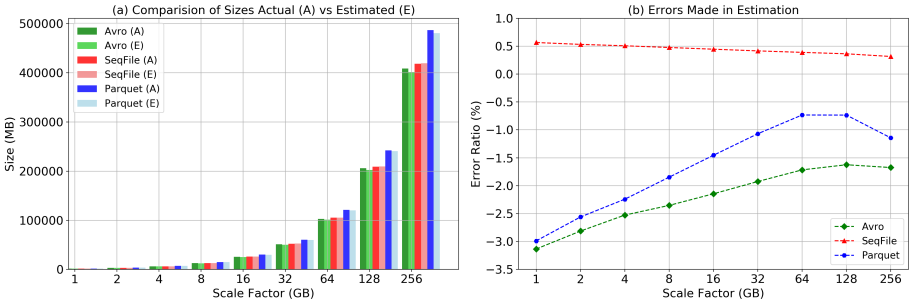


Fig. 4.11: Validating the size estimation

4.7.2 Validation of file size estimations

In this section, we are validating the accuracy of our size estimation by creating a synthetic IR (i.e., join of Lineitem and Part tables of TPC-H), and compare the actual size with the estimated one for each operation, namely scan, projection, and selection. We have chosen JOIN over other operations, because typically it is a computationally expensive operation and very common in modern DIFs. Figure 4.11 shows the results for scan operation on

[12]http://www.ac.upc.edu/serveis-tic/altas-prestaciones
[13]https://github.com/twitter/elephant-bird

different scale factors. Figure 4.11a shows the results for the size, while Figure 4.11b shows the corresponding error rate for each studied format. We see that Avro and Parquet are slightly underestimated (up to -3% error), while SeqFile is slightly overestimated (up to 0.5%).

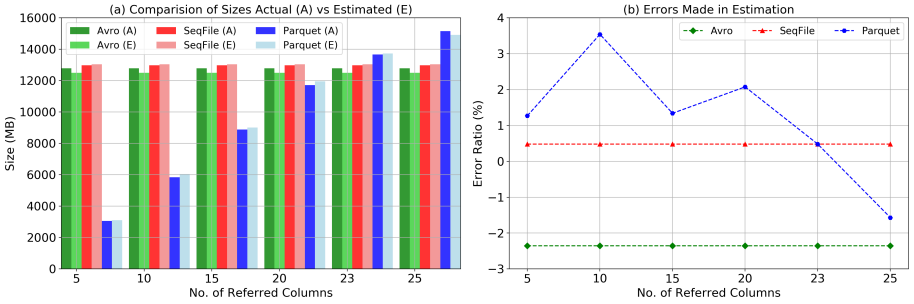


Fig. 4.12: Validating the projection cost model

Similarly, Figure 4.12 shows the results for validating our file size estimation after a projection. To do so, we read different number of columns, ranging from 5 to 25, by executing 100 different runs, randomly selecting different columns on each run, over 8GB and took the average of all runs. Figure 4.12a compares the actual and estimated size, and Figure 4.12b shows the percentage of error. SeqFile and Avro perform a scan for projection and their errors are the same as of the scan. However, Parquet has errors between +4% to -2%, whose variance is due to variable column sizes (e.g., column with string data type), whereas we use average column size for all columns.

Finally, Figure 4.13 validates the file size after a selection operation. For this experiment, we generate different selectivity factors. Also, since the sorting order of the filter column affects the reading, we are validating our results for both sorted and unsorted columns. Moreover, we repeated our experiments 100 times over 8GB by randomly choosing different search values and took the average of all executions. Figures 4.13a and 4.13b show the results of size estimations and errors in estimations, respectively, for unsorted columns. Observe that our cost model slightly underestimates the sizes (i.e., up to -4%). Moreover, the errors are more irregular when having small selectivity factors. This is due to the fact that when searching for few values, it is more difficult to find the exact row groups that contain those values. Figures 4.13c and 4.13d show the results for sorted columns. Here our cost model for Parquet has errors in the range between +2% to -4% for the same reason discussed for unsorted columns.

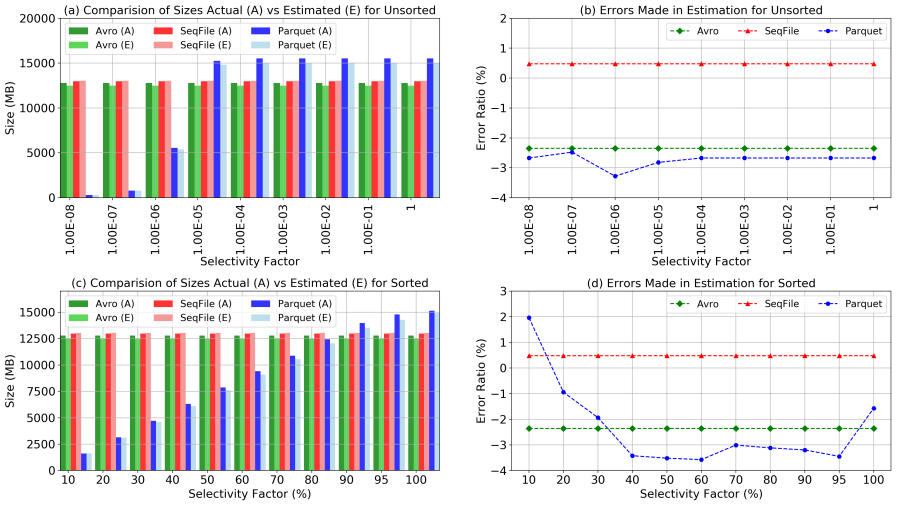


Fig. 4.13: Validating the selection cost model

All in all, the errors obtained in all our tests are rather small and consistent. Most importantly, we show next that these errors do not affect our prediction to choose the right storage format in all the experiments we conducted, since the estimated values still preserve the partial order among the actual values. Moreover, most of the deviations appear due to the hidden metadata contained in the implementations of these layouts. Thus, it is not possible to improve in a generic way the cost model for all implementations.

4.7.3 Validation of file format choice

We have selected 6 out of 16 queries for TPC-H, and 16 out of 99 queries for TPC-DS based on two main criteria: the selectivity factor (from 1% to 92%), and the number of referred columns (from 3 to 66). All the selected queries were grouped based on these criteria. Then, representative queries are chosen with the goal of covering all the possible scenarios.

In order to create a complex DIF, we used Quarry [48] to combine all TPC-H and TPC-DS queries into one integrated DIF as shown in Figure 3.3, and Figure 4.14, respectively. To perform realistic experiments, we generate data with scale factors ranging from 1GB to 256GB. In our experiments, nine nodes are selected to be materialized. We choose two metrics to analyze our approach, namely write cost (Section 4.5.1) and read cost (Section 4.5.2) for each materialized node. However, due to limitations in the native mea-

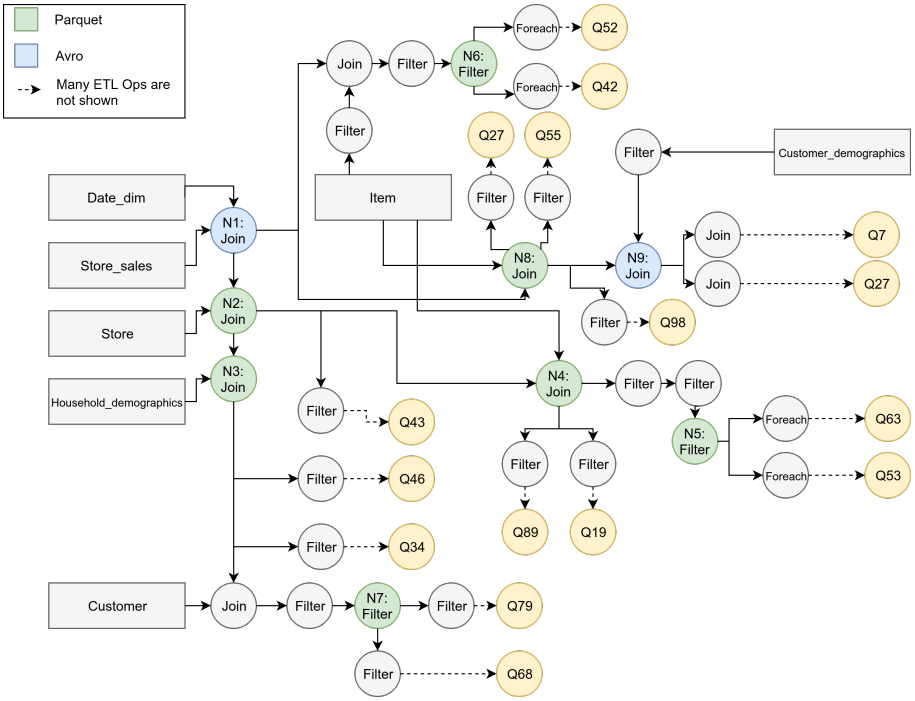


Fig. 4.14: DIF of 16 TPC-DS queries

surement of Hadoop performance, the charts corresponding to the read cost include also the execution cost of the first operation right after reading the *IR*, since their costs cannot be decoupled.

Rule-based approach

Table 4.7 shows all nine nodes that have been materialized, together with their outgoing operators and storage formats decided by applying the heuristic rules. The rule-based column shows the choices made by our rule-based approach. The rationale behind these choices is as follows. Avro is chosen for N1 and N9, because the outgoing operators are joins, that use a scan access pattern, where Avro excels, as discussed in Section 4.2.1. For all other nodes, the rule-based approach is choosing Parquet. For Nodes N5 and N6, the outgoing edges contain FOREACH operations, where Parquet benefits from independent column storage. Nodes N4, N7 and N8 have FILTER operations in their outgoing edges, where Parquet can benefit from its native predicate push-down. Both FOREACH and FILTER operations only require a subset

Node	Outgoing Operators	Rule-based	Trojan	Cost-based	Real Best Choice
N1	JOIN, JOIN	Avro	Avro	Avro	Avro
N2	JOIN, JOIN, FILTER (SF: 0.19)	Parquet	Avro	Avro	Avro
N3	JOIN, FILTER (SF: 0.59) , FILTER (SF: 0.01)	Parquet	Avro	Avro	Avro
N4	FILTER (SF: 0.03), FILTER (SF: 0.2), FILTER (SF: 0.19)	Parquet	Avro	Avro	Avro
N5	FOREACH (Ref Cols: 3), FOREACH (Ref Cols: 3)	Parquet	Parquet	Parquet	Parquet
N6	FOREACH (Ref Cols: 15), FOREACH (Ref Cols: 4)	Parquet	Avro	Parquet	Parquet
N7	FILTER (SF: 0.13), FILTER (SF: 0.92)	Parquet	Avro	Avro	Avro
N8	JOIN, FILTER (SF: 0.19), FILTER (SF: 0.03), FILTER (SF: 0.01)	Parquet	Avro	Avro	Avro
N9	JOIN, JOIN	Avro	Avro	Avro	Avro

*Projection is implemented as FOREACH in Apache PIG

Table 4.7: Materialized nodes with the statistics about their operations and chosen storage formats

of data, and Parquet excels whenever a subset of data is read. Finally, nodes N2 and N3 have JOIN and FILTER as outgoing edges, and there would be different options to choose. However, for our rule-based approach Parquet is chosen, since, in case of several options available, it chooses the richest format providing more features.

Trojan cost-model

The Trojan cost-model [41] provides cost formulas only for reading operations (i.e., scan). Since there is no estimation for writing operations and therefore, in our comparison we consider only the reading cost. For reading, Trojan considers the number of referred columns and always assumes a 100% selectivity factor. We executed the Trojan cost model on top of our scenario and the results obtained were as follows. Trojan selects Avro for all the nodes except N5. In N5, the number of referred columns are considered and there-

fore Parquet is predicted as the best choice. However, Trojan fails to predict N6 correctly. This is due to the fact that it assumes a high random reading cost (which in reality is not true, see Figure 4.16). For the other nodes, Trojan predicts correctly, however this is due to the fact that Parquet cannot filter when the selectivity factor is high. Thus, the high selectivity factor always favors Avro. In a scenario with a low selectivity factor, Trojan would fail to correctly predict the storage layouts.

Cost-based approach

Note that Table 4.7 also shows some relevant collected statistics, such as the selectivity factor (SF) and the number of referred columns (Ref Cols), of the outgoing operators. It also shows the choices made by our cost-based approach. Here, we have not reported the estimated cost of our cost model, but we validated its choices with the actual executions, which confirms the accuracy of its predictions. Moreover, we have divided these nine nodes into three different color groups which are green, grey, and white. Green and grey groups contain nodes for which our rule-based approach works fine. Whereas, white group contains all the nodes for which our rule-based approach fails.

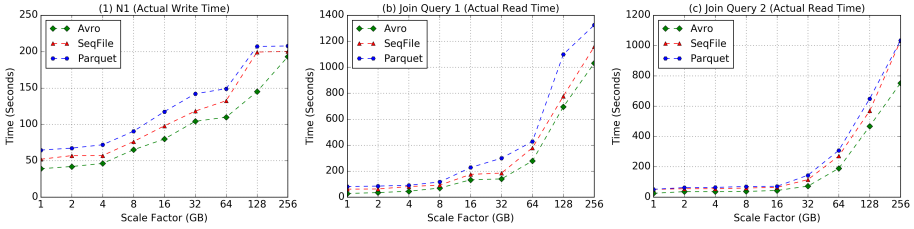


Fig. 4.15: Detailed experimentation conducted for node N1

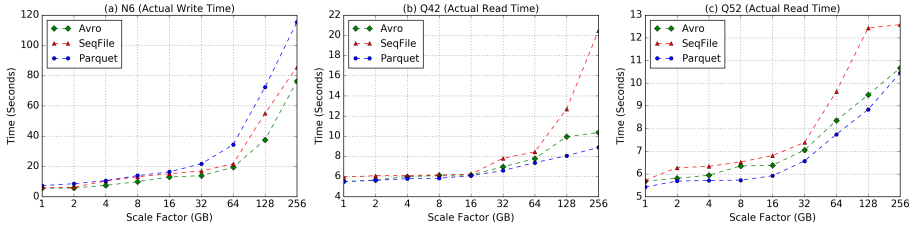


Fig. 4.16: Results for N6

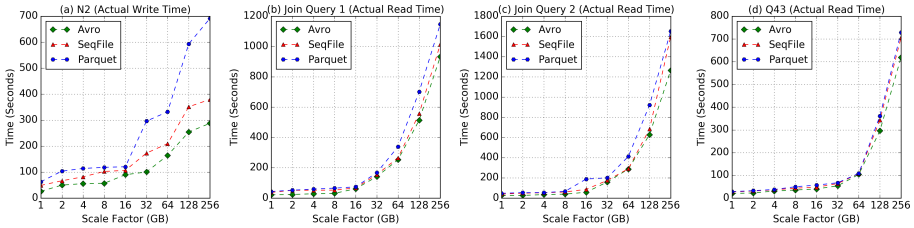


Fig. 4.17: Results for N2

Let us focus on N1 from the green group, for which the rule-based approach chooses the correct storage format (i.e., Avro). Figure 4.15 shows the actual write / read time of each storage format. It can be verified that the chosen layout (i.e., Avro) is always faster for both write and read operations.

Similarly, the rule-based approach also chooses the right storage format for grey group, which contains nodes with projection operations. However, the amount of data read is less than 70% and that's why it is better to use Parquet. Figure 4.16 shows the actual execution for N6. Figure 4.16a shows the actual execution time for both write and read operations. It can be seen that Parquet takes more time in writing (i.e., it writes more metadata), but its reading benefits compensate it as shown in Figures 4.16b and 4.16c read time for Q42 and Q52, respectively.

On the other hand, the rule-based approach failed to choose the correct storage formats for the white group. All these nodes involve filter operations, where the amount of data to be read depends on the selectivity factor and all of them are greater than or equal to 0.1 (see Table 4.7). As already shown in Figure 4.13, different storage formats perform differently depending on the amount of data read. Therefore, since the rule-based approach does not leverage on statistics, the data volume to be read is not considered and it fails when choosing the right storage format. As it can be seen in Figure 4.13b, the predicate push-down mechanism implemented by Parquet is useless when the selectivity factor is greater than *1.0E-05* for unsorted columns. The rule-based approach always considers predicate push-down to be worth and thus still chooses Parquet. Oppositely, since our cost-based model considers the selectivity factor, it is able to select the right format for these nodes. For example, the results of N2 for the white group are shown in Figure 4.17, where the optimal choice is Avro, which takes less time than Parquet in both write and read operations. All the nodes of the white group follow the same trend and our cost-based approach successfully choses the right storage format.

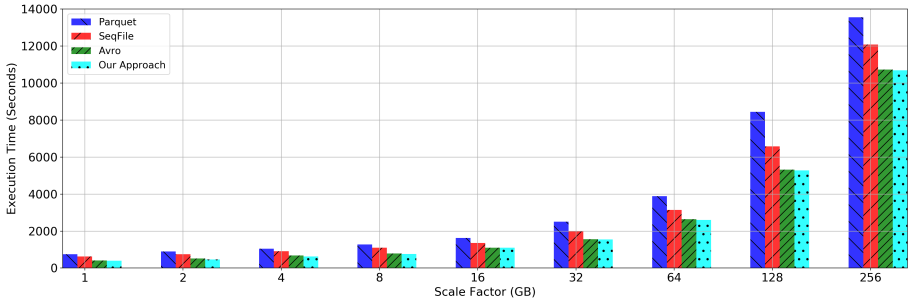


Fig. 4.18: Single Fixed Format vs Our Approach for TPC-DS

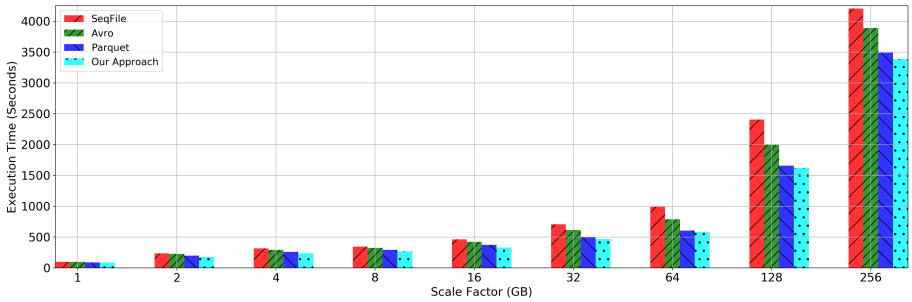


Fig. 4.19: Single Fixed Format vs Our Approach for TPC-H

In general, our cost model is able to decide the right format in all cases shown in Table 4.7, because it considers the amount of data read (which in this case is determined by the format file size and the selectivity factor of the operation), which actually depends on two operations, namely projection and selection. Figure 4.18 compares our approach with a typical approach materializing all chosen *IR* with a fixed format (i.e., always SequenceFile, Parquet or Avro). It shows the overall execution time of the DIF when using a single fixed format for *IR* with regard to a dynamic choice of the format based on our cost model. Our approach on average provides 1.6x speed up over fixed Parquet, 1.34x speedup over fixed SequenceFile, 1.03x speedup over fixed Avro and, in the average, it provides 1.33x speedup for TPC-DS.

Note that in TPC-DS, our cost model favors Avro, and this is due to the fact that the chosen *IR* have subsequent operations with high selectivity factors. In contrast, when we changed the workload to TPC-H, the cost model recommends Parquet in the majority of materialized nodes, due to the low selectivity queries. The overall results of TPC-H are shown in Figure 4.19. Observe that, for TPC-H, our approach on average provides 1.32x speedup

over fixed SequenceFile, 1.19x speedup over fixed Avro, 1.04x speedup over fixed Parquet and overall, it provides 1.18x speedup.

Note that, a given kind of workload may favor a certain format. However, a system should be able to adapt to different workloads as we saw when comparing TPC-DS (scan-based favors Avro) and TPC-H (selection-based and projection-based favor Parquet). In conclusion, our cost model is capable of choosing the appropriate storage format for different workloads, which always leads to improvements in query execution time.

4.8 Conclusion

Modern analytical workloads involve different types of queries in which a fixed storage format for *IRs* does not guarantee the best performance. Additionally, the currently available solutions have not considered at the same time choosing both *IRs* and the types of the storage layouts to be used for their storage. They consider these problems separately and therefore fail to provide an optimal solution. We explicitly focus on choosing the storage layouts for IRs and propose a whole process-cycle. Our proposed approach uses any existing solution to choose *IRs* and after deciding which *IRs* in a data intensive workflow to store, it chooses the best storage format, which improves performance, by analyzing their access patterns. Overall, this reduces the load time and, in general, the total workflow execution time. We have implemented our generic cost-based model for Hadoop and instantiated on different storage formats to show its effectiveness. Our evaluation results show the benefits of our approach and support our hypothesis that *IR* should be materialized by considering the best storage format.

5

Auto Tuning of Hybrid Layouts Using Workload and Data Characteristics

Ad-hoc analysis implies processing data in near real-time. Thus, raw data (i.e., neither normalized nor transformed) is typically dumped into a distributed engine, where it is generally stored into a hybrid layout. Hybrid layouts divide data into horizontal partitions and inside each partition, data are stored vertically. They keep statistics for each horizontal partition and also support encoding (i.e., dictionary) and compression to reduce the size of the data. Their built-in support for many ad-hoc operations (i.e., selection, projection, aggregation, etc.) makes hybrid layouts the best choice for most operations. Horizontal partition and dictionary sizes of hybrid layouts are configurable and can directly impact the performance of analytical queries. Hence, their default configuration cannot be expected to be optimal for all scenarios. In this chapter, we present ATUN-HL (Auto TUNing Hybrid Layouts), which based on a cost model and given the workload and the characteristics of data, finds the best values for these parameters. We prototyped ATUN-HL for Apache Parquet, which is an open source implementation of hybrid layouts in Hadoop Distributed File System, to show its effectiveness. Our experimental evaluation shows that ATUN-HL provides on average 85% of all the potential performance improvement, and 1.2x average speedup against default configuration.

5.1 Introduction

Data analysis plays a decisive role in today's data-driven organizations, which increasingly produce and store large volumes of data in the order of petabytes to zettabytes [71]. The storage and processing of such data has imposed a shift in the hardware, from single machines to large scale distributed systems. Apache Hadoop[1] is a pioneer large-scale distributed system and consists of a storage layer, namely Hadoop Distributed File System (HDFS)[2], and a processing layer, namely MapReduce[18]. The former allows to keep data in raw format without any normalization or pre-processing. The latter allows data-intensive flows (DIFs) to process raw data such that they are ready for the analysis.

Hadoop and many modern in-memory processing engines (i.e., Apache Spark[3]) provide high-level languages (i.e., Apache Pig, Hive, and SparkSQL) that facilitate writing DIFs for processing raw data (e.g., removing dirty data, integrating multiple data sources) stored in HDFS. Typically, the processed data is stored as a very wide table for analytical queries, because of its advantages over normalized tables [10, 54]. Hybrid layouts are de-facto preferred options for storing such wide tables, due to their built-in support for many basic operations (i.e., selection, projection, aggregation, etc.) allowing ad-hoc analysis, without the need of moving the data to other storage (i.e., relational, document store, etc.).

There are several available hybrid layout implementations, such as: Optimized Record Columnar (ORC)[4], Parquet[5] and CarbonData[6]. All of them follow the same physical structure. Data is stored into multiple horizontal partitions, known as stripes in ORC, row groups (RGs) in Parquet and blocklet in CarbonData, and each horizontal partition stores its data column-wise. Hybrid layouts also store min-max statistics [57] for each horizontal partition to help in filtering (i.e., partitions that do not match predicates of a query are skipped). In addition, they support dictionary encoding to encode repetitive values, that can also be used for further filtering partitions.

Despite having default values, the sizes of horizontal partition and dictionary are configurable. Thus, their values should be decided based on the

[1] https://hadoop.apache.org
[2] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[3] https://spark.apache.org
[4] https://orc.apache.org
[5] https://parquet.apache.org
[6] https://carbondata.apache.org

	Small Partition	Large Partition
Parallelism	+	-
Task overhead	-	+
Filtering	+	-
Metadata size	-	+
Dictionary encoding	-	+
Memory buffering	+	-
Load balancing	+	-

Table 5.1: Effect of horizontal partition size

data characteristics and usage. For instance, it is recommended to have a small size of horizontal partition for low selectivity queries and, a large size for high selectivity queries. However, it is not straight-forward to find an optimal size for all the queries, because this depends on their concrete selectivity and the type of data they access, therefore the problem becomes challenging. Moreover, the size of horizontal partitions can also effect different execution settings, which is shown in Table 5.1. It can be seen that small partitions positively impacts parallelism (by increasing the number of parallel tasks), filtering (by skipping unmatched partitions using statistics), memory buffering (they require less memory to buffer the data before flushing to the disk), and load balancing (by better distributing the loads among multiple machines). Whereas, large horizontal partitions help positively to reduce task overhead (by reading less metadata and reducing Java garbage collector overhead), metadata size (by storing less statistics), and also helps in performing better encoding (by encoding large number of repetitive values). In this chapter, we aim at improving filtering, metadata size and dictionary encoding by choosing the optimal partition size.

Similarly, the characteristics of data require different dictionary sizes to handle different attribute lengths and number of distinct values. The dictionary is not only important for compression, but it can also be used to filter partitions. Specifically, when data is unsorted and it is not possible to filter partitions simply using min-max statistics, as we will show in Section 5.4.3.

In this chapter, we present our approach, namely ATUN-HL, which helps to find the best values for the aforementioned parameters using a cost model, which estimates the optimal values for the size of the horizontal partition and the dictionary, based on the given workload and data characteristics. Moreover, it should also be noted that the chunk size of HDFS is always greater than or equal to the horizontal partition size. Hence, it should be configured

accordingly. We instantiated ATUN-HL for Parquet, to show its applicability in real scenarios and conducted an extensive evaluation on TPC-H[7] to show that ATUN-HL can significantly improve the query response times over Parquet with default configuration.

The main contributions of this chapter can be summarized as follows:

- We extend the cost model for hybrid layouts presented in Chapter 4.
- We propose ATUN-HL, a framework to optimize hybrid layouts.
- We prototype ATUN-HL on Parquet to show its benefits.
- We report the results of our extensive evaluation with TPC-H benchmark.

The remaining chapter is organized as follows. In Sections 5.2 and 5.3, we discuss the cost model and our approach in detail. In Section 5.4, we show our experimental results. Finally, in Section 5.5, we conclude the chapter.

5.2 Cost Model

In this section, we extend the cost model of Chapter 4. Specifically, we refine the selection cost model based on the use of min-max statistics and dictionary encoding. Further, we extend it to estimate the dictionary size for hybrid layouts.

First, we present the physical structure of hybrid layouts, which helps to build the cost model. Based on that, we estimate the cost of selections and the size of the dictionary. The former helps to find the optimal RG size. Our cost model considers two scenarios to estimate the selection cost, which are as follows: filtering using min-max statistics and using the dictionary. Likewise, it considers two types of dictionaries, i.e., global and local.

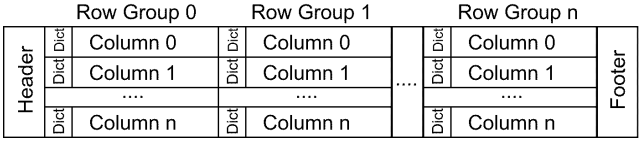


Fig. 5.1: Physical structure of hybrid layouts

[7]http://www.tpc.org/tpch

As shown in Figure 5.1, the data is divided into RGs (i.e., horizontal partitions), and inside each RG, it is stored column-wise. Further, if dictionary encoding is possible, first dictionaries are stored per column and afterwards the corresponding encoded data. If dictionary encoding is not possible, then the data values are stored contiguously without any encoding. Moreover, hybrid layouts also store metadata (e.g., min-max statistics) for each RG inside either the header or footer section. Thus, the size of hybrid layouts depends on the size of the actual data and metadata.

Variable	Description
	Data Statistics
$\|T\|$	Number of rows in a table
$\|C\|$	Distinct values of a column
$\|D\|$	Number of values in the dictionary
$Sorted_{Col}$	True for sorted and False for unsorted data
	Hybrid Layouts Variables
$Meta_{RG_{Size}}$	Size of metadata for an RG
$Marker_{Size}$	Size of sync marker

Table 5.2: Additional parameters introduced for the cost model

Our cost model for hybrid layouts relies on a wide range of statistical information that are summarized in Table 4.2, containing system constants, data statistics, workload statistics as well as hybrid layout variables. We assume that the constants which depend on the configuration of the environment (e.g., BW_{Disk}, BW_{Net}) are provided. In Table 5.2, we only show the new variables introduced for this chapter. Furthermore, we discuss the collection of statistics (e.g., dataset and workload) in Section 5.3.

$$Used_{RowGroups} = \frac{ColValues_{Size} \cdot |T| \cdot \#Cols}{RG_{Size} - (Marker_{Size} \cdot \#Cols)} \tag{5.1}$$

$$|RG| = \frac{|T|}{Used_{RowGroups}} \tag{5.2}$$

$$TotalMeta_{Size} = (Meta_{RG_{Size}} \cdot \#Cols) \cdot Used_{RowGroups} \tag{5.3}$$

5.2.1 Estimating the selection cost

The selection cost model estimates the number of RGs read from the disk and as well as the total read size. For this, first we need to estimate the total number of RGs using Equation 5.1, and the number of rows in an RG ($|RG|$) using Equation 5.2. Further, we also need to estimate the total size of

metadata (cf. in Equation 5.3), which is always read from disk to check the matching RGs. Our selection cost model focuses on two cases as discussed earlier. The first one considers filtering using min-max statistics of each RG, and second one filtering using the dictionary.

$$Read_{RowGroups} = \begin{cases} SF \cdot Used_{RowGroups} + 1 & \text{sorted data} \\ Used_{RowGroups} & \text{unsorted \& min-max} \\ (1 - (1 - SF)^{|RG|}) \cdot Used_{RowGroups} & \text{unsorted \& dictionary} \end{cases} \quad (5.4)$$

Filtering using min-max statistics.

There are two extreme cases when hybrid layouts use min-max statistics to filter RGs, depending on whether data is sorted or not. If data is completely sorted then the selected data will always be contiguous and we can calculate the total number of read RGs based on the selectivity factor as shown in Equation 5.4. We add one to handle the effect of position variation inside the RGs for sorted data, because hybrid layouts read the whole RG even if there is only one matching row. The reason to add one is illustrated in Figure 5.2. It shows two RGs and each has 5 rows. Let us assume that we select 3 rows. There are two possible scenarios: (A) there is no overlap and only one RG is read from disk; and (B) there is an overlap and two RGs are read. If we take the average of all possible positions of the first selected row in the first RG, it gives approximately $(SF \cdot Used_{RowGroups}) + 1$.

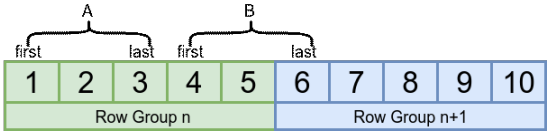


Fig. 5.2: Effect of position variation inside the RGs

If data is completely unsorted (i.e., uniform distribution), it is unlikely (shown in Section 5.4.3) to skip any RG, because the distribution of data makes the min-max range of each RG too wide. Hence, the read RGs will be the same as the total number of RGs. We will also experimentally show in Section 5.4.3 the ineffectiveness of min-max statistics for uniformly distributed unsorted data. Intermediate cases exist for different kinds of skewness, and Equation 5.4 could be enriched with corresponding estimations without affecting the rest of the chapter.

Filtering using the dictionary.

The dictionary can also be used to filter RGs when data is encoded. When min-max statistics fail to filter any RG, the dictionary is still very useful, because it contains all existing values. The number of RGs required to be read from disk can be estimated as in Equation 5.4 (borrowed from bitmap indexes [13]).

$$Used_{Chunks} = \left\lceil \frac{Used_{RowGroups} \cdot RG_{Size}}{Chunk_{Size}} \right\rceil \tag{5.5}$$

$$Read_{Size} = (Read_{RowGroups} \cdot RG_{Size}) + (TotalMeta_{Size} \cdot Used_{chunks}) \tag{5.6}$$

$$|Chunk| = \left\lfloor \frac{Chunk_{Size}}{RG_{Size}} \right\rfloor \tag{5.7}$$

$$Chunk_{Seeks} = \begin{cases} \frac{Read_{RowGroups}}{|Chunk|} + 1 & \text{if sorted} \\ Used_{Chunks} \cdot \left(1 - (1 - \frac{Read_{RowGroups}}{Used_{RowGroups}})^{|Chunk|}\right) & \text{if unsorted} \end{cases} \tag{5.8}$$

$$W_{ReadTransfer} = \frac{Time_{Disk} + (1-p) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (1-p) \cdot Time_{Net}} \tag{5.9}$$

$$\begin{aligned} Query_{Cost} = {} & \frac{Read_{Size}}{Chunk_{Size}} \cdot W_{ReadTransfer} \\ & + (Chunk_{Seeks} + Used_{Chunks}) \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{5.10}$$

The above equations give the expected number of RGs being read from disk, which helps in estimating the total query cost. In distributed processing engines, the data is processed in multiple tasks in parallel and the number of tasks equals to the number of chunks used to store the data, which can be estimated using Equation 5.5.

Moreover, we observed that each task reads all the metadata separately. The reason is that the distributed processing engines (such as Hadoop and Spark) create a separate process for each task with its own memory. This memory is not accessible to other tasks and hence, forces to read all metadata, and consequently, increases the reading size. We consider this in Equation 5.6, where we estimate the total read size.

Additionally, we take into consideration the disk seek cost, which depends on the number of chunks being read and also on the number of seeks required to fetch the metadata. The former is equal to the number of read chunks if data is sorted, because it reads consecutive RGs. In Equation 5.7, we calculate the total number of RGs inside a chunk, which is used in Equa-

tion 5.8 to estimate the total number of seeks for sorted data. Similar to filtering, we add add one to Equation 5.7 to handle the effect of position variation of RGs inside chunks. On the other hand, when data is unsorted, the number of seeks is directly influenced by the distribution of the read RGs, which are non-consecutive due to fact that any RG can match the predicate independently of its position. Thus, it can be approximated by estimating how many RGs are read from a chunk, which depends on the total number of RGs inside a chunk, again calculated using Equation 5.7. Similarly, we need to estimate the total seeks for reading metadata. As discussed earlier, typically, metadata is stored in the header or footer sections and one seek is required to locate it on the disk. Additionally, it is always read separately in every task, hence the total seeks of metadata will be equal to the total number of tasks (which is equal to the number of chunks).

In distributed processing engines, sometimes, they require to read the data remotely (for instance, it depends on occupancy of machines and unbalanced distribution of workload) and for it, we use a probability p to indicate the likelihood of chunks being accessed locally (i.e., data shipping through the network is needed to reach the operation executor). This is used in Equation 5.9 to estimate the weight (to calculate the resources usage) of transferring the chunk data compared to the corresponding seek time. Further, it is used along with the total number of seeks in Equation 5.10 to estimate the total query cost.

$$|D| = \begin{cases} |C| & \text{for global dictionary} \\ \lceil |C| \cdot (1 - ((|C| - 1)/|C|)^{|RG|} \rceil & \text{for local dictionary} \end{cases} \tag{5.11}$$

$$Dictionary_{Size} = |D| \cdot ColValue_{Size} \tag{5.12}$$

$$Used_{bits} = \lceil log_2 |D| \rceil \tag{5.13}$$

$$EncodedCol_{Size} = \begin{cases} Used_{bits} \cdot |T| & \text{for global dictionary} \\ Used_{bits} \cdot |RG| & \text{for local dictionary} \end{cases} \tag{5.14}$$

5.2.2 Estimating the size of the dictionary

As discussed earlier, hybrid layouts support dictionary encoding, which helps to encode repetitive values to reduce the size and also to facilitate filtering RGs. There are different implementations of dictionary encoding in different types of hybrid layouts. For instance, CarbonData uses a global dictionary

to encode the data, whereas Parquet uses a local dictionary inside every RG. However, these two implementations can be easily handled by the same cost model.

Global dictionary. The size of the dictionary depends on the number of values to store inside, which is the total distinct values (i.e., $|C|$) of a column estimated in Equation 5.11. The size of the dictionary for one column can be then estimated using Equation 5.12. Further, the average number of bits required to encode one value are estimated in Equation 5.13, and used in Equation 5.14 to estimate the encoded size of the data.

Local dictionary. Similarly, the size of the local dictionary depends on the number of values to be put inside the dictionary of an RG, which is the same as the distinct values of a column inside an RG. We estimate the total number of expected distinct values[8] inside an RG as shown in Equation 5.11. Next, similar to global dictionary, the average number of bits required to encode one value are estimated in Equation 5.13, and used further in Equation 5.14 to estimate the encoded size of the data.

5.3 ATUN-HL

In this section, we first discuss about the collection of data and workload characteristics. Next, we explain our methodology, which utilizes the cost model to find the optimal sizes for RG and dictionary.

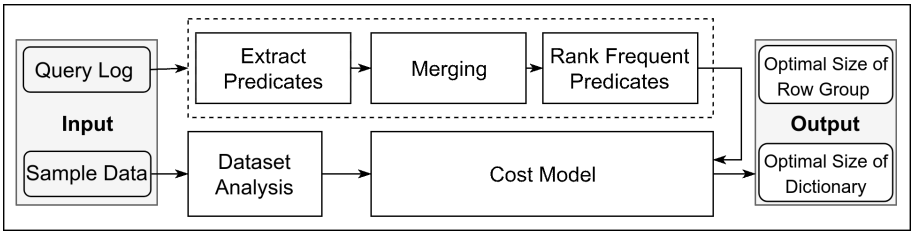


Fig. 5.3: Overview of ATUN-HL

[8]https://math.stackexchange.com/questions/72223/finding-expected-number-of-distinct-values-selected-from-a-set-of-integers

5.3.1 Collecting workload and data characteristics

Figure 5.3 shows the overview of our approach. It takes a query log and the sample data as input, and analyzes them in different components to extract statistical information. The query log is used to extract the information related to the workload. First, our approach extracts the clauses from all the query representatives. Second, it merges the similar clauses or the clauses that can be subsumed. Thirdly, it applies frequent itemset mining approach [32], to rank the most frequent clauses. Finally, it takes the top-k clauses to extract the workload information to be considered. On the other hand, our approach also takes a sample of data and computes the statistical information listed in Table 5.2. We use the single column profiling technique from [1].

The use of query log to optimize the parameters for future workloads is justified in [77, 78], which conclude that filters are recurring and only a small portion are entirely new over time.

Algorithm 1: Finding the best size of RG and dictionary

$Possible_{Dict_{Sizes}} = \{0\}$;
for $c \in Cols$ **do**
  $Dict_{Size} = RoundUpToKiloBytes(EstimateDictionarySize(c))$;
  $Possible_{Dict_{Sizes}}.insert(Dict_{Size})$;
end
$Best = [\infty, 0, 0]$; // $Best[Cost, RG_{Size}, Dict_{Size}]$
for $Dict_{Size} \in Possible_{Dict_{Sizes}}$ **do**
  $Z = EstimateEncodedSize(Dict_{Size})$;
  $Curr_{RG_{Size}} = Solver(\frac{d}{dRG_{Size}}(Cost_P(RG_{Size}, Z)) = 0)$;
  $Curr_{Cost} = Cost_P(Curr_{RG_{Size}}, Z)$;
  if $Curr_{Cost} < Best.Cost$ **then**
    $Best = [Curr_{Cost}, Curr_{RG_{Size}}, Dict_{Size}]$;
end
return $Best$;

5.3.2 Finding the best configuration parameters

Let us assume T is a wide table and has a set of columns defined as $C = \{c_1, c_2, ..., c_n\}$. Similarly, a query log is defined as $Q = \{q_1, q_2, ..., q_n\}$, the frequent clauses extracted from Q are defined as $P = \{p_1, p_2, ..., p_n\}$ the total cost of workload is calculated as $Cost_P(RG_{Size}, Z) = \sum_{p \in P} Query_{Cost}(RG_{Size}, Z)$, where Z represents the total size of T (considering dictionary encoding if needed). Our goal is to minimize $Cost_P$ by selecting the best RG and dictionary sizes.

Algorithm 1 shows the steps to find the optimal sizes of RG and dictionary. It initializes a set in line 1 with the element 0, which corresponds to the scenario where dictionary encoding is completely disabled for all columns. Next, in lines 2 to 4, it iterates over all the columns, computes their dictionary sizes, rounds them up to the nearest kilobytes, and stores them inside the set. Further, in lines 7 to 12, it iterates over all those dictionary sizes and computes the table size according to the current processed dictionary size. Then, the encoded size is used to find the optimal RG size by solving the derivative of the overall cost function. Finally, this value is used to compute the corresponding cost. If the cost is smaller than the best until now, we keep the current processed dictionary and RG sizes as the best ones.

In order to be able to find the minimum cost, we derive the function with respect to the RG size (i.e., $\frac{d}{dRG_{Size}}(Cost_P(RG_{Size}, Z)) = 0$). Equation 5.15 shows the overall query cost after replacing all variables except read RGs, which still depends on how data has been stored (see Equation 5.4). Notice that, we need to remove the ceiling function of Equation 5.5, as well as floor from Equation 5.7. We can do the former, because the number of chunks is much smaller than the total number of RGs, and it is only used in calculating the meta size and seek cost, and both are very small compared to the total reading size. Similarly, we also remove floor in Equation 5.7, due to its negligible impact on overall cost. We validated their removal with detailed experiments (see Section 5.4.3).

$$
\begin{aligned}
Z &= \begin{cases} (ColValue_{Size} \cdot |T| + Marker_{Size}) \cdot \#Cols & \text{no encoding} \\ (Dictionary_{Size} + EncodedCol_{Size} + Marker_{Size}) \cdot \#Cols & \text{encoding} \end{cases} \\
Y &= Meta_{RG_{Size}} \cdot \#Cols \\
Query_{Cost}(RG_{Size}, Z) &= \frac{Read_{RowGroups} \cdot RG_{Size} + \frac{Y \cdot Z^2}{RG_{Size} \cdot Chunk_{Size}}}{Chunk_{Size}} \cdot W_{ReadTransfer} \\
&\quad + \frac{Read_{RowGroups} + \frac{Z}{RG_{Size}}}{\frac{Chunk_{Size}}{RG_{Size}}} \cdot (1 - W_{ReadTransfer})
\end{aligned}
\tag{5.15}
$$

5.4 Experimental Results

In this section, we discuss the setup and the dataset used for our experiments. We also show the ineffectiveness of min-max statistics and usefulness of dictionary for unsorted data. Moreover, we provide the results to validate the accuracy of the cost model and to show the benefits of our approach.

Variable	Value
p	0.97
$Chunk_{Size}$	512MB
BW_{Disk}	1.30×10^8 bytes/second
BW_{Net}	1.25×10^8 bytes/second
$Time_{Seek}$	5.00×10^{-3} seconds
$Meta_{RG_{Size}}$	156 bytes
$Marker_{Size}$	16 bytes

Table 5.3: Values according to our environment

5.4.1 Setup

The machine used in our evaluation has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory, and 1TB SATA-3 of hard disk, and runs Hadoop 2.6.2 and Spark 2.1.10 on Ubuntu 14.04 (64 bit). Our approach is evaluated under two settings: a single node and a 4-machines cluster[9]. In the cluster, we dedicated one machine to HDFS name node and Spark master node together, and the remaining three machines to data nodes for Hadoop and workers for Spark. It should be noted that we use a very small cluster for our experiments, because our focus is on the initial map phase of the jobs that does not involve shuffling. Notice that the performance of the latter would be affected by the cluster size, but not that of the former.

We prototyped our approach for Apache Parquet 1.8.2, which further divides each column into multiple data pages (i.e., 1MB) and also stores min-max statistics per data page (i.e., 53 bytes). Nevertheless, currently Parquet does not support data page filtering, so we applied the cost model as described above. If needed, our cost model could be easily adaptabed to data page filtering by simply replacing RG size with data page size and $|RG|$ with the number of rows of a data page.

Table 5.3 shows the values of all environmental variables in our testbed.

[9]http://www.ac.upc.edu/serveis-tic/altas-prestaciones

In addition, default RG and dictionary sizes in Parquet are 128MB and 1MB, which we use in our evaluation together with best and worse obtained costs.

5.4.2 Dataset

As mentioned in [10, 54], very wide tables are common in modern analytical systems, because of their advantages in processing compared to normalizing data into narrower tables. Nevertheless, in TPC-H, the widest table has only 16 columns and in TPC-DS[10], only 26. To the best of our knowledge, there is no public benchmark available that consists of wide tables. Hence, we follow [77] to generate a wide table by completely denormalizing all other tables in TPC-H against *lineitem*. The FROM clauses in all queries are consequently changed to the corresponding denormalized table.

5.4.3 Results

We perform four types of evaluations for our approach. Firstly, we show the drawbacks of min-max based filtering for unsorted data through statistical and also experimental evaluation. Secondly, we show the benefits of dictionary based filtering for unsorted data. Thirdly, we validate the accuracy of our cost model. Finally, we show the performance improvements of our approach on the cluster by comparing it to the baseline setting.

Usefulness of min-max statistics.

As previously discussed, min-max statistics are not useful for unsorted data, because uniform data distribution makes it is unlikely to skip RGs. This behavior is validated with a detailed statistical and experiment evaluations.

$$P_{Skipping} = \frac{\sum_{i=1}^{|C|}\left(\left(\frac{i-1}{|C|}\right)^{|RG|} + \left(\frac{|C|-i}{|C|}\right)^{|RG|}\right)}{|C|} \tag{5.16}$$

$$Read_{RowGroups} = (1 - P_{Skipping}) \times Used_{RowGroups} \tag{5.17}$$

Since point queries (i.e., those that search one single value) have higher probability of skipping an RG than the other supported types (namely interval and list of values), we only provide a statistical cost model for point queries in Equation 5.16. This estimates the probability of being outside of an

[10]http://www.tpc.org/tpcds

RG, which would be the case if the value is less than the minimum of the RG or greater than the maximum. Thus, our cost model adds the probability of both (i.e., minimum and maximum) for each value of that column. Further, the probability of skipping one RG is used in Equation 5.17, to find the total number of RGs read.

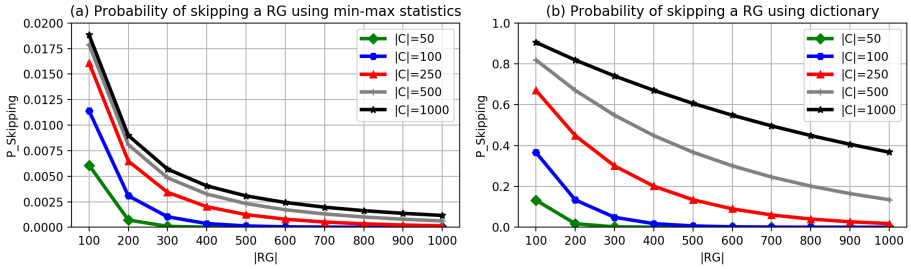


Fig. 5.4: Probability of skipping one RG

Figure 5.4a plots Equation 5.16 for different number of rows $|RG|$, and different number of distinct values of a column $|C|$, which was confirmed with the corresponding experiments. We took 100 as the minimum for $|RG|$, because Parquet does not allow less rows per RG than that. Thus, it can be observed that the probability of skipping an RG is very low (i.e., always less than $< 2\%$), confirming that min-max statistics are useless for unsorted data. Moreover, when the number of rows in an RG increases, the probability of skipping decreases, which means that it is almost certain that a full scan will be performed. A higher number of distinct values slightly increase the chances of skipping an RG, but it is still very unlikely for RGs with many rows.

Benefits of dictionary encoding.

We also plot Equation 5.4 for dictionary encoding (see Figure 5.4b), confirming its superiority over min-max statistics. It can be seen that this clearly gives higher probability of skipping, but the chances of skipping still decrease quickly as the number of rows in an RG grows. Yet, it helps with low selectivity queries (when min-max statistics still fail).

Cost model validation.

Figure 5.5 shows the comparison of our cost model, the estimation through its simplified version (which allows derivation as presented in Equation 5.15),

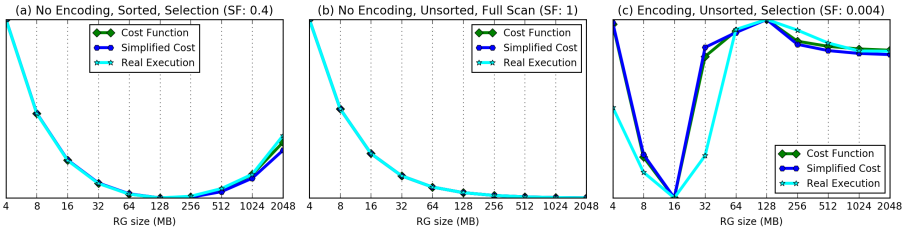


Fig. 5.5: Comparison between cost model, simplified version, and real execution

and also actual execution (averaging 250 random runs and in each run, filtering on a random value). We normalized them ($(x - min)/(max - min)$) to facilitate visual comparison. Moreover, as we will show below, the different units (as our cost model only considers I/O cost) do not affect the quality of our prediction to choose the optimal RG size, since the estimated values always preserve the shape of the actual ones (i.e., minimum real cost is obtained for approximately the same value in the model).

We empirically validated the estimations on both sorted and unsorted data, with and without encoding. It can be seen that our cost model and its simplified version are very close and result in approximately the same value. Hence, the derivative can be safely used to find the optimal RG size. Moreover, these both versions follow exactly the same trend as the actual execution.

Performance evaluation.

We analyzed TPC-H queries to extract the clauses and ranked them according to their usage. The top 6 clauses which appear in 82% of the queries, are used to find the optimal RG and dictionary sizes. ATUN-HL chooses 30.76MB (that we round up to 32MB) for RG and 1MB for dictionary (that is the default one).

Figure 5.6a shows our estimated overall cost for TPC-H queries. It can be seen that ATUN-HL predicts the default RG size (i.e., 128MB) as the worst configuration (being the minimum at 32MB). As discussed earlier, it is very unlikely for Parquet to skip any RG, when the number of rows in an RG grows. When this turning point is crossed, the larger the RG the better, and our estimated cost depicts this behavior after 128MB. Moreover, we also verified our estimation with detailed experiments as shown in Figure 5.6b and Figure 5.6c. Figure 5.6b compares the time improvements of ATUN-HL against the optimal, default, and worst configurations. ATUN-HL is not

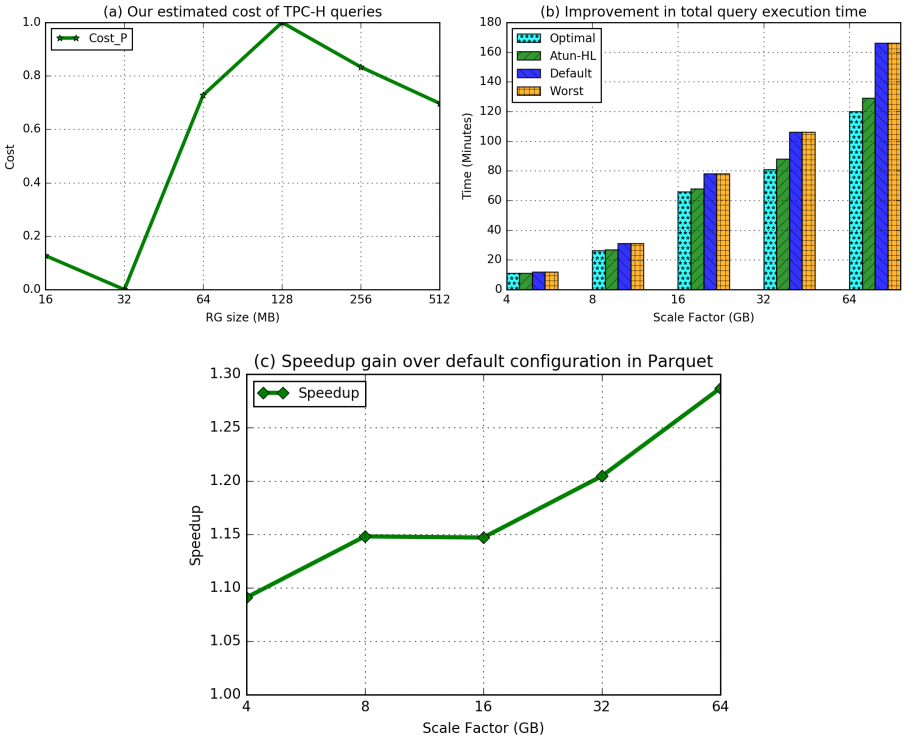


Fig. 5.6: Speedup gain

far from the optimal configuration, resulting in 85% of all potential gain. Additionally, Figure 5.6c shows the relative gain with regard to default RG size, which is 1.2X speedup on average (for the tested scale factors), clearly increasing with the increase in scale factor.

Finally, in Figure 5.7, we also scrutinize the effect on individual query execution time for scale factor $64GB$. This shows that our approach improves the execution time of most of the queries, but does not help those actually performing a full scan (i.e., Q1, Q13, Q15, and Q16) because of one reason (i.e., high SF, $> 10\%$) or another (i.e., string matching using regular expression, which is not yet supported by Parquet). As shown above, the large RG size is always better for full scan.

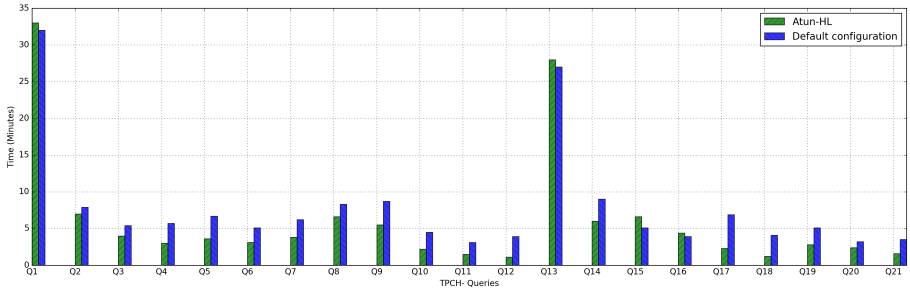


Fig. 5.7: Improvement in query execution time for 64GB scale factor

5.5 Conclusions

Hybrid layouts are widely used to store processed data in highly distributed Big Data systems to perform ad-hoc analysis. Nevertheless, they have many configurable parameters that need to be tuned according to the characteristics of the data and workload, which can heavily impact query performance. Consequently, we proposed a cost-based approach to help optimizing such hybrid layouts. We prototyped our approach for Apache Parquet, evaluated it on TPC-H queries, and showed the improvement it provides.

6

Configuring Parallelism for Hybrid Layouts using Multi-Objective Optimization

Distributed processing frameworks (e.g., Hadoop, Spark) divide the data into multiple partitions and process each partition in separate tasks. These tasks can be executed in parallel and thus speedup the analysis. Furthermore, the advent of hybrid layouts has additionally sped-up the analysis by allowing to read less data for certain operations (i.e., projection, selection). Yet, distributed frameworks do not consider the actual data read when creating the tasks to process the partitions. Thus, the number of tasks is always created based on the total file size and not on the actual data being read. However, this may lead in launching more tasks than needed, which in turn may increase the job execution time and induce significant waste of computing resources. The latter due to the fact that each task introduces extra overhead (e.g., initialization, garbage collection, etc.). To allow a more efficient use of resources and reduce the job execution time, we propose a method that decides the number of tasks based on the data being read. To this end, we first propose a cost-based model for estimating the size of data read in hybrid layouts. Next, we use the estimated reading size in a multi-objective optimization method to decide the number of tasks and computational resources to be used. To show the effectiveness of our approach we prototype it for Apache Parquet and Spark, and found that our estimations were highly (0.96) correlated to the real execution times. Additionally, we perform experimental evaluation on TPC-H to show that, on average, our recommended configurations are only 5.6% away from the Pareto front and provide 2.1x speedup against the default solutions.

6.1 Introduction

The competition in businesses demands quick insights from data, which is exponentially growing from petabytes to zettabytes [71]. Researchers have proposed distributed processing frameworks (e.g., Hadoop ecosystem[1] and Spark[2]) for quickly processing such large volumes of data to meet the business demands. These frameworks provide distributed storage (e.g., HDFS[3]) and distributed processing [18]. In addition, for more efficient analysis, very wide tables [10, 54] are being used to store non-normalized data in hybrid layouts [9, 58]. Through their built-in operations (e.g., projection, selection), these layouts read data more efficiently from the disk.

There are several available hybrid layout implementations, such as: Optimized Record Columnar (ORC)[4], Parquet[5] and CarbonData[6]. All of them follow the same physical structure as shown in Figure 5.1. Data is stored into multiple horizontal partitions, known as stripes in ORC, row groups (RGs) in Parquet and blocklets in CarbonData, and each horizontal partition stores its data column-wise, which is beneficial for projection. Statistics about the data are stored in each partition, and they may help on filtering partitions. Furthermore, hybrid layouts support dictionary encoding for compressing repetitive values of individual columns. The dictionary can also be used to filter partitions.

Therefore, hybrid layouts allow to read less data from the disk. Nevertheless, this is not thoroughly exploited by distributed frameworks when deciding the number of tasks for processing the data. They always decide the number of tasks based on the total table size and not on the portion of the table being read. This leads to the over-provisioning of tasks, where many tasks remain idle — without any data to process, but still present extra overhead (e.g., initialization time, garbage collection). Furthermore, the idle tasks also waste the computational resources which are assigned to them. The latter is not considered even in the area of cloud computing [35, 40, 72, 79], where computational resources are decided based on the total data size. This leads to wastage of resources and money.

[1] `https://hadoop.apache.org`
[2] `https://spark.apache.org`
[3] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`
[4] `https://orc.apache.org`
[5] `https://parquet.apache.org`
[6] `https://carbondata.apache.org`

Motivational Example. As mentioned above, data is first partitioned and then processed in parallel. Let us assume that a partition is equal to one RG and as shown in Figure 1.3, we have a hybrid layout containing four RGs. Let us further assume that we are applying a filter operation, which only satisfies two RGs. By default a distributed framework would create four tasks. However, two of them would remain idle (c.f. Figure 1.3a), and yet would read extra metadata from the disk and would require extra initialization time. This would increase the makespan — execution time. Furthermore in terms of computational resources, four executors[7] would be required to execute all these tasks in parallel. Whereas, in an ideal scenario, based on the amount of data read (c.f. Figure 1.3b) only two tasks with two executors would be enough. The latter would help on saving computational resources and reduce the makespan.

As argued above, we need to decide the number of tasks based on the actual data read from the disk. To do that, we first need to estimate the read size, which can be done by utilizing our cost model presented in Chapter 4. The cost model estimates the scan, projection, and selection sizes for hybrid layouts.

In this chapter, we extend it further to estimate the makespan of the job implementing a query based on the estimated reading size. Thus, we design a framework which takes a user query and data statistics as inputs to estimate the reading size, and then through a multi-objective optimization method [19] decide the number of *tasks* and *executors*.

After configuring the number of tasks and executors, the query would be automatically submitted to a distributed processing framework. We implemented our approach for Parquet and Spark to show its applicability in real scenarios.

The main contributions of this chapter can be summarized as follows:

- We extend the cost model for hybrid layouts presented in Chapter 4 to estimate the makespan of a job.
- We propose a framework based on a multi-objective optimization method [19] that using our extended cost model, configures the number of tasks and executors for a given query.
- We prototype our approach on Parquet and Spark to show its benefits.

[7] An executor is a computational resource/unit which can execute a task.

- We report the results of our extensive evaluation with TPC-H benchmark.

The remainder of this chapter is organized as follows: In Sections 6.2 and 6.3, we present the cost model and the architecture of our approach. In Section 6.4, we discuss a multi-objective method to find the number of tasks and executors. In Section 6.5, we present our experimental results and finally, in Section 6.6, we conclude the chapter.

6.2 Cost Model for Hybrid Layouts

In Chapter 4, we did not consider configuring the number of tasks and machines, but focused on choosing different storage layouts based on their reading and writing cost. Thus, we extend the cost model to consider new factors (e.g., $Used_{Executors}$, P_{Size}, etc.) and estimations to help in deciding the number of tasks and machines for a given query. In this section, we present the extended cost model for estimating the number of tasks and executors. It should be noted that the number of tasks depends on the partition size (also known as *input split*).

Variable	Description
	System Constants
$Used_{Executors}$	Number of executors for processing
P_{Size}	Size of partition to control the number of tasks

Table 6.1: Additional parameters introduced for the cost model

6.2.1 Parameters of the Cost Model

Our cost model for hybrid layouts relies on a wide range of statistical information that are summarized in Table 4.2, containing system constants, data statistics, workload statistics as well as hybrid layout variables. We assume that the constants which depend on the configuration of the environment (e.g., BW_{Disk}) are provided (they can be taken from the configuration files of the cluster). Furthermore, we discuss the collection of statistics (i.e., dataset and workload) in Section 6.3. In Table 6.1, we only show the newly introduced variables for this chapter.

6.2.2 Physical Format of Hybrid Layouts

As shown in Figure 5.1, hybrid layouts divide the data into multiple RGs (estimated using Equation 5.1 on page 89) and each RG contains a subset of rows (estimated using Equation 5.2 on page 89). In each RG, hybrid layouts store data column-wise and its size can be estimated using Equation 4.11 (on page 64). Moreover, hybrid layouts also store metadata (e.g., min-max statistics) for each RG inside either the header or footer section, which can be estimated using Equation 5.3 (on page 89). The size of actual data and metadata are further used in Equation 4.1 (on page 61) to estimate the total size of the file.

$$Used_{Tasks} = \left\lceil \frac{Body_{Size}}{P_{Size}} \right\rceil \tag{6.1}$$

$$Used_{Waves} = \left\lceil \frac{Used_{Tasks}}{Used_{Executors}} \right\rceil \tag{6.2}$$

$$LastWave_{Executors} = ((Used_{Tasks} - 1) \mod Used_{Executors}) + 1 \tag{6.3}$$

$$\#RGs_{Partition} = \left\lceil \frac{P_{Size}}{RG_{Size}} \right\rceil \tag{6.4}$$

6.2.3 Estimating Number of Tasks

Modern distributed processing frameworks decide the number of tasks based on the total file size (which is the size of actual data without metadata) and the partition size (estimated using Equation 6.1). Moreover, the degree of parallelism depends on the number of executors. All tasks cannot be executed at once, if the number of executors is less than the total number of tasks. Thus, we need multiple rounds/waves to finish the job (estimated using Equation 6.2). Further, we can calculate the number of executors active in the last wave using Equation 6.3. Additionally, each partition contains one or more RGs, which can be estimated using Equation 6.4.

6.2.4 Estimating MakeSpan

In this chapter, we focus on read-only analytical jobs, to estimate the amount of data read for their first operation and based on that, we try to find the best partition size to control the number of tasks. Given the simplicity of a file system (far from that of a DBMS), only three operations need to be considered: scan, projection, and selection. These three operations can be

generalized to *selection sorted* and *selection unsorted*, because scan and projection operations are just the extreme cases of selection unsorted with selectivity factor of 1 (i.e., they read all RGs).

Data read estimation. As mentioned above, hybrid layouts help to read only the referred columns and their size can be estimated using Equation 4.19 (on page 67). Additionally, they use the available metadata (e.g., min-max statistics) to filter some RGs. If selection is applied on sorted data, the average number of read RGs can be calculated directly based on the selectivity factor as shown in Equation 5.4 (on page 90). We add one to handle the effect of position variation inside the RGs, because hybrid layouts read the whole RG even if there is only one matching row [58]. Whereas, for selection of unsorted data, the expected number of read RGs can be estimated using Equation 5.4 (on page 90).

$$Full_{Partitions} = \begin{cases} Used_{Tasks} - 1 & \text{selection unsorted} \\ \left\lceil \frac{Read_{RGs}}{\#RGs_{Partition}} \right\rceil & \text{selection sorted} \end{cases} \tag{6.5}$$

$$Partial_{Partitions} = \begin{cases} 0 & \text{selection unsorted} \\ 2 & \text{selection sorted} \end{cases} \tag{6.6}$$

$$Last_{Partition} = \begin{cases} 1 & \text{selection unsorted} \\ 0 & \text{selection sorted} \end{cases} \tag{6.7}$$

$$Empty_{Partitions} = Used_{Tasks} - Full_{Partitions} - Partial_{Partitions} - Last_{Partition} \tag{6.8}$$

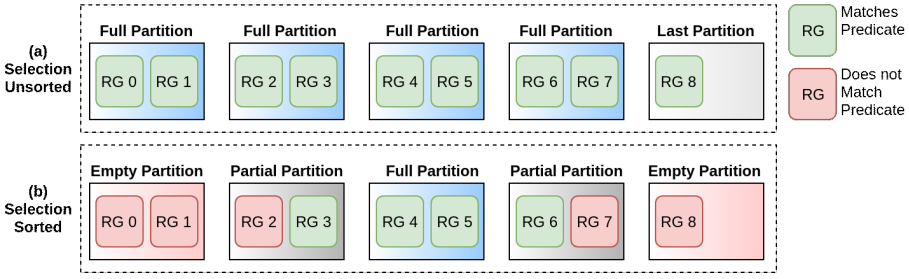


Fig. 6.1: Type of partitions in selection sorted and unsorted

Types of partitions. Distributed processing frameworks process data by dividing them into multiple partitions, where each partition is processed in a separate task. For *selection unsorted*, every task processes a full partition except the last task, whose partition might not be completely full, as shown in Figure 6.1a. Equation 6.5 and Equation 6.7 indicate the number of full and last partitions. Thus, for unsorted data, any partition has the same probability of containing data. However, *selection sorted* guarantees that we read full partitions, except for, potentially, the first (from where selection starts) and last one (where selection ends), because requested data will not start just at the beginning and finish just at the end of a partition. To reflect this, we always have two partial partitions (Equation 6.6) and the number of full partitions depends on the number of RGs to be read (Equation 6.5). Importantly, note that all other partitions will nevertheless read their metadata to determine no data matches the predicate (Equation 6.8). Figure 6.1b exemplifies these partitions.

Cost estimation. The total cost of a task depends on four factors: *initialization cost*, *I/O cost*, *CPU cost*, and *networking cost*. The *initialization cost* is constant and can be determined according to the execution environment. The *I/O cost* depends on the amount of data read within a task and the disk bandwidth. We do not consider *CPU cost* due to its negligible impact compared to *I/O cost* (existing works [9, 58] already proved that this is enough to capture the execution trend). Finally, we do not need any shuffling [9], because we focus only on the first operation loading data and therefore, the *networking cost* for shuffling is considered to be zero.

However, there might be some cases when partition size goes beyond the chunk size and it may require some chunks to be transferred over the network. There are two solutions to handle this scenario. One solution is to put a maximum limit on the partition size and always keep it less than the chunk size. The other solution is to use an existing approach [47], which transfers data in advance to avoid idle cycles on the processing machines. The approach to be used should be chosen based on the business requirements.

$$Cost_{Metadata} = \frac{Meta_{Size}}{BW_{Disk}} + \frac{Meta_{Size}}{BW_{Net}} \cdot (Used_{Executors} - 1) \tag{6.9}$$

There is still a networking cost for metadata (Equation 6.9), because current solutions require to sequentially transfer metadata to all other executors

before start processing the data. Typically, it is read and transferred by the master or driver executor.

$$Cost_{Full_{Partition}} = Cost_{Init} + \frac{Meta_{Size} + RefCols_{Size} \cdot \#RGs_{Partition} \cdot (1 - (1 - SF)^{|RG|})}{BW_{Disk}} \quad (6.10)$$

$$Odd_{Data} = \frac{RefCols_{Size} \cdot (Full_{Partitions} \cdot \#RGsPartition - Read_{RGs})}{Partial_{Partitions}} \quad (6.11)$$

$$Cost_{PartialPartition} = Cost_{Init} + \frac{Meta_{Size} + Odd_{Data}}{BW_{Disk}} \quad (6.12)$$

$$Residual_{Data} = RefCols_{Size} \cdot (Used_{RGs} - \#RGsPartition \cdot Full_{Partitions}) \cdot (1 - (1 - SF)^{|RG|}) \quad (6.13)$$

$$Cost_{Last_{Partition}} = Cost_{Init} + \frac{Meta_{Size} + Residual_{Data}}{BW_{Disk}} \quad (6.14)$$

$$Cost_{Empty_{Partition}} = Cost_{Init} + \frac{Meta_{Size}}{BW_{Disk}} \quad (6.15)$$

Each partition has an initialization cost, which is a constant, and I/O cost (which depends on metadata and the amount of data read inside the partition). As shown in Figure 6.1, *full partitions* read all matched RGs inside a partition, and their cost can be estimated using Equation 6.10. Equation 6.11 estimates data read from partial partitions and Equation 6.12 its cost. Equation 6.13 reads the data left in the last partition and Equation 6.14 its cost. The other partitions just read metadata and its cost is in Equation 6.15.

$$Cost_{AllTasks} = Full_{Partitions} \cdot Cost_{Full_{Partition}} + Empty_{Partitions} \cdot Cost_{Empty_{Partition}} + Partial_{Partitions} \cdot Cost_{Partial_{Partition}} \quad (6.16)$$

$$AvgCost_{Task} = \frac{Cost_{AllTasks}}{Used_{Tasks} - Last_{Partition}} \quad (6.17)$$

These cost of all partitions help to estimate the total cost of all tasks using Equation 6.16, which is used in Equation 6.17 to estimate the average cost of a task. It should be noted that the cost of last partition is only applied for selection unsorted and it is considered separately when estimating the total makespan. Thus, we do not consider its cost here.

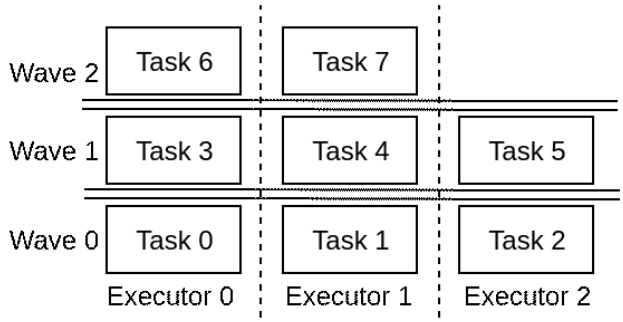


Fig. 6.2: Execution of tasks

Estimating makespan. As discussed earlier, each task processes different amounts of data and thus, some tasks can finish earlier compared to others. Likewise, each executor can finish their assigned tasks on different times. Thus, we should estimate makespan based on the executor that is processing largest stack of tasks (e.g., in Figure 6.2, Executor 0 and Executor 1 are the ones with the largest stack). This can be done by estimating standard deviation among tasks and used it further for estimating overall makespan of an operation.

$$Used'_{RGs} = \#RGsPartition \cdot Full_{Partitions} \tag{6.18}$$

$$Read'_{RGs} = Used'_{RGs} \cdot (1 - (1 - SF)^{|RG|}) \tag{6.19}$$

For standard deviation, first we need to estimate the number of RGs inside full partitions, using Equation 6.18. It is further used in Equation 6.19 to estimate the actual read RGs based on the selectivity factor.

$$Stdev = \begin{cases} \left(\#RGs_{Partitions} \cdot \dfrac{Read'_{RGs}}{Used'_{RGs}} \cdot \dfrac{Used'_{RGs} - Read'_{RGs}}{Used'_{RGs}} \right. & \text{Selection unsorted} \\ \left. \cdot \dfrac{Used'_{RGs} - \#RGs_{Partitions}}{Used'_{RGs} - 1} \right) & \\ \sqrt{\dfrac{\sum_{i=1}^{Used_{Tasks}} (Cost_{Task_i} - AvgCost_{Task})^2}{Used_{Tasks} - 1}} & \text{Selection sorted} \end{cases} \tag{6.20}$$

Finally, we use hypergeometric distribution [76] for *selection unsorted* to es-

timate the standard deviation of a full partition in Equation 6.20, based on the read RGs. Hypergeometric distribution estimates the standard deviation of choosing a subset of items without replacement from the total available items. This is similar to our case where we are also trying to select RGs (i.e., $Read'_{RGs}$) from the total RGs (i.e., $Used'_{RGs}$). Similarly, we also estimate standard deviation in Equation 6.20 for *selection sorted*.

$$MakeSpan = \begin{cases} \textbf{When LastWave}_{\textbf{Executors}} = \textbf{1} \\ Cost_{Full_{Partitions}} * (Used_Waves - 1) + Cost_{Last_{Partition}} + Cost_{Metadata} \\ \\ \textbf{When LastWave}_{\textbf{Executors}} > \textbf{1} \\ (Used_{Waves} \cdot AvgCost_{Task}) + Cost_{Metadata} \\ +Stdev \cdot \sqrt{Used_{Waves} \cdot 2 \cdot log_e(LastWave_{Executors})} \end{cases} \tag{6.21}$$

Finally, we estimate makespan for an operation using Equation 6.21. There are two scenarios based on the number of executors active in the last wave. In the first scenario, there is only one executor in the largest stack. In this case, the last task is processing $Last_{Partition}$. Then, we do not need to take any standard deviation, because there is one single largest stack. Thus, we just add the average duration of all task in that stack.

In the second scenario, the makespan depends on metadata transfer, the average cost of a task, the number of executors running in the last wave, and their standard deviation. Thus, we need to estimate expected maximum [16] of those by using the standard deviation as presented in Equation 6.21, which accounts for the standard deviation of the addition of tasks (i.e., $\sqrt{Used_{Waves}}$), as well as the maximum among executors in the last wave (i.e., $\sqrt{2 \cdot log_e(LastWave_{Executors})}$).

6.3 Our Approach

In this section, we discuss our approach in detail. Figure 6.3 shows its architecture, which does not require any change in a distributed processing framework (i.e., it is fully transparent for users). The main function blocks of our architecture are the following ones:

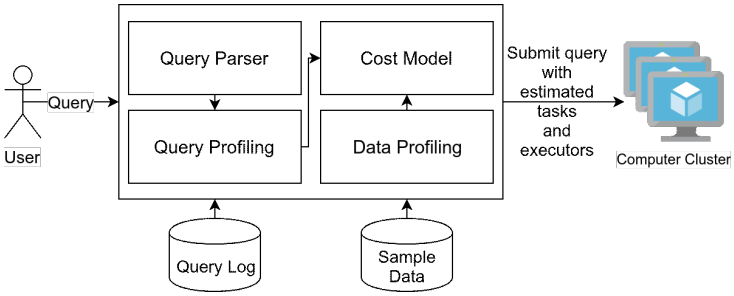


Fig. 6.3: Architecture of our approach

6.3.1 Query Parser

The query parser takes a query as input and uses an existing parser (i.e., SparkSQL parser[8]) to validate its syntax. After validation, it generates the physical plan of the query as an XML and forwards it to the next module. The physical plan represents a tree that starts from input sources to the final output. It also highlights the operations, which can be push-down to the storage layer.

6.3.2 Query Profiling

The query profiling takes physical plan as an input and extracts pushdown operations from the plan. Hybrid layouts can only pushdown two operations: projection and selection. It is easy to extract referred columns from the physical plan. Whereas, for selection, it is not possible to extract selectivity factor (SF) from the physical plan. To extract SF, query log needs to be parsed for analyzing the old executions of the same query. Finally, this module passes the pushdown operations along with required statistical information of operations to the cost model.

6.3.3 Data Profiling

The data profiling module takes a sample of data and computes the statistical information listed in Table 6.1. We rely on an existing approach, namely single column profiling technique from [1].

[8] `https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-AstBuilder.html`

6.3.4 Cost Model

The cost model is used to estimate the reading size for a given query. Typically, a query can have many operations linked together as a Directed Acyclic Graph (DAG). The operations are ordered based on their possibility of pushdown to the storage layer. Hence, the first operation is always a pushdown operation, which reads directly from the disk and impacts parallelism. The subsequent operations takes processed data from the first operation, which modern processing frameworks (e.g., Spark) always keep in memory.

The cost model takes a pushdown operation, workload, data statistics, and cluster configuration as inputs, which are used to estimate the makespan for a given partition size and the number of executors as presented in Section 6.2. Our goal is to find the best partition size and the number of executors, which can be done using a multi-objective optimization method (see Section 6.4).

6.4 Multi-Objective Optimization

In this chapter, we focus on optimizing two objectives, which are contradicting to each other. These objectives are *makespan* of query and *resource usage* (i.e., number of executors) required to run the query. We would like to minimize both together. However, they are mutually contradicting, i.e., if we want to reduce makespan, we require more computational resources. In the same way, if we want to save computational resources, we have to compromise makespan. Thus, we need to find a trade-off between them that satisfies user requirements and constraints.

The **first objective function** (i.e., $MakeSpan(Operation_{Type}, P_{Size}, Used_{Executors})$) is based on the makespan estimation according to Equation 6.21 (as defined in Section 6.2) for a given operation type, partition size, and the number of executors. Similarly, the **second objective function** (i.e., $Resource_{Usage}(P_{Size}) = Cost_{AllTasks}$ as defined in Equation 6.16 estimates the resource usage, which increases with the number of tasks.

$$P_{Size} >= RG_{Size} \quad and \quad P_{Size} <= \frac{Total_{Size}}{Used_{Executors}} \tag{6.22}$$

$$P_{Size} <= Executor_{Memory_{Size}} \tag{6.23}$$

$$Used_{Executors} <= Max_{Executors} \tag{6.24}$$

To avoid unfavorable or even impossible configurations, we need to add three constraints. Firstly, Equation 6.22 guarantees that the partition size is always greater than or equal to the RG size and at the same time, we have enough partitions to utilize all assigned executors as shown in Equation 6.23. Finally, Equation 6.24 enforces the maximum number of executors.

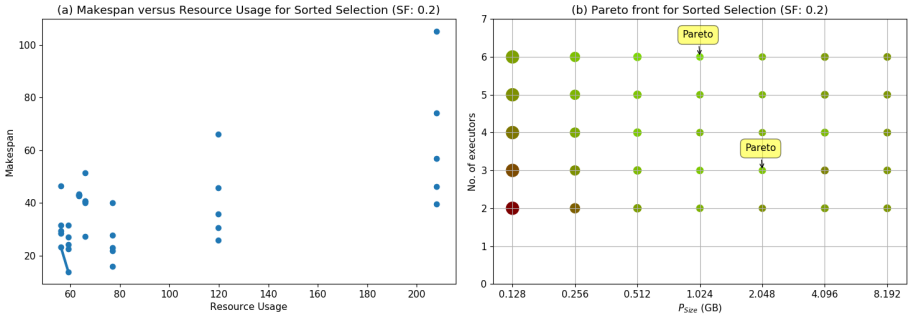


Fig. 6.4: Pareto front for a selection (circle size represents resource usage, the bigger the more resources; and color represents makespan, red for high and green for low)

Typically, there is no single optimum in a multi-objective optimization problem, but a Pareto front which contains many potentially optimal solutions depending on user prioritization of one objective or another (as shown in Figure 6.4a). Thus, the user has to choose one configuration from the Pareto front to, in the end, execute the query at hand. Our framework[9] facilitates the user choice by reducing the many possible configurations to very few (belonging or close to the Pareto front), so helping her to select one according to her preferences. As shown in Figure 6.4b, the position in the solution space does not determine the position in the configuration space, which hinders user's choice. In this case, our framework leaves only two (out of thirty-five possible solutions), which satisfy both objectives according to our estimations. When the user selects one of those two, the framework submits the query seamlessly to a processing engine by configuring the partition size and number of executors accordingly.

In this chapter, we do not focus on proposing a new multi-objective method, rather we focus on finding the best possible configuration (i.e., number of tasks and executors) for a given query. Thus, we use an existing multi-objective optimization approach, namely NSGA-II [19], implementing genetic algorithms. It simply takes objective functions along with constrains as input, and produces the Pareto front as an output.

[9]http://www.essi.upc.edu/dtim/tools/adbis2019

6.5 Experimental Results

In this section, we discuss the setup and dataset used in our experiments. We also provide the results that validate the accuracy of the cost model and show the benefits of our approach.

Variable	Value
$Used_{Executors}$	2, 3, 4, 5, and 6
$Chunk_{Size}$	128 MB
BW_{Disk}	1.3 x 10^8 bytes/second
BW_{Net}	1.25 x 10^8 bytes/second
$Cost_{Init}$	1 second
RG_{Size}	128 MB
$Marker_{Size}$	16 bytes
$Meta_{Cols_{Size}}$	156 bytes
$Header_{Size}$	4 bytes

Table 6.2: Values according to our environment

6.5.1 Setup

We perform experiments on 5-machines cluster. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128 GB of main memory, and 1TB SATA-3 of hard disk, and runs Hadoop 2.6.2 and Spark 2.1.10 on Ubuntu 14.04 (64 bit). In the cluster, we dedicated one machine for the HDFS name node and Spark master node together, and the remaining machines to data nodes for Hadoop and executors for Spark. It should be noted that we use a very small cluster for our experiments, because our focus is on the initial map phase of the jobs that does not involve shuffling. Notice that the performance of the latter would be affected by the cluster size, but not that of the former. We prototyped our approach for Apache Parquet 1.8.2. Table 6.2 shows the values of all environmental variables in our testbed. We also configured *replication factor* equals to the number of machines to have replicas on every machine thus avoiding chunk transfer in the case of having partition size greater than the chunk size. For real use cases, [47] can be used to transfer HDFS chunks in advance to avoid idle cycles on the processing machines or configure the maximum limit on partition size to make sure it should not exceed HDFS chunk size.

We also instantiated our cost model presented in Section 6.2 for scan, projection, and selection (both sorted and unsorted). *Scan* operation is just a

selection unsorted with selectivity factor 1, referring all the columns of the table. Similarly, *Projection* is also a selection unsorted with selectivity factor 1 and based only on the referred columns. For *Selection*, we just need to give selectivity factor and it would work for both.

6.5.2 Results

As mentioned in [10, 54], very wide tables are common in modern analytical systems, because of their advantages in processing compared to normalizing data into narrower tables. Nevertheless, to the best of our knowledge, there is no public benchmark available that consists of wide tables. Therefore, in this section, we first validate the accuracy of our cost model for makespan with a synthetic dataset of a very wide table. Further, we present the results to show the benefits of our approach to choose the best configuration for queries over the TPC-H denormalized schema.

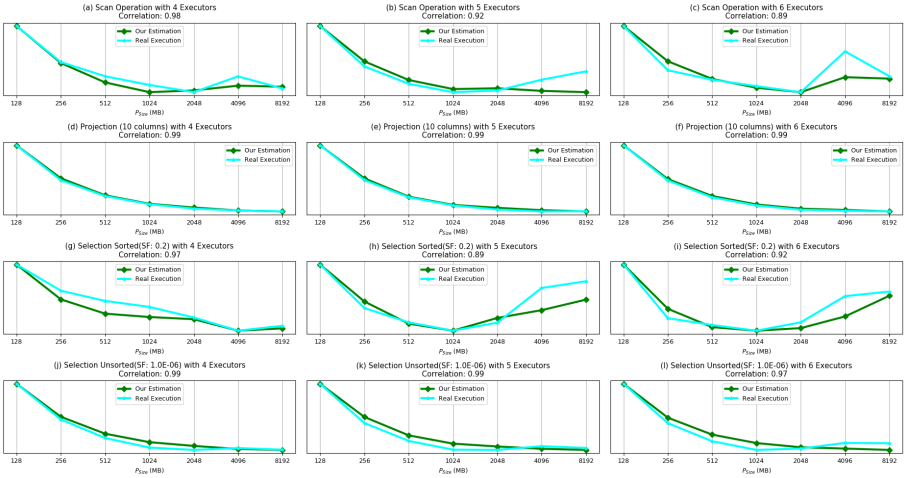


Fig. 6.5: Validation of our estimation for makespan

Cost model validation.

We generated a synthetic dataset of a very wide table with 1186 columns with different data types and 32 GB of size. We executed scan, projection with 10 referred columns, and selection with 0.2 selectivity factor to compare the real executions with our estimations. Figure 6.5 shows that comparison (notice

that, we normalized the results, both real and estimation, like $\frac{x - min}{max - min}$ to facilitate visual comparison).

Figure 6.5a, Figure 6.5b, and Figure 6.5c show the results for a scan operation with different number of executors. Similarly, Figure 6.5d, Figure 6.5e, and Figure 6.5f show the results for a projection operation with different number of executors. Finally, Figure 6.5g, Figure 6.5h, and Figure 6.5i show the accuracy of our estimations in comparison with the real executions for selection operation against sorted data. And, Figure 6.5j, Figure 6.5k, and Figure 6.5l shows the results for selection operation against unsorted data. Observe that, our estimations successfully capture the trends of real executions in almost all cases. Most of our predictions closely follow the real trends. In case of Figure 6.5c, 6.5h, and 6.5i the divergences with the real trend are due to the different units used in our estimation (i.e., these estimations are in cost units corresponding to I/Os, whereas the real executions are in seconds). Yet, the trends are predicted correctly and suffice to find the optimal partition size. The only exception is 6.5b, where we estimated a lower cost for large partition. Nevertheless, even in this case, our choice is still better than the default partition size.

We also confirm the accuracy of our estimations with the real executions using statistical correlation, which measures how well the cost model estimates are related to the real execution. In Figure 6.5, it can be seen that our estimations are highly correlated (i.e., overall Pearson correlation coefficient 0.96) to real executions.

Query	SF	Ref Cols	Similar Queries
Q1	[0.98, 0.98]	[7, 7]	-
Q3	[0.0026, 0.0056]	[5, 7]	Q8, Q12, Q17
Q10	[0.011, 0.031]	[4, 11]	Q4, Q5, Q6, Q7, Q11, Q14
Q16	[0.04, 0.08]	[2, 8]	Q2, Q13, Q15, Q18
Q20	[0.000025, 0.0007]	[5, 11]	Q19, Q21
Q22	[0.11, 0.2]	[2, 7]	Q9

Table 6.3: Representative queries of TPC-H

Performance evaluation.

In TPC-H, the widest table has only 16 columns and in TPC-DS[10], only 26. Hence, we follow [77] to generate a wide table by completely denormalizing

[10] `http://www.tpc.org/tpcds`

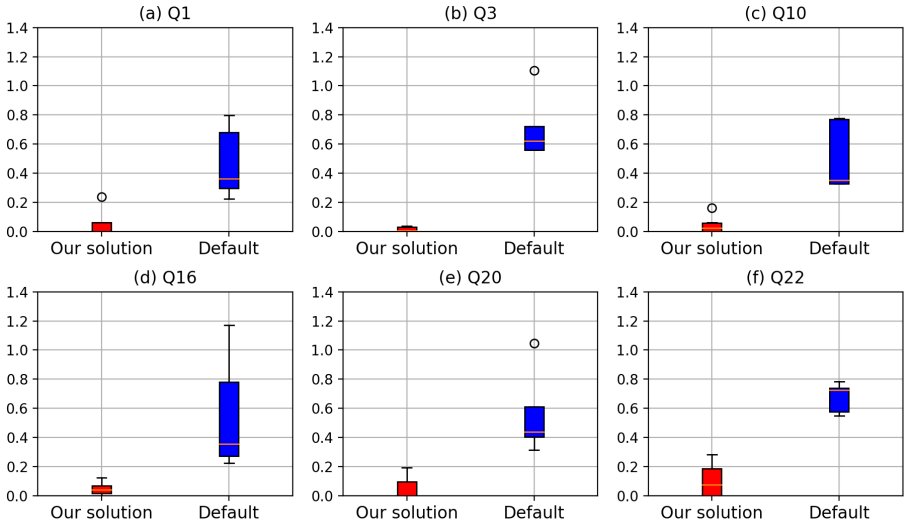


Fig. 6.6: Comparison between our configurations and default ones for TPC-H. Y-axis represents the normalized values.

all other tables in TPC-H against *lineitem*. The FROM clauses in all queries are consequently changed to the corresponding denormalized table. This results, for a scale factor $16GB$, in a denormalized table of $124GB$ being generated for the evaluation. We have chosen six representative queries based on different projected attributes and selectivity factors as shown in Table 6.3. The table shows the intervals of selectivity factor and number of referred columns of each group of queries. The other queries follow similar access patterns to those selected.

As presented earlier, there is no optimal solution in a multi-objective optimization, but there are many best solutions referred to as Pareto front. The Pareto front of our estimation is denoted as $Pareto_{Estimated}$, and the Pareto front of the real execution is denoted as $Pareto_{Real}$. It could happen that in the $Pareto_{Estimated}$, we miss some real Pareto solutions. These are referred to as $Pareto_{Missed}$. Furthermore, we have the default set of solutions — when a default partition size (i.e., 128MB) is used, denoted as $Default_{Real}$. Finally, each solution has two metrics based on our objectives, namely, *makespan* and *resource usage*.

We compute the Euclidean distance between $Pareto_{Estimated}$ and $Pareto_{Real}$ (both *makespan* and *resource usage* components are normalized to mitigate differences in the scales, resulting on a maximum distance of $\sqrt{2}$), and also to

penalize the missed solutions, we compute the distance between $Pareto_{Miss}$ and $Pareto_{Estimated}$ — all these distances compute to *Our Solution*. Furthermore, we also compute the distance between $Default_{Real}$ and $Pareto_{Real}$ — which are represented as *Default Configurations*. More precisely, the Euclidean distance is computed between each solution of one set and all the solutions of the other set, and each time the minimum distance between them is taken.

In Figure 6.6, we show the Boxplot of the distances corresponding to *Our Solution* compared to the boxplot of the distances to *Default Configurations*. We are showing the results of the representative queries (chosen based on their referred columns and selectivity factors) of TPC-H. Observe the boxplots in *Our Solution* are smaller and closer to zero distance, which means that the solutions proposed (i.e., $Pareto_{Estimated}$) are much closer to the real Pareto front (i.e., $Pareto_{Real}$) than the default configurations (i.e., $Default_{Real}$). In summary, on average we are as close as 5.6% to the real Pareto front, whereas, the default configuration is much further from the Pareto front (on average 58.2%).

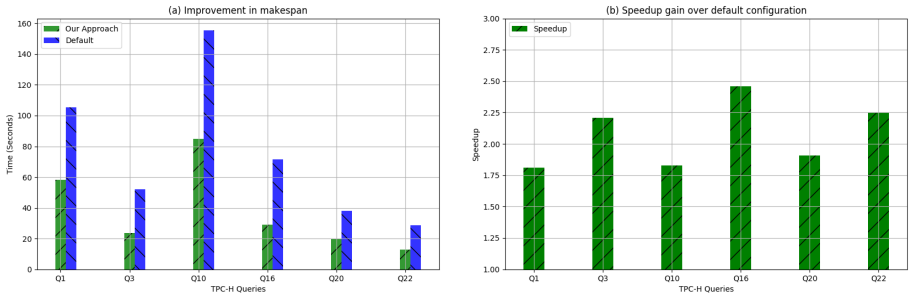


Fig. 6.7: Speedup gain for TPC-H queries

We also compare the query execution time (i.e., makespan) of our approach with the default configuration (e.g., default partition size of 128MB). As mentioned earlier, we have multiple solutions for a query and we took the minimum makespan among these solutions for comparison. Similarly, we have multiple default configurations and we took the average of their makespans. Figure 6.7a compares the makespan of TPC-H queries, which highlights the advantage of our approach over the default solutions. Likewise, we also present the speedup gain in Figure 6.7b, which is between 1.8x to 2.5x. On average, our approach provides 2.1x speedup against the default configuration.

6.6 Conclusions

Hybrid layouts are widely used to store processed data in highly distributed Big Data systems to perform ad-hoc analysis. These Big Data systems process data on a computers cluster by creating multiple tasks. Typically, they create tasks based on the total size of the table, rather than based on the reading size of the query. Moreover, always using the default configuration has a heavy impact on performance. Thus, we proposed a cost-based framework which utilizes a multi-objective approach to decide the number of tasks and executors for a given query based on the reading size. We prototyped our approach for Apache Parquet, evaluated it on TPC-H queries, and showed the improvement it provides.

7

Conclusions and Future Directions

7.1 Conclusions

Big data systems are the core of modern data analytics. These systems provide off-the-shelf tools to store and process data quickly. Yet, there is room for improvements. Typically, different users have many duplicate computations in their DIFs, which can be avoided by materializing and reusing them in the future. We provided the whole life-cycle of selecting common parts among different DIFs for materialization and storing them in the best possible storage layout for reducing their loading cost. Additionally, we also proposed optimization mechanisms for storage layouts to further improve their performance.

The main goal of this thesis is to provide a set of solutions, which do not require any modification in the existing running system. The proposed solutions can be adapted by real scenarios without any change in their infrastructure. Yet, they help in improving the execution of DIFs and saving computational resources.

In the following, we provide conclusions for each chapter, separately.

Intermediate Results Materialization Selection. In Chapter 3, we proposed a generic framework, which can take any quantifiable SLA and based on them,

find the best possible intermediate results for materialization using multi-objective optimization. We consider four SLA's metrics (i.e., loading time, storage cost, query cost, and freshness) for this study to show the importance and benefits of our approach. Freshness is considered to allow stale data in some materialized intermediate results, which help to reduce loading time for frequent update input sources. We compared our approach with an existing state of the art solution and showed that our approach provides 40% speedup on average.

Storage Format Selection. In Chapter 4, we focused particularly on the storage of materialized intermediate results. We found that the first operation has direct impact on reading from the disk. Whereas, subsequent operations read all the data produced by the first operation. Thus, only the first operation can decide to read less data from disk. Consequently, we proposed heuristic-rules based on the type of first operation and features provided by different storage layouts. We utilized the heuristic-rules in a hybrid approach to decide the storage layouts for the chosen intermediate results for materialization. The heuristic-rules are used for cold-start, when there is no statistical information available. When statistical information is available, the cost model is used to estimate the I/O cost of different storage layouts and choosing the one, which has overall minimum cost. Our detailed experiments on open source benchmarks showed the significant reduction in the loading time of materialized intermediate results and overall, it improves the execution of DIFs.

Configuring Parameters of Hybrid Layouts. In Chapter 5, we proposed a cost-based approach to configure hybrid layout's parameters and find their best possible values based on the characteristics of data and workload. The cost model requires statistical information about data and workload. We have utilized a single-column profiling technique to profile the data and job history files are used to extract the characteristics of the workload. Based on these characteristics, the best possible values are configured for hybrid layouts. Our detailed experiments showed the improvements in the execution of DIFs.

Configuring Parallelism for Hybrid Layouts. In Chapter 6, we proposed an approach to estimate the number of tasks for improving both execution time

and resource consumption. Typically, Big Data systems process data on a computers cluster by creating multiple tasks. They create tasks based on the total size of the table, rather than based on the reading size of the DIF. Moreover, always using the default configuration has a heavy impact on performance. Thus, we proposed a cost-based framework which utilizes a multi-objective approach to decide the number of tasks and executors for a given DIF. We prototyped our approach for Apache Parquet, evaluated it on TPC-H queries, and showed the improvement it provides.

Finally, in summary, this thesis provides solutions for selecting materialized intermediate results and storing them in the best possible storage layouts. It also provides solutions to optimize storage layouts to best fit for the current running workload.

7.2 Future Directions

In this thesis, we provided a way to choose the best intermediate results for materialization by considering different quantifiable SLAs. However, the management of intermediate results (when to update/delete) has not been considered in this thesis. This would help in deciding when to update and delete a materialized result, it would also help to decide when it is beneficial to update an existing materialized intermediate result.

Furthermore, the cost model proposed for storage layouts has not considered compression techniques, which are used often when storing the data to reduce their size. It would be nice to extend the cost model for estimating the size of compressed data, which would help to estimate the time needed to compress/decompress and process the data. This would help in choosing storage layouts with different compression algorithms. The extended cost model would also help in deciding when it is beneficial to compress the data, because if storage is cheaper and query response time is critical then it might be a good idea to disable compression.

Additionally, it would be nice to extend the configurable parameters list of hybrid layouts to consider more parameters for improving the execution time of DIFs. Specifically, the parameters related to compression and encoding can be very important for improving the performance of different DIFs. Moreover, the approach for controlling the degree of parallelism focuses only on improving the execution of the map phase. It would also be nice to apply

the same approach for the shuffle phase. This would help to improve the overall performance of an individual DIF. Further, we have not considered data locality when deciding the degree of parallelism.

Finally, data integration can also benefit from our proposed solutions. In data integration, data are stored in different layouts, which might needed to be virtually integrated for crossing their data. However, it is always challenging to find the optimal query plan, because different data sources have different features. Our cost model can help to estimate the cost in an individual storage layout, which would help in finding the overall best query execution plan.

Bibliography

[1] Z. Abedjan, L. Golab, and F. Naumann. “Data Profiling: A Tutorial”. In: *SIGMOD*. ACM, 2017 (cit. on pp. 94, 113).

[2] F. N. Afrati and J. D. Ullman. “Optimizing Multiway Joins in a Map-Reduce Environment”. In: *IEEE TKDE* 23.9 (2011), pp. 1282–1298 (cit. on p. 35).

[3] A. Aggarwal and J. S. Vitter. “The Input/Output Complexity of Sorting and Related Problems”. In: *ACM Commun.* 31.9 (1988), pp. 1116–1127 (cit. on p. 34).

[4] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. “Database Tuning Advisor for Microsoft SQL Server 2005”. In: *VLDB*. 2004, pp. 1110–1121 (cit. on p. 16).

[5] I. Alagiannis, S. Idreos, and A. Ailamaki. “H2O: a hands-free adaptive store”. In: *SIGMOD*. 2014, pp. 1103–1114 (cit. on pp. 4, 19, 53).

[6] Atscale. “Big Data Maturity Survey”. In: *Cloudera* (2016). `https://www.atscale.com/wp-content/uploads/2019/07/AtScale_Big_Data_Maturity_Survey_2016_-_C4c__1_.pdf` (cit. on p. 75).

[7] T. Azim, M. Karpathiotakis, and A. Ailamaki. “ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data”. In: *PVLDB* 11.3 (2017), pp. 324–337 (cit. on p. 19).

[8] L. Baldacci and M. Golfarelli. “A Cost Model for Spark SQL”. In: *IEEE TKDE* 31.5 (2019), pp. 819–832 (cit. on p. 19).

[9] H. Bian, Y. Tao, G. Jin, Y. Chen, X. Qin, and X. Du. “Rainbow: Adaptive Layout Optimization for Wide Tables”. In: *ICDE*. 2018, pp. 1657–1660 (cit. on pp. 19, 21, 104, 109).

[10] H. Bian, Y. Yan, W. Tao, L. J. Chen, Y. Chen, X. Du, and T. Moscibroda. "Wide Table Layout Optimization based on Column Ordering and Duplication". In: *SIGMOD*. 2017, pp. 299–314 (cit. on pp. 3, 4, 19, 21, 86, 97, 104, 117).

[11] J. Camacho-Rodriguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. "Reuse-based Optimization for Pig Latin". In: *CIKM*. 2016, pp. 2215–2220 (cit. on p. 17).

[12] A. F. Cardenas. "Analysis and Performance of Inverted Data Base Structures". In: *ACM Commun.* 18.5 (1975), pp. 253–263 (cit. on p. 68).

[13] A. F. Cardenas. "Analysis and Performance of Inverted Data Base Structures". In: *ACM Commun.* 18.5 (1975) (cit. on p. 91).

[14] Y. Chen, S. Alspaugh, and R. H. Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads". In: *PVLDB* 5.12 (2012), pp. 1802–1813 (cit. on pp. 3, 26, 46, 52).

[15] T. Chiba and T. Onodera. "Workload characterization and optimization of TPC-H queries on Apache Spark". In: *ISPASS*. 2016, pp. 112–121 (cit. on p. 21).

[16] G. Dasarathy. *A Simple Probability Trick for Bounding the Expected Maximum of n Random Variables*. Tech. rep. Arizona State University, 2011 (cit. on p. 112).

[17] A. Davidson and A. Or. *Optimizing Shuffle Performance in Spark*. Tech. rep. UC Berkeley, 2013 (cit. on p. 23).

[18] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *ACM Commun.* 51.1 (2008) (cit. on pp. 1, 26, 86, 104).

[19] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE TEC* 6.2 (2002), pp. 182–197 (cit. on pp. 105, 115).

[20] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. "Split query processing in polybase". In: *SIGMOD*. 2013, pp. 1255–1266 (cit. on p. 19).

[21] I. Elghandour and A. Aboulnaga. "ReStore: Reusing Results of MapReduce Jobs". In: *PVLDB* 5.6 (2012), pp. 586–597 (cit. on pp. 17, 26, 28).

[22] A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Çetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. G. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. "A Demonstration of the BigDAWG Polystore System". In: *PVLDB* 8.12 (2015), pp. 1908–1911 (cit. on p. 19).

[23] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. "SAP HANA database: data management for modern business applications". In: *SIGMOD Record* 40.4 (2011), pp. 45–51 (cit. on p. 18).

[24] M. Ferreira, J. Paiva, M. Bravo, and L. E. T. Rodrigues. "SmartFetch: Efficient Support for Selective Queries". In: *CloudCom*. IEEE Computer Society, 2015 (cit. on p. 21).

[25] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. "Column-Oriented Storage Techniques for MapReduce". In: *PVLDB* 4.7 (2011), pp. 419–429 (cit. on p. 53).

[26] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)* Pearson Education, 2009 (cit. on p. 33).

[27] A. Gounaris and J. Torres. "A Methodology for Spark Parameter Tuning". In: *Big Data Research* 11 (2018), pp. 22–32 (cit. on p. 21).

[28] O. Grodzevich and O. Romanko. "Normalization and other topics in multi-objective optimization". In: *FMIPW*. 2006, pp. 42–56 (cit. on p. 41).

[29] H. Gupta and I. S. Mumick. "Selection of Views to Materialize in a Data Warehouse". In: *IEEE TKDE* 17.1 (2005), pp. 24–43 (cit. on p. 26).

[30] R. Halasipuram, P. M. Deshpande, and S. Padmanabhan. "Determining Essential Statistics for Cost Based Optimization of an ETL Workflow". In: *EDBT*. 2014, pp. 307–318 (cit. on p. 33).

[31] A. Y. Halevy. "Answering queries using views: A survey". In: *VLDB J.* 10.4 (2001), pp. 270–294 (cit. on pp. 16, 27).

[32] J. Han, H. Cheng, D. Xin, and X. Yan. "Frequent pattern mining: current status and future directions". In: *Springer Data Min. Knowl. Discov.* 15.1 (2007) (cit. on p. 94).

[33] V. Harinarayan, A. Rajaraman, and J. D. Ullman. "Implementing Data Cubes Efficiently". In: *SIGMOD*. 1996, pp. 205–216 (cit. on p. 26).

[34] H. Herodotou and S. Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *PVLDB* 4.11 (2011), pp. 1111–1122 (cit. on p. 21).

[35] H. Herodotou, F. Dong, and S. Babu. “No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics”. In: *SOSP*. 2011, p. 18 (cit. on pp. 23, 104).

[36] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. “Starfish: A Self-tuning System for Big Data Analytics”. In: *CIDR*. 2011 (cit. on p. 21).

[37] K. Herrmann. “Multi-Schema-Version Data Management”. English. TUD Ph.D. Supervisor: Prof. Dr.-Ing. Wolfgang Lehner, Technische Universität Dresden, Dresden, Germany AAU Ph.D. Supervisor: Prof. Torben Bach Pedersen, Aalborg University, Aalborg, Denmark. PhD thesis. 2017 (cit. on p. 20).

[38] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. “Opening the Black Boxes in Data Flow Optimization”. In: *PVLDB* 5.11 (2012), pp. 1256–1267 (cit. on p. 35).

[39] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. “Here are my Data Files. Here are my Queries. Where are my Results?” In: *CIDR*. 2011, pp. 57–68 (cit. on p. 19).

[40] M. T. Islam, S. Karunasekera, and R. Buyya. “dSpark: Deadline-Based Resource Allocation for Big Data Applications in Apache Spark”. In: *e-Science*. 2017, pp. 89–98 (cit. on pp. 23, 104).

[41] A. Jindal, J. Quiané-Ruiz, and J. Dittrich. “Trojan data layouts: right shoes for a running elephant”. In: *SOCC*. 2011, p. 21 (cit. on pp. 19, 53, 60, 70, 80).

[42] A. Jindal, J. Quiané-Ruiz, and J. Dittrich. “WWHow! Freeing Data Storage from Cages”. In: *CIDR*. 2013 (cit. on p. 19).

[43] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. “Computation Reuse in Analytics Job Service at Microsoft”. In: *SIGMOD*. 2018, pp. 191–203 (cit. on pp. 3, 52).

[44] A. Jindal, K. Karanasos, S. Rao, and H. Patel. “Selecting Subexpressions to Materialize at Datacenter Scale”. In: *PVLDB* 11.7 (2018), pp. 800–812 (cit. on pp. 3, 52).

[45] P. Jovanovic, O. Romero, and A. Abelló. “A Unified View of Data-Intensive Flows in Business Intelligence Systems: A Survey”. In: *Springer TLSDKCS* 29 (2016), pp. 66–107 (cit. on p. 26).

[46] P. Jovanovic, A. Simitsis, and K. Wilkinson. “Engine independence for logical analytic flows”. In: *ICDE*. 2014, pp. 1060–1071 (cit. on p. 30).

[47] P. Jovanovic, O. Romero, T. Calders, and A. Abelló. “H-WorD: Supporting Job Scheduling in Hadoop with Workload-Driven Data Redistribution”. In: *ADBIS*. 2016, pp. 306–320 (cit. on pp. 109, 116).

[48] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. “Incremental Consolidation of Data-Intensive Multi-flows”. In: *IEEE TKDE*. 2016 (cit. on pp. 42, 78).

[49] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. “Incremental Consolidation of Data-Intensive Multi-Flows”. In: *IEEE TKDE* 28.5 (2016), pp. 1203–1216 (cit. on p. 31).

[50] V. Kalavri, H. Shang, and V. Vlassov. “m2r2: A Framework for Results Materialization and Reuse in High-Level Dataflow Systems for Big Data”. In: *CSE*. 2013, pp. 894–901 (cit. on p. 17).

[51] T. Kathuria and S. Sudarshan. “Efficient and Provable Multi-Query Optimization”. In: *PODS*. 2017, pp. 53–67 (cit. on p. 16).

[52] S. Kipp. “Will SSD replace HDD?” `http://www.ieee802.org/3/CU4HDDSG/public/sep15/Kipp_CU4HDDsg_01a_0915.pdf`. 2015 (cit. on p. 62).

[53] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff. “Lynx: a learning linux prefetching mechanism for SSD performance model”. In: *NVMSA*. 2016, pp. 1–6 (cit. on p. 62).

[54] Y. Li and J. M. Patel. “WideTable: An Accelerator for Analytical Data Processing”. In: *PVLDB* 7.10 (2014), pp. 907–918 (cit. on pp. 4, 86, 97, 104, 117).

[55] I. Mami and Z. Bellahsene. “A survey of view selection methods”. In: *SIGMOD Record* 41.1 (2012), pp. 20–29 (cit. on p. 16).

[56] R. T. Marler and J. S. Arora. “Survey of multi-objective optimization methods for engineering”. In: *Springer Struct. Multidiscipl. Optim.* 26.6 (2004), pp. 369–395 (cit. on p. 32).

[57] G. Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *VLDB*. 1998, pp. 476–487 (cit. on p. 86).

[58] R. F. Munir, A. Abelló, O. Romero, M. Thiele, and W. Lehner. "ATUN-HL: Auto Tuning of Hybrid Layouts using Workload and Data Characteristics". In: *ADBIS*. 2018, pp. 200–215 (cit. on pp. 104, 108, 109).

[59] R. F. Munir, O. Romero, A. Abelló, B. Bilalli, M. Thiele, and W. Lehner. "ResilientStore: A Heuristic-Based Data Format Selector for Intermediate Results". In: *MEDI*. 2016, pp. 42–56 (cit. on p. 4).

[60] P. P. Nghiem and S. M. Figueira. "Towards efficient resource provisioning in MapReduce". In: *JPDC* 95 (2016), pp. 29–41 (cit. on p. 22).

[61] T. Nguyen, S. Bimonte, L. d'Orazio, and J. Darmont. "Cost models for view materialization in the cloud". In: *EDBT/ICDT Workshops*. 2012, pp. 47–54 (cit. on p. 35).

[62] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. "MRShare: Sharing Across Multiple Queries in MapReduce". In: *PVLDB* 3.1 (2010), pp. 494–505 (cit. on pp. 17, 26).

[63] P. Petridis, A. Gounaris, and J. Torres. "Spark Parameter Tuning via Trial-and-Error". In: *INNS*. 2016, pp. 226–237 (cit. on p. 21).

[64] W. Qu and S. Dessloch. "A Real-time Materialized View Approach for Analytic Flows in Hybrid Cloud Environments". In: *Datenbank-Spektrum* 14.2 (2014), pp. 97–106 (cit. on p. 17).

[65] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. "DB2 with BLU Acceleration: So Much More than Just a Column Store". In: *PVLDB* 6.11 (2013), pp. 1080–1091 (cit. on p. 18).

[66] A. Roukh, L. Bellatreche, A. Boukorca, and S. Bouarar. "Eco-DMW: Eco-Design Methodology for Data warehouses". In: *DOLAP*. 2015, pp. 1–10 (cit. on p. 35).

[67] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization". In: *SIGMOD*. 2000, pp. 249–260 (cit. on p. 16).

[68] K.-U. Sattler. "Data Quality Dimensions". In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, 2009, pp. 612–615 (cit. on p. 37).

[69] M. Schaarschmidt, F. Gessert, and N. Ritter. "Towards Automated Polyglot Persistence". In: *BTW*. 2015, pp. 73–82 (cit. on p. 19).

[70] T. K. Sellis. “Multiple-Query Optimization”. In: *ACM TODS* 13.1 (1988), pp. 23–52 (cit. on p. 16).

[71] K. V. Shvachko. “HDFS scalability: the limits to growth”. In: *Login* 35.2 (2010), pp. 6–16 (cit. on pp. 1, 52, 86, 104).

[72] S. Sidhanta, W. M. Golab, and S. Mukhopadhyay. “OptEx: A Deadline-Aware Cost Optimization Model for Spark”. In: *CCGrid*. 2016, pp. 193–202 (cit. on pp. 23, 104).

[73] Y. N. Silva, P. Larson, and J. Zhou. “Exploiting Common Subexpressions for Cloud Query Processing”. In: *ICDE*. 2012, pp. 1337–1348 (cit. on p. 16).

[74] A. Simitsis and K. Wilkinson. “Revisiting ETL Benchmarking: The Case for Hybrid Flows”. In: *TPCTC*. 2012, pp. 75–91 (cit. on p. 35).

[75] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. “QoX-driven ETL design: reducing the cost of ETL consulting engagements”. In: *SIGMOD*. 2009, pp. 953–960 (cit. on p. 26).

[76] M. Skala. “Hypergeometric tail inequalities: ending the insanity”. In: *CoRR* abs/1311.5939 (2013) (cit. on p. 111).

[77] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. “Fine-grained partitioning for aggressive data skipping”. In: *SIGMOD*. ACM, 2014 (cit. on pp. 21, 94, 97, 118).

[78] L. Sun, M. J. Franklin, J. Wang, and E. Wu. “Skipping-oriented Partitioning for Columnar Layouts”. In: *PVLDB* 10.4 (2016) (cit. on pp. 21, 94).

[79] L. Thai, B. Varghese, and A. Barker. “Budget Constrained Execution of Multiple Bag-of-Tasks Applications on the Cloud”. In: *CoRR* abs/1507.05467 (2015) (cit. on pp. 23, 104).

[80] D. Theodoratos and M. Bouzeghoub. “A General Framework for the View Selection Problem for Data Warehouse Design and Evolution”. In: *DOLAP*. 2000, pp. 1–8 (cit. on pp. 29, 30).

[81] D. Theodoratos and T. K. Sellis. “Data Warehouse Configuration”. In: *VLDB*. 1997, pp. 126–135 (cit. on p. 30).

[82] D. Theodoratos and T. K. Sellis. “Dynamic Data Warehouse Design”. In: *DaWaK*. 1999, pp. 1–10 (cit. on p. 30).

[83] A. Verma, L. Cherkasova, and R. H. Campbell. “Resource Provisioning Framework for MapReduce Jobs with Performance Goals”. In: *USENIX*. 2011, pp. 165–186 (cit. on p. 22).

[84] G. Wang and C. Chan. “Multi-Query Optimization in MapReduce Framework”. In: *PVLDB* 7.3 (2013), pp. 145–156 (cit. on pp. 17, 26).

[85] W. Wu, W. Lin, C. Hsu, and L. He. “Energy-efficient hadoop for big data analytics and computing: A systematic review and research insights”. In: *Elsevier FGCS* 86 (2018), pp. 1351–1367 (cit. on p. 23).

[86] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. “Efficient exploitation of similar subexpressions for query processing”. In: *SIGMOD*. 2007, pp. 533–544 (cit. on p. 16).