OctopusDB: Flexible and Scalable Storage Management for Arbitrary Database Engines

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Alekh Jindal
Saarbrücken

July 10, 2012

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr. Mark Groves
Vorsitzender der Prüfungskommission	Prof. Dr. Sebastian Hack
Berichterstatter	Prof. Dr. Jens Dittrich
Berichterstatter	Prof. Dr. Gerhard Weikum
Berichterstatter	Prof. Dr. Anastasia Ailamaki
Beisitzer	Dr. Jorge Quiané
Tag des Promotionskollquiums	24.08.2012

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Ort, Datum

(Unterschrift)

To my mother and father

Acknowledgements

Jens Dittrich has been my guiding force over the last two and a half years. He has a unique disruptive style of doing research, which has fascinated me tremendously. Under his supervision, I had the opportunity to work on some very exciting problems. I have learned many things from him including questioning earlier assumptions, doing things differently each time, and aiming for no less than the very best. All these made this thesis an enjoyable research experience for me. His high standards for quality were truly inspiring and he was a pillar of continuous motivation and never ending support all along. It has been a great learning curve working with him and I would like to thank him for everything.

I have deep high regards for Jorge Quiané. He has been my mentor and friend. I have worked with him on several projects and I highly appreciate his team player and hard working attitude. He has always given me the confidence to take bigger leaps. And his insightful feedback and detailed comments were of great help all along. I would like to express him my sincere gratitude for his kindness and support.

I would like thank the reviewers Prof. Gerhard Weikum and Prof. Anastasia Ailamaki for kindly agreeing to review this thesis, and for their remarks and feedback. I have profound admiration for both of them and it is an honor to have them on the examination board of this thesis. Prof. Gerhard Weikum has also often given me professional advice from the depth of his experiences. Talking to him has always widened my scope of thinking and I would like to thank him for giving me his valuable time. I would also like to thank Prof. Sebastian Hack for agreeing to chair my defense as well as for agreeing to be my scientific companion during this thesis.

The research work presented in this thesis was partly supported by the funding from Multimodal Computing and Interaction (M2CI) Cluster of Excellence. The remaining financial support came from German Ministry of Education and Science. I am thankful to them for their support.

I am also extremely thankful to our secretarial staff, Angelika Scholl-Danopoulos, for helping me with the administrative issues throughout my work.

I would like to thank Jörg Schad for helping me with the german translation of the abstract. I would also thank Stefan Richter for revising the german translation and for proof-reading certain parts of this thesis.

I made some great friends here in Germany and I am thankful to them for the great time I had outside work. Jörg Schad has been a great friend, pulling me into all kinds of sports here. Mohammad and Jana have been great people to spend the weekend with. And Sharath has always been around, smiling and cheerful.

Finally, I would like to thank my family — my parents who have been a great source of inspiration and purpose for me, my sister who has tremendous faith in me, and my wife who is truly my better half.

Above all, God has been gracious to me and I am grateful for the luck that he has bestowed upon me.

Abstract

We live in a dynamic age with the economy, the technology, and the people around us changing faster than ever before. Consequently, the data management needs in our modern world are much different than those envisioned by the early database inventors in the 70s. Today, enterprises face the challenge of managing ever-growing dataset sizes with dynamically changing query workloads. As a result, modern *data managing systems*, including relational as well as big data management systems, can no longer afford to be *carved-in-stone* solutions. Instead, data managing systems must inherently provide flexible data management techniques in order to cope with the constantly changing business needs. The current practice to deal with changing query workloads is to have a different specialized product for each workload type, e.g. row stores for OLTP workload, column stores for OLAP workload, streaming systems for streaming workload, and scan-oriented systems for shared query processing. However, this means that the enterprises have to now glue different data managing products together and copy data from one product to another, in order to support several query workloads. This has the additional penalty of managing a zoo of data managing systems in the first place, which is tedious, expensive, as well as counter-productive for modern enterprises.

This thesis presents an alternative approach to supporting several query workloads in a data managing system. We observe that each specialized database product has a different data store, indicating that different query workloads work well with different data layouts. Therefore, a key requirement for supporting several query workloads is to support several data layouts. Therefore, in this thesis, we study ways to *inject* different data layouts into existing (and familiar) data managing systems. The goal is to develop a flexible storage layer which can support several query workloads in a single data managing system. We present a set of *non-invasive* techniques, coined *Trojan Techniques*, to inject different data layouts into a data managing system. The core idea of Trojan Techniques is to drop the assumption of having one fixed data store per data managing system. Trojan Techniques are non-invasive in the sense that they do not make heavy untenable changes to the system. Rather, they affect the data managing system from inside, almost at the core. As a result, Trojan Techniques bring significant improvements in query performance. It is interesting to note that in our approach we follow a *design pattern* that has been used in other non-invasive research works as well, such as PAX, fractal prefetching B^+-trees, and RowCol.

We propose four Trojan Techniques. First, *Trojan Indexes* add an additional index access path in Hadoop MapReduce. Second, *Trojan Joins* allow for co-partitioned joins in Hadoop MapReduce. Third, *Trojan Layouts* allow for row, column, or column-grouped layouts in Hadoop MapReduce. Together, these three techniques provide a highly flexible data storage layer for Hadoop MapReduce. Our final proposal, *Trojan Columns*, introduces columnar functionality in row-oriented relational databases, including closed source commercial databases, thus bridging the gap between row and column oriented databases. Our experimental results show that Trojan Techniques can improve the performance of Hadoop MapReduce by a factor of up to 18, and that of a top-notch commercial database product by a factor of up to 17.

Zusammenfassung

Wir leben in einer dynamischen Zeit, in der sich Wirtschaft, Technologie und Gesellschaft schneller verändern als jemals zuvor. Folglich unterscheiden sich die Anforderungen an Datenverarbeitung heute sehr von dem, was sich die Pioniere dieses Forschungsgebiets in den 70er Jahren ursprünglich ausgemalt hatten. Heutzutage sehen sich Firmen mit der Herausforderung konfrontiert, stark fluktuierende Anfragelasten über einer stetig wachsender Datenmengen zu bewältigen. Daher können es sich moderne Datenbanksysteme, sowohl relationale als auch Big Data Systeme, nicht mehr leisten, wie starre, in Stein gemeißelte Lösungen zu funktionieren. Stattdessen sollten moderne Datenbanksysteme von Grunde auf für flexible Datenverwaltung konzipiert werden, um mit sich ständig ändernden Anforderungen Schritt halten zu können. Die gegenwärtige Praxis im Umgang mit häufig wechselnden Anfragemustern besteht allerdings noch darin, jeweils unterschiedliche, spezialisierte Lösungen für die verschiedenen Anfragetypen zu nutzen - zum Beispiel zeilenorientierte Systeme für OLTP Anfragen, spaltenorientierte Systeme für OLAP Anfragen, Data Stream Management Systeme für kontinuierliche Datenströme und Scan-basierte Systeme für die Bearbeitung von vielen gleichzeitigen Anfragen. Leider setzt dieses Vorgehen aber voraus, dass die Unternehmen es schaffen die verschiedensten Systeme irgendwie miteinander zu verknüpfen und einen Datenaustausch zwischen ihnen zu gewährleisten. Ein zusätzlicher Nachteil ist, dass hierbei oft ein ganzes Sortiment von Datenbankprodukten eingerichtet und gepflegt werden muss, was sowohl zeit- als auch kostenintensiv und damit letztlich aufwendig ist.

Diese Dissertation präsentiert eine alternative Lösung, um wechselnde Anfragemuster effizient mit einem einzigen Datenverwaltungssystem zu unterstützen. Aus der Beobachtung, dass jedes spezielle Datenbankprodukt unterschiedliche Ansätze zur Daten-speicherung nutzt, folgern wir, dass verschiedene Anfragen jeweils auf bestimmten Datenlayouts effizienter beantwortet werden können als auf anderen. Deshalb ist eine zentrale Anforderung zur effizienten Verarbeitung unterschiedlicher Anfragetypen mit nur einem System, dass dieses System verschiedene Datenlayouts unterstützen muss. Dazu untersuchen wir in dieser Arbeit Möglichkeiten, um verschiedene Datenlayouts nachträglich in bestehende (und bekannte) Datenbanksysteme einzuschleusen. Das Ziel hierbei ist die Entwicklung einer flexiblen Speicherschicht, die verschiedenste Anfragen in einem einzigen Datenbanksystem unterstützen kann. Wir haben hierzu eine Reihe von nichtinvasiven Techniken, auch Trojanische Techniken genannt, entwickelt, mit denen sich verschiedene Datenlayouts nachträglich in existierende Systeme einschleusen lassen. Die Grundidee hinter diesen Trojanischen Techniken ist es, die Annahme, dass jedes Datenbanksystem nur eine festgelegte Art der Datenspeicherung haben kann, fallen zu lassen. Die Trojanischen Techniken erfordern nur minimale Änderungen am ursprünglichen Datenbanksystem, sondern beeinflussen dessen Verhalten von innen heraus. Der Einsatz Trojanischen Techniken kann die Anfragegeschwindigkeit erheblich steigern. Wir folgen mit diesem Ansatz einem Entwurfsmuster, das auch in anderen nichtinvasiven Forschungsprojekten wie PAX, fpB$^+$-Bäume und RowCol verwendet wurde.

Wir stellen in dieser Arbeit vier verschiedene Trojanische Techniken vor. Als erstes zeigen wir, wie Trojanische Indexe die Integration eines Index in Hadoop MapReduce ermöglichen. Ergänzt wird dies durch Trojanische Joins, welche kopartitionierte Joins in Hadoop MapReduce ermöglichen. Danach zeigen wir, wie Trojanische Layouts Hadoop MapReduce um zeilen-, spalten- und gruppierte spaltenorientierte Datenlayouts erweitern. Zusammen bilden diese Techniken eine flexible Speicherschicht für das Hadoop MapReduce Framework. Unsere vierte Technik, Trojanische Spalten, erlaubt es uns, spaltenorientierte Datenverarbeitung nachträglich in zeilenbasierten Datenbanksysteme einzuführen und lässt sich sogar auf kommerzielle closed-source Produkten anwenden. Wir schließen damit die Lücke zwischen zeilen- und spaltenorientierten Datenbanksystemen. In unseren Experimenten zeigen wir, dass die Trojanischen Techniken die Leistung des Hadoop MapReduce Frameworks um das bis zu 18fache und die Geschwindigkeit einer aktuellen kommerziellen Datenbank um das 17fache erhöhen können.

Contents

Chapter 1

Introduction

1.1 Motivation

We live in a dynamic age. Whether it is economy, technology, or people, the world around us is changing faster than ever before. Consequently, the data management needs in our modern world are much different than those envisioned by the early database inventors in the 70s. In our modern world, we are witnessing businesses which are flexible and scale-up quickly, and user behaviors which are rapidly changing and harder to predict. For example, Facebook grew into almost a 100 billion dollar company in just 8 years, all along transforming the way people communicate socially. Such rapid growth leads to the task of managing ever-growing dataset sizes with dynamically changing data access/update patterns (query workloads). As a result, today, *data managing systems*, including relational as well as big data management systems, can no longer afford to be *carved-in-stone* solutions offered by businesses to users. Instead, the data managing systems must provide inherently flexible data management techniques in order to cope with the constantly changing business needs.

To deal with ever growing dataset sizes, MapReduce has emerged as a popular data processing framework recently. MapReduce allows businesses to process their data at massive scales. However, MapReduce systems, such as Hadoop, have a fixed data processing pipeline with a hard-coded physical query execution plan. This means that all queries are executed in exactly the same manner and the system cannot adapt to different or changing workloads. In other words, even though we achieve scalability, we loose heavily on performance. Traditional databases (relational row-stores), on the other hand, offer very good query performance. To do so, they rely on physical design tools. The current practice is to tune the database (either automatically or with the help of an administrator) every time it sees different data or a different query workload. However, traditional databases are optimized for a specific workload, e.g. OLTP, OLAP, data streams, scan-oriented processing etc, at the core. As a result, they have several hard-coded decisions, such as a fixed data store, which cannot be changed using a physical design tool. Therefore, traditional databases cannot handle mixed or changing query workloads very well.

To deal with different query workloads, the current practice is to have a different specialized database product or an ad-hoc system for each workload type, e.g. row stores for OLTP workload, column stores for OLAP workload, streaming systems for streaming

workload, scan-oriented systems for shared query processing. Thus, in order to support mixed and changing query workloads, an enterprise user has to now glue different database products together and copy data from one product to another. As an example, complex ETL-style data pipelines are used to copy data from different sources into a data warehouse for reporting applications. This has the additional penalty of managing a zoo of database systems in the first place. Moreover, both traditional as well as specialized database products are not capable of handling massive dataset sizes.

This thesis takes an alternative approach to support several query workloads in a data managing system. We observe that each specialized database product has a different data store, indicating that different query workloads work well with different data layouts. Therefore, a key requirement for efficiently supporting several query workloads is to support several data layouts (physical data representation). Obviously, we do not want to create a zoo of systems, nor to overwhelm the user with a highly complex one-size-fits-all data managing system. Instead, we want to discover ways to *inject* different data layouts into existing (and familiar) data managing systems. Thus, the goal of this thesis is to develop a flexible storage layer which can support several query workloads in a single data managing system. We present a set of *non-invasive* techniques, coined *Trojan Techniques*[1], to inject different data layouts into an existing data managing system, which could be a relational data management system or a MapReduce data processing system.

In the following, first we discuss the techniques to handle dynamic query workloads in Section 1.2. Then, we present an overview of recent advances in Hadoop MapReduce in Section 1.3. We list the contributions made in this dissertation in Section 1.5.

1.2 Techniques for Dynamic Workloads

1.2.1 Physical Database Design & Tuning

Physical database designs have been researched heavily in the past [9, 10, 19, 21, 43, 84, 95, 100, 134]. As a consequence, nowadays, most DBMSs offer design advisory tools [8, 12, 135]. These tools help DBAs in defining indexes, e.g. [29], as well as horizontal and/or vertical partitions, e.g. [9, 60]. The idea of these tools is to analyze the workload at a given point in time and suggest different physical designs. These suggestions are computed by a what-if analysis. What-if analysis explores the possible physical designs. However, just finding the right set of partitions is NP-hard [109]. Therefore the search space must be pruned using suitable heuristics, i.e. typically some greedy-strategy [10]. The cost of each candidate configuration is then estimated using an existing cost-based optimizer, i.e. the optimizer is tricked into believing that the candidate configuration already exists. Eventually, a suitable partitioning strategy is proposed to the DBA who then has to re-partition the existing database accordingly.

The traditional physical database design described above is an *offline* process. The DBA will only reconsider the current physical design at certain points in time. This

[1]The term "Trojan" may mean many things such as an automobile, a singing group, a record label, a person from Villarrobledo (Spain), a computer virus, or the nuclear Power Plant in Oregon. However, in this thesis, by "Trojan" we refer to the *Trojan Horse* way of affecting deep changes *inside*, while leaving everything from the *outside* intact. Still, in contrast to Trojan Horse and Trojan viruses, the Trojans introduced in this thesis are *good* ones and injected into the different systems to improve these systems.

is problematic. Assume the workload changes over time, e.g. changes in the workload due to new database applications, an increasing dataset size, or an increasing number of queries. In these situations the existing physical design decisions should be revisited to improve query times. This might involve a human — the DBA — to trigger an advisory tool with the most recent query logs and eventually decide the new physical design. Involving a DBA is especially problematic if the database system has to handle bursts and peaks. For instance consider (i) a ticket system selling 10 million Rolling Stones tickets within three days; (ii) an online store such as Amazon selling much higher volumes before christmas; or (iii) an OLAP system having to cope with new query patterns. In these types of applications it is not acceptable for users to wait for the DBA and the advisory tool to reconfigure the system. If the system stalls due to a peak workload, the application provider may loose a lot of money [128].

To address the above concerns, several online and continuous physical design tuning techniques have been proposed in recent times [20, 111]. For instance, dynamic materialized views [134] materialize the frequently accessed rows dynamically. Similarly, database cracking [63, 71–74] and adaptive merging [58, 59] have emerged as two major technologies for adaptive indexing. Likewise, AutoStore [79] applies online vertical partitioning to cope with dynamically changing query workloads.

The above approaches to physical database design still have several practical problems in dealing with changing or mixed query workloads. This is because even though, theoretically, DBMSs have a clear separation between the logical and the physical database design, physical data independence is not entirely true in practice [9]. As a result, existing database stores encapsulate just the data layouts, while further physical database tunings like partitioning are implemented on top by the query layer. This mixes the logical and physical design and affects the database administrators, the application programmers as well as the query workload performance. For example, in order to partition a table vertically, a database administrator needs to create and load multiple tables — one for each partition — in the database store. To change the vertical partitioning, the administrator needs to either create new partition tables or modify the existing ones. At query time, internally, the database system needs to retrieve data from multiple tables, typically using join operators. The basic problem here is that the database system is not aware of the multiple tables being just different vertical partitions of the same logical table. Similarly, a recent technique mimics a column store within a row store [18]. However, again, it works at the schema level and thus it is not efficient.

1.2.2 Different Workload, Different System

Several papers have claimed that one size does not fit all, i.e. we cannot have a single system for different query workloads. It started with [53] who noted that DBMSs do not work well for decision support system (DSS)-type workloads. This work lead to one of the first column-oriented data warehouses: SybaseIQ [121]. Later on, other authors supported the idea of different types of database systems for different markets as well [115, 117]. This split the landscape into several different data management systems such as Online Transaction Processing (transactional row-store), Online Analytical Processing (read-only column-store) and Data Stream Management System (continuous window queries on unbounded streams) which originated from append-only databases [122]. As a result, today, we have a plethora of data management systems, each tailored to a

specific use-case application. These include column stores e.g. Vectorwise [129], streaming systems e.g. StreamBase [120], key-value stores e.g. BerkeleyDB [14], document stores e.g. MongoDB [91], graph databases e.g. Neo4j [96], object databases e.g. ObjectDB [98], OLTP-OLAP hybrid databases e.g. [82], array databases e.g. SciDB [112], in-memory databases e.g. VoltDB [130], scan-oriented systems e.g. QPipe [65], and databases on flash devices e.g. Hyder [15].

With the above approach, we created a range of niche database products, each suited for a different application. For an enterprise, this means that a change in query workload typically leads to a change in the database product. Thus, in order to cope with different query workloads, the enterprise ends up managing a zoo of database products. This can be very tedious, expensive, as well as unproductive for the enterprise.

1.2.3 Trojan Techniques

In this thesis, we introduce Trojan Techniques for data management. The core idea of Trojan Techniques is to challenge the assumption of a fixed data store in a data managing system, which could be a relational database or Hadoop MapReduce or any other data management system. Instead, Trojan Techniques create arbitrary physical representations of data (such as layouts and indexes), termed *Storage Views*, into an *existing* data managing system i.e. without coding a new system from scratch. This is done by identifying and injecting a number of user defined functions (UDFs) into a given data managing system. Trojan Techniques use these UDFs as a mapping from logical data view to its arbitrary physical representation. The data managing systems call back these UDFs during query processing. As a result, Trojan Techniques bring significant improvements in query performance. Note that UDFs have been there since long, not just in data managing systems, but also in operating systems, middleware, and programming runtime environments. However, Trojan Techniques exploit the UDFs in a novel way — to develop flexible storage layer in existing data managing systems. Finally, Trojan Techniques are *Trojan* in the sense that they do not make heavy untenable changes into the system (i.e. non-invasive) and yet they affect the data managing system from inside, right at the core. It is interesting to note that in our approach we follow a *design pattern* that has been used in other non-invasive research works as well:

- PAX [11] reorganized the internal layout of a page without changing the outside DBMS.
- fractal prefetching B^+-trees [32] changed the internal representation of a B^+-tree node without changing its implementation.
- and RowCol [18] emulates column stores without changing the underlying row store.

Trojan Techniques are a part of a bigger vision of a highly flexible data managing system [48]. That vision, aptly coined *OctopusDB*[2] because of the inherent system

[2]An octopus may adapt to its surroundings using a camouflage unmatched by any other species on earth: it may change both the color *and* the texture of its skin. Additionally, some octopus species may even mimic movements and shape thus *impersonating* other species, e.g. http://marinebio.org/species.asp?id=260

flexibility, envisages to store data as a logical journal of data operations in the beginning. Thereafter, depending on the workload, OctopusDB creates arbitrary physical representations (Storage Views) of that journal. As a result of this flexible data storage layer, OctopusDB can mimic a variety of systems and efficiently support dynamic query workloads.Trojan Techniques are the first steps towards this vision. We do not claim to be building a fully functional one-size-fits-all database system using Trojan Techniques. However, by building flexible storage layers in existing data managing systems, we do push the boundaries of one-size-fits-all reality. Note that even though the OctopusDB vision was proposed in the context of relational databases, in this thesis we focus more on implementing this vision in Hadoop MapReduce. This is because Hadoop MapReduce was in a much more nascent state at the beginning of this thesis and hence the impact was much more visible. Still, Trojan Techniques could be applied to other data managing systems by adapting them to the system specific UDF interfaces. In order to prove the general applicability of Trojan Techniques, we present one Trojan Technique in relational databases as well. To understand why Trojan Techniques are well suited for Hadoop MapReduce, let us walk through the research advances in Hadoop MapReduce in chronological order below.

1.3 Hadoop MapReduce Overview

Google invented MapReduce as a framework for large scale data analysis in 2004 [46]. MapReduce is inspired from the functional programming concepts of the 60s. Using MapReduce, developers can describe their analysis tasks simply by using two functions: `map` and `reduce`. Everything else including parallelization, replication, and failover will then be handled by the MapReduce framework. Thus, MapReduce allows even non-expert users to run complex analytical tasks on very large clusters and clouds, without having to know about DBMS technology like SQL, data partitioning synchronization, load distribution, failure handling, indexing, data schemas, and so on. This is in sharp contrast to parallel DBMSs (PDBMS) which require advanced knowledge from the user-side on database management, data models, and query processing in general, as well as on the specific product in particular. As a result, MapReduce has become very popular and has attained considerable influence in recent times. Google's MapReduce implementation is not freely available. However, an open source implementation of MapReduce, called Hadoop [62], is freely available and is used by companies such as Yahoo!, Facebook, IBM, Adobe, AOL, A9, Last.fm, eBay, LinkedIn, Twitter, and The New York Times.

On one hand, the ease-of-use and massive scalability made Hadoop MapReduce extremely popular. On the other hand, however, researchers soon realized that the performance of Hadoop often does not match the one of a well-configured parallel DBMS. An experimental study published in 2009 compared Hadoop MapReduce with parallel DBMSs [101]. The study showed that parallel DBMSs outperform Hadoop MapReduce by a large factor in a variety of analytical tasks. Supporting this claim, a follow-up work in 2009 combined the techniques from DBMSs and Hadoop MapReduce [5]. The resulting system, called HadoopDB (later spin-off as Hadapt [61]), essentially combines Hadoop's data distribution framework with local DBMSs to form a shared-nothing DBMS. However, HadoopDB has severe drawbacks since it: (i) forces users to install and configure a parallel DBMS, which is a complex process, (ii) changes the interface to SQL, replacing the simple programming model of Hadoop, (iii) uses ACID-compliant

DBMS engines, even though only the indexing and join processing techniques of the local DBMSs are useful for read-only, MapReduce-style analysis, and (iv) requires deep changes in the Hadoop framework.

In 2010, Hadoop++ [49] was proposed to overcome the drawbacks of HadoopDB. This was the first work to show that the performance of Hadoop MapReduce can match that of parallel DBMS, if database techniques such as indexing and join processing are applied in Hadoop MapReduce as well. In fact, Hadoop++ even showed that Hadoop MapReduce and parallel DBMS are two sides of the same medal; the difference being the hard-coded physical query execution plan in Hadoop MapReduce. In order to add flexibility to Hadoop's physical query execution plan, Hadoop++ allows for two additional data processing techniques: (i) indexed data access, and (ii) co-partitioned join processing. This is done by changing the physical representation of data blocks in Hadoop MapReduce: indexed data blocks in order to have index data access and co-partitioned data blocks in order to have co-partitioned joins. Thus, a flexibility in the Hadoop MapReduce storage layer leads to a flexibility in Hadoop MapReduce physical query execution plans. Hadoop++ [49] changed the perception of (inefficient) performance of Hadoop MapReduce. As a result, the focus shifted from parallel DBMSs back to core Hadoop.

Following Hadoop++, several subsequent works in 2011 looked at creating different data layouts in Hadoop [52, 68, 80]. Essentially, they all challenged the *strict-row-layout* assumption in Hadoop MapReduce. This is because with row layout the input data has to be read entirely, even if only some portion of it is required. RCFile [68] solves this problem by storing data in PAX layout. RCFile divides each HDFS data block into *row-groups* (equivalent to data pages in PAX) and stores data within each row-group in column-oriented fashion. Another approach stores each HDFS data block in column-oriented fashion [52]. To do so, this approach stores each column of a HDFS data block in a different file. Furthermore, in order to support efficient querying, the files belonging to the same HDFS data block are co-located, i.e. they are stored on the same data node. Trojan Layouts [80] pushes data layouts to even greater flexibility in Hadoop MapReduce. Trojan Layouts stores each HDFS data block replica in a different data layout. Trojan Layouts allow for row, column, and even column-grouped layouts. This means that at query time, the incoming query can choose the best available layout. As a result, the physical query execution plan is now even more flexible in Hadoop MapReduce.

There is still a lot of ongoing work to push the flexibility of data layouts in Hadoop MapReduce to the extreme. One such recent work, HAIL [50], a follow-up of Hadoop++, pushes indexing to the extreme. HAIL is also inspired from Trojan Layouts in that it does not create just a single index. Instead, HAIL utilizes the idle CPU ticks in the Hadoop MapReduce pipeline to aggressively create as many indexes, each for a different data block replica, as possible when uploading the data itself. This has several important consequences. First, several indexes are available at query time for incoming MapReduce jobs. As a result, there is a higher likelihood for doing an index scan, which is much faster than full scan. Second, there is no prepare time to create the indexes, as in Hadoop++. Instead, indexes are readily available as soon as the data is uploaded. Finally, since HAIL integrates index creation tightly with the Hadoop upload pipeline, the index creation overheads are negligible in the overall data upload costs. As a result, the indexes are created almost for free, i.e. no additional creation costs. This is in sharp contrast to Hadoop++, which has very dominant index creation costs. Such aggressive

techniques are indeed believed to make Hadoop physical query execution plans even more flexible in the future.

1.4 Thesis Statement

Statement. This thesis shows that it is possible to build data managing systems which are not tied to a fixed data store and can create arbitrary physical representations of data. Our experimental results demonstrate that such a flexible storage layer allows us to achieve much better performance over a set of analytical workloads.

Summary. To support the above thesis statement, we proceed as follows.

We first present OctopusDB, a data managing system with a completely flexible data storage layer. OctopusDB is a vision for a one-size-fits-all data managing system — a unified system which can support several query workloads. For this, OctopusDB introduces the concept of Storage Views — an abstraction for any physical representation of data. OctopusDB collects all data in a central log and can create arbitrary storage views on that log. As a result, OctopusDB can mimic a variety of data managing systems including row stores, column stores, streaming systems, scan-oriented systems, etc.

We then present Trojan Techniques, a novel mechanism to introduce storage layer flexibility into an existing data managing system. Trojan Techniques are the first steps towards the OctopusDB vision. Trojan Techniques allow for injecting additional storage views such as layouts and indexes into an existing data managing system. To do so, Trojan Techniques exploit user defined functions (UDFs), provided by almost all data managing systems, to affect the data storage and access from inside. In this thesis, we focus on solving the data analytics problems using MapReduce, which is the de facto standard for large scale data processing. We demonstrate Trojan Techniques on Hadoop MapReduce, the most popular open source implementation of MapReduce paradigm. We take the default row-oriented storage of Hadoop MapReduce and inject several additional storage views, including indexes, co-partitions, vertical partitions, and PAX to support a variety of analytical workloads. All this without changing the Hadoop MapReduce pipeline and only minor changes to user queries (MapReduce jobs). In addition to Hadoop MapReduce, we also demonstrate Trojan Techniques in one top-notch commercial database system.

Overall, Trojan Techniques look to achieve significant advances in data management technology. These include: (i) *Storage layer flexibility.* With Trojan Techniques, a data managing system is not tied to a fixed data store anymore. Instead, users can inject additional storage views and support several query workloads. This will lead to saving costs, including licensing and administration costs, of maintaining a zoo of data managing systems. (ii) *Plug and play storage views.* Trojan Techniques inject additional storage views seamlessly and on demand into a data managing system. This makes Trojan Techniques very easy to use and manage. (iii) *Boosting performance of existing data managing systems.* Trojan Techniques can be applied to existing data managing systems, i.e. there is no need for a complete system rewrite. This saves a lot of money and effort, and creates a win-win situation: flexible storage layer and yet the data managing system remains unchanged. (iv) *True physical data independence.* Unfortunately, physical data independence still remains elusive in modern data managing systems. Trojan Techniques look to change this by decoupling the storage layer from

Trojan Technique	**Data Managing System**	**Description**
Trojan Index	Hadoop MapReduce	indexed data layouts per-HDFS block
Trojan Join	Hadoop MapReduce	co-partitioned data layouts per-HDFS block
Trojan Layouts	Hadoop MapReduce	row, PAX, column-grouped layouts per-HDFS block replica
Trojan Columns	Row-oriented RDBMS	column layouts per horizontal data segments

Table 1.1: Overview of Trojan Techniques

the query execution layer. As a consequence, the storage views can be research and developed independently and then injected using Trojan Techniques.

1.5 Contributions

In this thesis, we present our OctopusDB vision for flexible data storage (Chapter 2) and four Trojan Techniques implementing our vision: Trojan Indexes and Joins (Chapter 3), Trojan Layouts (Chapter 4), and Trojan Columns (Chapter 5). Table 1.1 summarizes these four Trojan Techniques. The contributions of this thesis are as follows.

1.5.1 OctopusDB Vision

First of all, we present the broad vision of our one-size-fits-all database system, coined OctopusDB. Traditional databases have a fixed store (row, column) to store the data. Additionally, they maintain a log recording database operations to support recovery. OctopusDB turns this model upside down: it maintains only the logical log, which is lightweight having no maintenance requirement in the first place and may optionally build additional data structures, called *storage views* on top. This radical shift in design allows OctopusDB to have a highly flexible data storage layer and to mimic a variety of systems.

There could be several ways to realize our OctopusDB vision of flexible data storage layer. In this thesis, we focus on Trojan Techniques to approach this vision. Another way would be to do a complete system rewrite. This is currently underway as a separate work in our group.

We discuss OctopusDB vision and architecture in Chapter 2.

1.5.2 Trojan Indexes and Joins

To put things in perspective, in the following we go back to year 2010, at the very beginning of this thesis.

Hadoop MapReduce has an inflexible data processing pipeline to analyze large datasets. Therefore, Hadoop MapReduce is an ideal candidate to start off with Trojan Techniques. The biggest problem of Hadoop MapReduce is the scan-only MapReduce job execution, even for highly selective queries such as those in [101]. This is inspite of decades of database research having proved indexes to be very useful for selective queries. However,

given the massively parallel data storage and query execution in Hadoop, creating global indexes in Hadoop MapReduce is not trivial. This is because we have to take care of data (and index) distribution, replication, fault tolerance, load balancing, and index fetching at query time. To solve this, we introduce Trojan Indexes in Hadoop MapReduce. Trojan Indexes are per-HDFS-block indexes which are embedded within the HDFS data block itself. As a result, Trojan Indexes are seamlessly integrated within the Hadoop MapReduce framework. From the users perspective, Trojan Indexes are very easy to create as well as to query.

Another major problem in Hadoop MapReduce is data shuffling. Hadoop shuffles the data from all nodes in order to collect the same keys on the same nodes. This process is especially problematic when joining two tables. Hadoop's shuffle phase performs the cross product of two tables across all nodes in order to join them. Obviously, this is very expensive for large datasets. We learn from distributed databases that co-grouping is very useful when performing joins in a distributed system. However, massive parallelization and the hard-coded data processing pipeline in Hadoop makes co-grouping non-trivial for the user. To solve this, we introduce Trojan Joins in Hadoop MapReduce. Trojan Joins co-group two tables per-HDFS-block. Using Trojan Joins, each HDFS block now contains data from two tables in the same join-key range. As a result, the user can now perform a join without transferring any data across the network. Trojan Indexes and Trojan Joins can be freely combined or used in any MapReduce subquery.

We present Hadoop++ as a system implementing Trojan Indexes and Trojan Joins in Chapter 3.

1.5.3 Trojan Layouts

Even after Hadoop++, data layouts still remained inflexible in Hadoop MapReduce. Below let us see the state of Hadoop MapReduce immediately after Hadoop++.

Hadoop MapReduce stores all data strictly in row layout. This means that all data has to be accessed even if only some of the attributes in each tuple are needed. This is a significant overhead given that MapReduce jobs are I/O intensive. To address this issue, we introduce Trojan Layouts in Hadoop. Trojan Layouts allow for row, column, or column grouped layouts in Hadoop. Furthermore, Trojan Layouts are per-HDFS-block, thereby keeping all of Hadoop's data distribution, replication, failover, and load balancing intact. However, the most striking feature of Trojan Layouts is that it exploits default data replication in Hadoop MapReduce and keeps each replica in a different data layout. This means that with default replication factor of 3, we have 3 different data layouts for free. This has significant impact on query performance, since now each query can choose the best of the three available data layouts. Of course, we have to now introduce novel algorithms to find the data layouts for each replica in the first place. Also, we have to tweak the scheduler to take into account both data locality as well as the data layout, when scheduling map tasks.

We detail Trojan Layouts in Hadoop MapReduce framework in Chapter 4.

1.5.4 Trojan Columns

Finally, we look at the use of Trojan Techniques in relational databases. One major problem in relational databases is that they are strictly row-oriented and thus perform poorly on OLAP-style workloads. The first thought is to apply full vertical partitioning [83] in order to mimic columnar behavior in row-oriented databases. Other techniques, such as RowCol [18], have also been proposed to impose columnar behavior on them. However, all these approaches work at the schema level, incur expensive table joins, and hence very inefficient [3]. To solve this, we introduce Trojan Columns in relational databases. Trojan Columns introduce columnar functionality in row-oriented relational databases from *inside*. As a result, at query time, only the required data is read at the storage layer itself. Additionally, Trojan Columns applies several column store features such as compression and late projection. Trojan Columns are plug-and-play and work even with closed source commercial databases.

We show Trojan Columns in action in a commercial database system in Chapter 5.

1.5.5 Publications, Patents, Grants, Awards

Some of the material in this thesis has been previously published in different versions in international conferences and international patents.

- **Chapter 2 — OctopusDB Vision**

  Publications:

  [48] Jens Dittrich, Alekh Jindal.
  Towards a One Size Fits All Database Architecture.
  CIDR 2011, Outrageous Ideas and Vision Track, Asilomar, USA.

  Best Outrageous Ideas and Vision Paper Award, CIDR 2011.

  [78] Alekh Jindal.
  The Mimicking Octopus: Towards a one-size-fits-all Database Architecture.
  VLDB 2010 PhD Workshop, Singapore.

  Patent:

  A method for storing and accessing data in a database system (LU 91726 and WO2012032184).

- **Chapter 3 — Trojan Indexes and Trojan Joins**

  Publication:

  [49] Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad.
  Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing).
  VLDB 2010, Singapore.

- **Chapter 4 — Trojan Data Layouts**

  Publication:

  [80] Alekh Jindal, Jorge-Arnulfo Quiane-Ruiz, Jens Dittrich.
  Trojan Data Layouts: Right Shoes for a Running Elephant.
  ACM SOCC 2011, Portugal.

  Patent:

  Replicated data storage system and methods (filed Internationally by Saarland University).

- **Chapter 5 — Trojan Columns**

  Grant:

  German Ministry of Education and Science (BMBF) Validation of Innovation Potential (VIP) Grant for OctopusDB covering 1.1 million Euros.

Chapter 2

Towards A One Size Fits All Database Architecture

Modern enterprises need to pick the *right* DBMSs e.g. OLTP, OLAP, streaming systems, scan-oriented systems among others, each tailored to a specific use-case application, for their data managing problems. This makes using specialized solutions for each application costly due to licensing fees, integration overhead and DBA costs. Additionally, it is tedious to integrate these specialized solutions together. Alternatively, enterprises use a single specialized DBMS for all applications and thereby compromise heavily on performance. Furthermore, a particular DBMS (e.g. row store) cannot adapt and change into a different DBMS (e.g. streaming system), as the workload changes, even though much of the code and technology is replicated anyways.

In this chapter, we discuss building a new type of database system which fits several use-cases while reducing costs, boosting performance, and improving the ease-of-use at the same time (Section 2.1). We present the research challenges in building such a system. We believe that by dropping the assumption of a fixed store, as in traditional systems like row store and column store, and instead having a flexible storage scheme we can realize much better performance without compromising the cost. We outline OctopusDB as our plan for such a system and discuss how it can mimic several existing as well as newer systems (Section 2.2). To do so, we present the concept of *storage view* as an abstraction of all storage layouts in OctopusDB (Section 2.3). We discuss how the heterogenous optimization problems in OctopusDB can be reduced to a single problem: *storage view selection*; and describe how a Holistic Storage View Optimizer can deal with it (Section 2.4). For completeness, we also describe how OctopusDB supports several core database features, including purging, checkpointing, recovery, transactions, and isolation (Sections 2.5, 2.6, and 2.7). Finally, we present simulation results to justify our core idea and experimental evidence on our initial prototype to demonstrate our approach (Section 2.8).

2.1 Introduction

2.1.1 Background

Database management systems started off as monolithic systems. However, database engineers soon started tuning their performance for specific applications. Consequently, currently we are witnessing a split of data management systems into several specialized solutions [115, 117] e.g. OLTP for transactional queries, OLAP for read-only complex reporting queries, DSMS (data stream management systems) for continuous window queries, and search engines for read-only keyword queries. It started in the mid-nineties [53] when the database engineers understood that the DBMSs of that time were ill-equipped to cope with the size of the datasets and complexity of OLAP-queries. Therefore a separate type of system was forked from the one size fits all DBMS code line [54]. That system is based on a column store and became one of the most popular and successful approaches for OLAP; products include SAP BI Accelerator, InfiniDB and Paraccel. At the same time other types of systems were forked including DSMS (data stream management systems) [122]; products include StreamBase. In addition, search engines developed into a separate community sometimes re-inventing DBMS technology. Yet these people are unwilling to use DBMS systems as a backend [118]. As a result, we have one specialized system per use-case application.

2.1.2 Motivation

A typical enterprise, today, employs a variety of data managing applications, and hence a variety of DBMSs. For instance, a banking enterprise uses an OLTP system for real time transactions, OLAP system for business intelligence and analytics and Streaming system for stock trading. Additionally, in many cases applications need to adapt to new requirements or evolve over time and usage; possibly requiring a switch to a different DBMS for optimal performance. For instance, in the banking enterprise the stock data prior to a time window may be pushed into a Archival system. These heterogenous systems, however, need to be integrated by copying data from one database to another using complex ETL-style data pipelines. Moreover, these *specialized* systems have their specialized vendors and DBAs, thereby incurring further licensing and maintenance costs. Obviously, all of this leads to extra costs in terms of development costs, maintenance costs, and DBA costs. So rather than making the world of data management easier, we have created a *zoo* of systems that sometimes has the opposite effect: it makes life of a company harder and more costly. We agree that for companies who invest a lot into connecting the different species in their zoo, it will eventually lead to a well-integrated and efficient overall system. Still, we believe that the zoo-keeping costs are non-trivial, especially for small to medium-sized business. We also believe that adapting such a zoo to new requirements, changing workload, or new types of applications may be prohibitive.

Moreover, one technology (e.g. row stores, column stores etc.) per system does not always deliver the best performance. For example, consider a university database with Student and Lecture tables. Now for a query workload requesting students based on lecture types, credit points or instructor, we need to access only few attributes from the Lectures table but most (or all) attributes from Student table. For such a use-case scenario it makes sense to store the Student table in a row store and the Lecture table in a column store, thereby deriving the maximum performance. However, such a

configuration is not possible to achieve in traditional *per-application* database systems, except in Fractured Mirrors [106]. Fractured Mirrors, however, has an exorbitant update cost. Moreover, as students graduate and hence do not attend lectures anymore, their details are not fetched by our query workload. Therefore, to improve performance it might make sense to partition the cold data (graduated students) in the Student table into a column store, utilizing compression, and the hot data (current students) into a row store. Again, such a configuration is impossible to achieve in a *single* traditional database system.

Furthermore, we analyzed the TPC-H benchmark to illustrate how the use-case scenario and the workload could determine the right DBMS technology. For each query in the benchmark we marked the attributes referenced by it in each of the 8 relations. Our analysis revealed that each relation has a different attribute access pattern such that some attributes are referenced more than others while some attributes are not referenced at all e.g. `retailprice`, `comment` in `PART` relation. Furthermore, several groups of attributes are co-referenced in the same queries. Thus, a single type of store (row, column) may not suit all relations. For instance, relations `LINEITEM`, `NATION` and `REGION` have several attributes referenced in the same queries and therefore a row store could be more suited. On the other hand, relations `PART`, `SUPPLIER`, `CUSTOMER` and `ORDER` have only few attributes referenced and hence column store would be the better choice. This analysis hints for a more holistic approach to database storage design.

2.1.3 Problem Statement

In this section we discuss the problem we focus on and describe our overall goal. As highlighted in the previous section: (1) A single specialized DBMS may not deliver the best performance for all applications, (2) Modern enterprises, anyway, end up having a zoo of database systems, (3) It is tedious to stitch together (complex ETL-style data pipelines) and costly to maintain (development, licensing, DBA costs) different database systems, and (4) Optimal performance over changing workloads, in a single or a zoo of DBMSs, remains an issue. Therefore, we need a *one-size-fits-all* database system which automatically adapts to different use-case scenarios and betters performance at the same time. We state our research goal as follows:

Research Goal. *A one-size-fits-all database system which caters well to all existing and newer data management use-cases and adapts automatically to initial configuration as well as to changing workload; all with improved performance, lowered cost and better maintainability at the same time.*

2.1.4 Research Challenges

Below we discuss the major research challenges associated with our research problem.

2.1.4.1 Different Storage Layouts under a single umbrella

Row stores are typically used in transactional processing systems (OLTP). Column stores on the other hand make heavy use of compression and are used in read-oriented workloads. Many people argue that these are completely different systems [3]. Furthermore, streaming systems may not have any store at all. The *one-size-fits-all* system needs to cater the different storage layouts for different use-case scenarios. Additionally, the system need to adapt and interchange the storage layouts to changing workoads. The challenge, therefore, is to have a flexible storage scheme by bringing the different layouts into a single system. This brings to the fore three additional issues that we have to care about.

First, layout selection and maintenance is a major concern with having different layouts in a single system. This is because with changing workloads, the system needs to automatically decide the most appropriate layouts to create, maintain them with future updates and finally decide upon when, if at all, to discard them altogether. Second, query processing across different storage layouts is another important issue. Ideally, we would want the query processor to abstract much of the functionality across different layouts. At the same time, we would not want to completely miss out the storage layout specific optimizations e.g. compression in column stores.

Finally, apart from row and column layouts of the full table, the system can also create other sub-structures to boost performance. For instance, it can crack the table, not only horizontally as in [71] but also vertically, depending upon which part of it is accessed by the incoming queries; partition the tables horizontally or vertically in case of a definitive workload information; or create materialized views on any subset of the data. The challenge is to automatically create and manage this inventory of storage layouts. Furthermore, all layouts discussed so far store data in rectangular fashion. It would be interesting to even consider non-rectangular storage layouts for the given data. The underlying idea in a *one-size-fits-all* system remains the same: relax the fixed layout assumption. The unlimited possibilities thereafter offer unique research challenges.

2.1.4.2 Automatic Adaptive Bifurcation instead of administered Eventual Integration

Currently, companies having several types of database systems spend a lot of time and effort to eventually integrate them together. As pointed before, these costs are non-trivial for small to medium sized companies. Furthermore, the integrated system is less adaptive to a change in workload followed by a consequent addition or removal of a database system. Therefore, starting with a bag of database systems in the first place might end up with a loosely integrated system which is difficult to manage and expensive to maintain. Instead, since much of the code and technology of different DBMSs is anyways replicated, we can start with a single system for all applications.

The challenge, therefore, is to adaptively bifurcate the system into specialized technology (row, column etc.) depending upon the workload. The system needs to continuously monitor the workload and reassess its configuration. This makes sense because (1) we have a tightly integrated system (2) we abstract much of the code and technology and fork out only the necessary one, and (3) the system is fairly simple initially and is later

adapted to be only as much complex as needed. We discuss the last point in more detail below.

2.1.4.3 Simplicity Vs Optimization

Simplicity is an important consideration while designing database systems. In several cases too much of optimization in a database system is an overkill. For example, the materialized views created for each operator output in MonetDB [17] can be detrimental for changing workloads. Simplicity pays off in terms of performance and house-keeping. The challenge, therefore, is to strike the right balance between simplicity and optimization in the single system. Of course, simplicity might be traded for performance by incorporating more complex optimizations as the workload gets more sophisticated. But the key is to avoid any overkill.

Another aspect of simplicity is to support several use-cases in the minimal configuration, i.e. the system should first try to *mimic* as many specialized systems as possible before upgrading the configuration. This is necessary because changing configuration could be an overkill as well as expensive. Other people have also tried to develop systems which mimic more than one system. For example, [18] tries to mimic a column store in a row store. The challenge, however, is to determine the limiting point for the mimic.

Finally, a database system which is always initialized in the most rudimentary configuration might be quite slow till it adapts to the initial workload. Depending upon the adaptability speed, this slow start can be quite expensive. Additionally, we may incur the startup cost always, even when the system is reset or the data is ported to another instance. Instead, the system should be able to set its initial configuration, depending upon the initial workload, in order to derive the maximum performance straightaway. The challenge again lies in deciding how much of the simplicity should be lost at the very outset.

2.1.5 Our Approach

We take a radically new approach: we propose a single type of database system coined *OctopusDB*[1] that is able to mimic the behavior of the different species in the zoo. In addition, OctopusDB is also able to mimic fictional species, e.g. mermaids, centaurs; as well as new species, e.g. "old elephants" having the fast legs of a young "fox" [88].

Core Idea. The core idea of our system is to drop the assumption that a database system is developed around a central store (be it a row, column, or a any hybrid store such as PAX [11] or fractured mirrors [106]). OctopusDB does not have a fixed store. In OctopusDB all data is collected in a central log, i.e., all insert and update-operations create logical log-entries in that log. Based on that log we may then define several types of optional *Storage Views*. A *Storage View (SV)* represents all or part of the log in a different (or the same) physical layout. For instance, we may define a *Row SV* representing all or part of the data from the log in a row store. That Row SV, however,

[1] An octopus may adapt to its surroundings using a camouflage unmatched by any other species on earth: it may change both the color *and* the texture of its skin. Additionally, some octopus species may even mimic movements and shape thus *impersonating* other species, e.g. http://marinebio.org/species.asp?id=260

is just one possible storage view. It may dynamically be replaced by a *Col SV* if it better suits the workload. We might even keep one part of a table, e.g. old entries that are not updated, in a Col SV, and another part in a Row SV, e.g. entries that are still being updated. This emulates a hybrid of OLTP and OLAP. In addition, we may also replace some of the "tables" by streaming windows, e.g. we emulate a data stream management systems. Furthermore, we may model combinations of continuous and store (=archival) queries which has been researched heavily in the past years [47]. In OctopusDB all of this is done transparently and solely based on the workload — and not based on some static decision for a concrete database product and hence a concrete storage layout.

In summary, the storage view concept allows us to model several important data managing concepts using a single abstraction only: both types of queries (point-in-time and continuous queries); different database stores (row-, col-, hybrid, etc.); and also the traditional query views (dynamic or materialized).

This has another interesting consequence: the query optimization, view maintenance, index selection, as well as the store selection problems suddenly become a single problem: *storage view selection*. OctopusDB treats all four problems inside a single holistic *storage view optimizer* which we are introducing with this chapter.

Contributions. We make the following contributions:

(1.) We propose a new unified database architecture called OctopusDB.
(2.) We introduce the concept of *Storage Views (SV)*. We identify different types of SVs and describe how to create and use them.
(3.) We present a holistic storage view optimizer for OctopusDB. We present techniques for operator Log-Pushdown, adaptive partial SVs, and stream transformations.
(4.) We present cost models for querying, updating, and transforming SVs.
(5.) We show results of a simulation as well as an experiment with a prototype of OctopusDB demonstrating the viability and efficiency of our approach.

2.2 OctopusDB Overview

2.2.1 Data Model

The data items managed by OctopusDB are tuples $t_i = (a_1, .., a_{n(i)})$, $0 \leq i \leq N$ where attributes $a_1, \ldots, a_{n(i)}$ may be of any type. The number of attributes for tuple i is given by $n(i)$. Each tuple is associated to a *bag* and a *key*. The *bag* is used to define subsets of the tuples, e.g. tables, partitions, collections. *key* identifies a tuple inside a bag. For simplicity, we assume a relational model throughout this chapter. Therefore all tuples having the same *bag* share the same set of attributes (=schema). However, in general this need not be the case: a tuple may specify only some attributes not specified by other tuples with the same *bag*-identifier.

2.2.2 The Primary Log Store

OctopusDB does *not* keep a row-, column- or any other store by default. All calls to the system interface are simply recorded in a sequential log, called *primary log*,

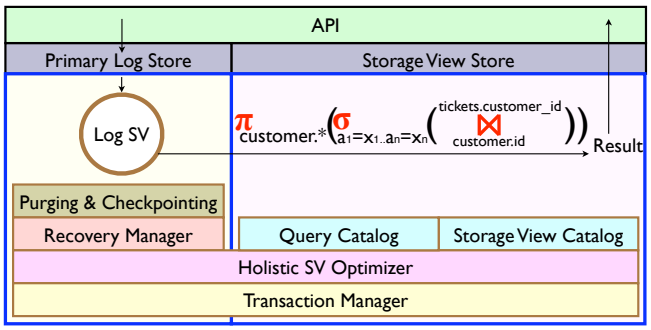


FIGURE 2.1: Initial, Non-Optimized OctopusDB for the Running Example (compare Figures 2.2 and 2.3)

creating appropriate logical log records. The primary log is itself an SV. OctopusDB stores its log persistently on durable storage (hard disk or SSD) following the write-ahead logging-protocol (WAL). For efficiency reasons we may keep a copy of the log in main memory, however this is no requirement. Each call to the system interface of OctopusDB internally creates a log record with an associated log sequence number *lsn*. As in traditional DBMSs no two log records may have the same lsn, therefore entries to the log are serialized. For the moment, all log records are *logical* and represent a *new state* defined by an operation[2]. Therefore, in contrast to ARIES, our log records do **not** represent changes that have been or should be applied to the database store. Our log simply contains the event history of operations without specifying how these events map to a particular store[3]. Thus, the format of our log record is (lsn, <method>, <parameters>) where <method> denotes the method of the system interface called and <parameters> denotes the parameters passed.

2.2.3 System Components and Interface

Figure 2.1 shows the main components of OctopusDB. The system contains components similar to the ones known from traditional DBMSs (transaction manager, query optimizer, etc.). The most striking difference is the *primary log store* as well as the *storage view store*.

OctopusDB has a simple yet powerful interface containing the following methods:

registerSV(String svID, Type svType<, additionalPar>): creates and registers an SV of type svType having a unique identifier svID. Additional parameters may be passed to the SV.
registerQuery(String queryID, Query Q<, callback>): registers a query having a unique identifier queryID. An additional callback function may be passed.
snapshot(String outputSVID, String queryID): computes the result of the query and materializes it into the output SV.
maintain(String ouputSVID, String queryID): Same as snapshot, however, future updates will be reflected in outputSVID.
drop(String ID): Drops a query or SV from the system.

[2]In addition, transitional log records may be used, e.g. $a = a + 42$.

[3]The major performance advantage of ARIES is that it is using *physical* logging for REDO, i.e. intertwining a particular store with the log is a *feature* of ARIES. However, this feature may also be implemented in OctopusDB without giving up the logical primary log. See Section 2.6 for details.

query(Query Q)→ Iterator it: Queries and/or modifies data in OctopusDB as specified in Q.
iterate(String ID) → Iterator it: Returns an iterator over the contents of the given query or SV.

Query definitions may be either a relational algebra expression as suggested in [31], SQL, or Pig Latin [99]. We will assume a relational algebra expression throughout this chapter.

2.2.4 Algorithms

Let us now see the algorithms to implement the OctopusDB interface in one specific instance. Algorithm 2.1 shows the **registerSV** algorithm in OctopusDB. It simply creates a SV and puts it in the StorageViewCatalog. Algorithm 2.2 shows **registerQuery** algorithm. Apart from putting the query in the QueryCatalog (Line 1), we check whether a callback is provided (Line 2) i.e. the query is maintained. If yes, we retrieve the input SVs appropriate for this query (Line 4). Finally we add the input SV IDs, the query and the callback to be maintained in the StorageViewCatalog (Line 6).

Algorithm 2.1: registerSV

```
   Input: String svID, Type svType<, additionalPar>
1  StorageView sv = CreateStorageView(svType<, additionalPar>);
2  StorageViewCatalog.putSV(svID,sv);
3  LogRecord newLR = new RegisterSVLogRecord(svID, svType<, additionalPar>);
4  this.getPrimaryLog().append(newLR);
```

Algorithm 2.2: registerQuery

```
   Input: String queryID, Query Q<, callback>
1  QueryCatalog.putSV(queryID,Q);
2  if callback != NULL then
3  |  // maintained query:
4  |  StorageView[] inputSVs = QueryOptimizer.getInputSVs(Q);
5  |  String[] inputSVIDs = StorageViewCatalog.getSVIDs(inputSVs);
6  |  StorageViewCatalog.maintain(inputSVIDs, queryID, callback);
7  end
8  LogRecord newLR = new RegisterQueryLogRecord(queryID, Q<, callback>);
9  this.getPrimaryLog().append(newLR);
```

Algorithm 2.3 shows the **snapshot** algorithm of OctopusDB. For efficiency reasons SV implementations may also provide ordered iterators like **iterateBackwards(String ID)** which is exploited in this algorithm. First, we retrieve the Query and the StorageView from the Query and StorageView catalog (Lines 1–2). Next, we fetch the input SVs appropriate for the given query (Line 3). The system checks whether the query is a selection or a projection query (Line 4). If yes, it sources the selected or projected tuples, respectively, to the output SV (Lines 5-12). Otherwise, the system invokes the query optimizer to retrieve the cheapest plan and executes it (Lines 15–16). In any case the system creates and inserts a snapshot log record into the primary log (Lines 18–19). If OctopusDB receives any further updates, the output SV will not be maintained, i.e., this method creates a snapshot of the data.

Algorithm 2.4 shows the **maintain** algorithm. The only difference from snapshot algorithm is that we add the input SV IDs, the query and the output SV ID to be maintained in the StorageViewCatalog (Line 3).

Algorithm 2.3: snapshot

```
   Input: String outputSVID, String queryID
 1 Query Q = QueryCatalog.getQuery(queryID);
 2 StorageView outputSV = StorageViewCatalog.getSV(outputSVID);
 3 StorageView[] inputSVs = QueryOptimizer.getInputSVs(Q);
 4 if Q == σ_P or Q == π_A then
 5 |   // single input:
 6 |   Iterator it = inputSVs[0].iterateBackwards(queryID);
 7 |   it.open();
 8 |   while it.hasNext() do
 9 |   |   LogRecord next = it.next();
10 |   |   Source(outputSV, next);
11 |   end
12 |   it.close();
13 else
14 |   // composing query:
15 |   PhysicalPlan plan = QueryOptimizer.getPlan(inputSVs,Q);
16 |   plan.execute(outputSV);
17 end
18 LogRecord newLR = new SnapshotLogRecord(outputSVID, queryID);
19 this.getPrimaryLog().append(newLR);
```

Algorithm 2.4: maintain

```
   Input: String outputSVID, String queryID
 1 // same as snapshot:
 2 String[] inputSVIDs = StorageViewCatalog.getSVIDs(inputSVs);
 3 StorageViewCatalog.maintain(inputSVIDs, queryID, outputSVID);
 4 LogRecord newLR = new MaintainLogRecord(outputSVID, queryID);
 5 this.getPrimaryLog().append(newLR);
```

Algorithm 2.5: drop

```
   Input: String ID
 1 if StorageViewCatalog.contains(ID) then
 2 |   StorageViewCatalog.remove(ID);
 3 else if QueryCatalog.contains(ID) then
 4 |   QueryCatalog.remove(ID);
 5 else
 6 |   // invalid ID:
 7 end
 8 LogRecord newLR = new DropLogRecord(ID);
 9 this.getPrimaryLog().append(newLR);
```

Algorithm 2.5 shows the **drop** algorithm. We check whether the input ID is present in the Query or the SV catalog and remove the respective query or SV. Algorithm 2.6 shows the **query** algorithm of OctopusDB. We check whether the query is a select query (Line 1). If yes, we get the input SVs for the query (Line 3) and return an iterator over the query result (Lines 4–11). Otherwise, we first put the incoming query into the log (Lines 14–16) and then source the log record to all maintained SVs recursively (Lines 17–20). Algorithm 2.7 shows the **iterate** algorithm of OctopusDB. We check whether the ID is in QueryCatalog (Line 1). If yes, we get the input SVs for the query (Line 4) and return an iterator over the query result (Lines 5–12). Otherwise, if the StorageViewCatalog contains the ID then we return the SV iterator (Lines 14–16).

2.3 Storage Views

Storage Views (SVs) allow us to define arbitrary physical representations on the log. The main idea of a SV is to store the entire or a subset of the log or any other SV using

Algorithm 2.6: query

```
Input  : Query Q
Output: Iterator if Q is SELECT query
1  if Q == SELECT then
2  |  // select query:
3  |  StorageView[] inputSVs = QueryOptimizer.getInputSVs(Q);
4  |  if Q == σ_P or Q == π_A then
5  |  |  // single input:
6  |  |  return inputSVs[0].iterate(queryID);
7  |  else
8  |  |  // composing query:
9  |  |  PhysicalPlan plan = QueryOptimizer.getPlan(inputSVs,Q);
10 |  |  return plan.execute();
11 |  end
12 else
13 |  // modify query:
14 |  LogRecord newLR = new QueryLogRecord(Q);
15 |  LogSV log = this.getPrimaryLog();
16 |  log.append(newLR);
17 |  StorageView[] maintainedSVs = StorageViewCatalog.getMaintainedSVs(log);
18 |  foreach sv in maintainedSVs do
19 |  |  SourceRecursive(sv, newLR);
20 |  end
21 end
```

Algorithm 2.7: iterate

```
Input  : String ID
Output: Iterator
1  if QueryCatalog.contains(ID) then
2  |  // iterate over query result:
3  |  Query Q = QueryCatalog.getQuery(queryID);
4  |  StorageView[] inputSVs = QueryOptimizer.getInputSVs(Q);
5  |  if Q == σ_P or Q == π_A then
6  |  |  // single input:
7  |  |  return inputSVs[0].iterate(queryID);
8  |  else
9  |  |  // composing query:
10 |  |  PhysicalPlan plan = QueryOptimizer.getPlan(inputSVs,Q);
11 |  |  return plan.execute();
12 |  end
13 else if StorageViewCatalog.contains(ID) then
14 |  // iterate over storage view:
15 |  StorageView sv = StorageViewCatalog.getSV(ID);
16 |  return sv.iterate(null);
17 else
18 |  // invalid ID:
19 end
20 LogRecord newLR = new DropLogRecord(ID);
21 this.getPrimaryLog().append(newLR);
```

a different physical layout. SVs **always materialize** their data. In general we create a network of SV dependencies with the goal to balance update and query processing costs. The dependency graph between different SVs is called the *SV lattice*. It is similar to the one used in materialized view maintenance (e.g. in data warehouses). However, the SV lattice is more general as it is not restricted to queries only but also has to consider the underlying storage layout. The interface to a SV contains the following private methods:

iterate(String queryID)→ Iterator it: Returns the result of the given query as an unordered iterator it. The query must be restricted to data covered by this SV.
iterationCost(String queryID)→ Cost c: returns the estimated cost c of the given query. In other words, it estimates the cost of **iterate(String queryID)**.

transformationCost(Type svType)→ Cost c: returns the estimated cost c for transforming this SV into svType.

The latter two methods estimate local costs within a SV. These local costs are then used by the optimizer to make a global cost estimate. Note that even though this interface resembles the one of a key-value store, OctopusDB is not restricted to key-access. We showed examples for these in Section 2.4.

Running Example. Consider a flight-booking system with a table `TICKETS` containing data on flight tickets; `CUSTOMERS` containing data on customer. Queries select tickets using predicates on different attribute subsets of `TICKETS`. For all selected tickets we retrieve all attribute values of matching customers. We assume that `TICKETS` is frequently updated. Thus, index maintenance on `TICKETS` is too expensive. This is a real-world example as proposed in [128]. This scenario calls for having a column layout on `TICKETS` and a row layout on `CUSTOMERS`. However, this flexible layout is not supported by current DBMSs. The update rate also precludes using fractured mirrors.

Figure 2.1 shows a non-optimized OctopusDB instance for the Running Example. Initially, the SV store does not contain any SVs; it only contains a single registered join query. It would be evaluated by scanning the primary log. We will optimize this Running Example in detail in the following.

2.3.1 Log SV

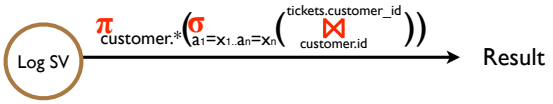
We start with the most simple type of SV: the *Log SV*. The primary log store of OctopusDB simply contains a single Log SV. However, Log SVs may also be used in other places. In the following we will gradually enrich the SV graph of Figure 2.1. In the first step we will partition the data into different SVs based on their bag identifiers.

```
registerSV("ticketsLog", LogSV);
registerSV("customersLog", LogSV);
registerQuery("customersOnly", σ_{bag=customers});
registerQuery("ticketsOnly", σ_{bag=tickets});
maintain("ticketsLog", "ticketsOnly");
maintain("customersLog", "customersOnly");
```

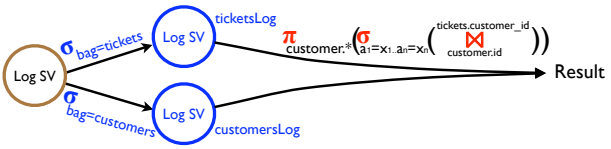
This results in the SV lattice as depicted in Figure 2.2(b). We observe that the join query is not connected to the primary log anymore but to the two new Log SVs. Thus, the decision how to connect new queries and how to snapshot or maintain SVs is fully up to the system. As the primary log does not discard entries having the same `(bag,key)`-pair, both `ticketLog` and `customersLog` contain different versions for the same `(bag,key)`-pair. We group these entries and only keep the most recent one.

```
registerQuery("custRecent", γ_{recent}(Γ_{bag,key}(customersOnly)));
registerQuery("tickRecent", γ_{recent}(Γ_{bag,key}(ticketsOnly)));
maintain("ticketsLog", "tickRecent");
maintain("customersLog", "custRecent");
```

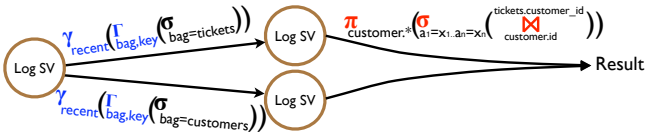
This results in the SV lattice as depicted in Figure 2.2(c). Both `ticketLog` and `customerLog` now only contain the most recent `(bag,key)`-pairs.



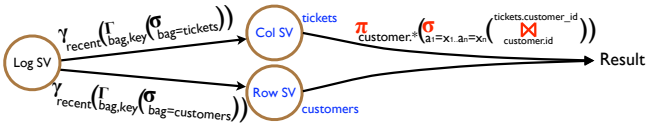
(a) Non-optimized OctopusDB



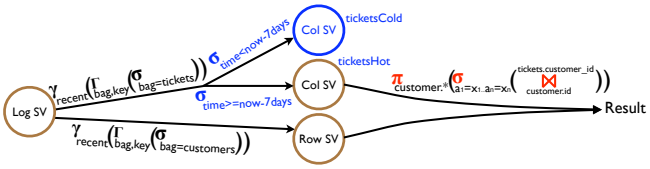
(b) Bag Partitioning: one Log SV per table



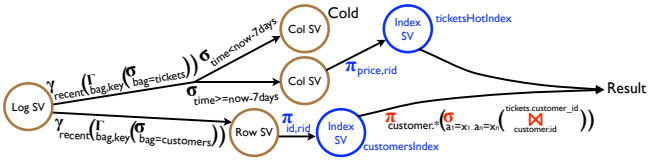
(c) Key Consolidation: remove different versions



(d) SV Transformation: use more efficient Row SV and Col SV



(e) Per-partitions SVs: use cold and hot SVs



(f) Indexing: create two indexes

FIGURE 2.2: OctopusDB static optimizations for the Running Example

2.3.2 Row, Col, and other SVs

Per-table SVs. The main idea of a Row SV, resp. Col SV, is to create a row store, resp. column store, for any given subset of the data as specified by a query. Thus, a different physical layout of the data is used. For instance, the scanning costs for the two logical logs `ticketLog` and `customerLog` may eventually become too high, e.g. surpassing a given SLA. In that case we may replace these Log SVs by a Col SV and a Row

SV.

```
registerSV("tickets", ColSV);
registerSV("customers", RowSV);
maintain("tickets", "ticketsOnly");
maintain("customers", "customersOnly");
drop("ticketsLog");
drop("customersLog");
```

Figure 2.2(d) shows the resulting SV lattice. In summary, `tickets` is kept in the same layout as done by a traditional DBMS using a column store. In contrast, `customers` is kept in the same layout as done by a traditional DBMS using a row store. Thus, OctopusDB mimics different stores for those tables. Obviously OctopusDB could use a different layout for each "table" being registered. In addition, we could also create a Row SV and a Col SV on the same input SV thus easily mimicking fractured mirrors [106]. However, our system can do much more.

Per-partition SVs. Assume that some of the rows in `tickets` are accessed more often than others. For instance, queries might only be interested in tickets not older than 7 days. In this case we could easily split `tickets` into two SVs `ticketsCold` and `ticketsHot` as shown in Figure 2.2(e).

```
registerSV("ticketsCold", ColSV);
registerSV("ticketsHot", ColSV);
registerQuery("tickRecentHot",σ_{time≥now−7days}(tickRecent));
registerQuery("tickRecentCold",σ_{time<now−7days}(tickRecent));
maintain("ticketsCold", "tickRecentCold");
maintain("ticketsHot", "tickRecentHot");
drop("tickets");
```

This basically creates two horizontal partitions using a range partitioning on time. It also faintly resembles buffer management: a DB buffer may keep the hot data on a different physical device (main memory) than the cold data (disk). Again, one of the core ideas of OctopusDB is to make all these decisions optional and automatable.

2.3.3 Index SV

We may also use any type of index in OctopusDB including B^+-trees, hash indexes, bitmaps, cache-optimized-trees, R-trees, inverted indexes, and so forth. This is because indexes are just another type of SV. The Index SV typically uses a subset of the existing attributes to build indexes. In this case an index lookup will point to data in an underlying SV, e.g. a Row SV. The index may also create a *covering index* by copying all attributes to avoid those lookups. Alternatively, a clustered index may be created by rearranging the order of tuples in the underlying SV. The index may also be build on only parts of a table thus mimicking partial indexing [114]. Obviously, adding indexes improves query performance for selective queries. On the downside update costs increase as all existing indexes affected by an update have to be maintained. Thus the decision

whether to create an index (or any other SV) will be affected by the query/update pattern of the particular workload. For instance, in our running example, we can build indexes on customer ids and hot tickets as follows.

```
registerSV("ticketsHotIndex", IndexSV, uncl, key=price);
registerSV("customersIndex", IndexSV, cl, key=id);
registerQuery("tickI1", π_{price,rid}(ticketRecentHot));
registerQuery("custI2", π_{ID,rid}(custRecent));
maintain("ticketsHotIndex", "tickI1");
maintain("customersIndex", "custI2");
```

2.4 Holistic SV Optimizer

2.4.1 Overview

In general, each class of SV may implement its own access algorithms optimized for the particular storage structure. For instance, a Row SV may use row-wise compression and row-oriented iteration, e.g. [70]. In contrast, a Col SV may implement column-oriented compression and vectorized iteration [17]. Additionally, OctopusDB may push down selections, projections, or entire subplans to the respective SVs. The SVs may then locally use their own optimizer to optimize subplans. This has the advantage, that query processing techniques optimized for a specific store may be leveraged. Outside those SVs OctopusDB's *Holistic Storage View Optimizer* then implements any appropriate techniques to:

(1.) speed-up query processing, i.e. pick the most promising physical execution plan to compute a query;
(2.) apply updates to any SV in the SV store, i.e. pick the best update method like a batch-oriented differential update or log-structured merge-trees;
(3.) decide on the SVs to create and keep in the SV store, i.e. whether to materialize a new SV or drop an existing one;
(4.) combine results spanning several SVs, e.g. to join data from a row, a column store, and a streaming window.

As mentioned above all these tasks may be generalized into a single task: *SV selection*. Therefore, OctopusDB allows us to handle these problems using a single holistic SV optimizer. In the following we sketch some of the features of our optimizer.

2.4.2 Cost Model

The holistic SV optimizer needs a cost model to perform the above mentioned optimizations. We present cost models for SV querying (scan and index), SV updating (scan and index), and SV transformation below. We consider Log, Row, Col, and Index SV. Table 4.2 describes the symbols used in our cost models.

Query Cost. Table 2.1 shows the query cost models for Log, Row, Col, and Index SVs. We express each of the cost functions as a summation of random and sequential I/O

costs. We consider the scan operation to be I/O-bound and hence neglect CPU costs. Notice that the scan operations for Row and Col SVs are buffered reads, i.e. OctopusDB reads as many tuples from a SV as can fit in the memory assigned to it. We need buffered reading for Col SV, because we need to join individual attributes to re-construct the tuple; for Row SV we also consider the additional random I/O costs when reading multiple relations competing for the same hard disk, e.g. for join processing.

Symbol	Meaning	Model
$C^{\log}_{\text{scan}}(N)$	Log SV scan cost	$\left\lceil \frac{\sum_{i=1}^{N} colsize(log_i)}{m} \right\rceil \cdot C_{\text{r}} + \left\lceil \frac{\sum_{i=1}^{N} colsize(log_i)}{p} \right\rceil /\text{BW}$
$C^{\text{row}}_{\text{scan}}(N)$	Row SV scan cost	$\left\lceil \frac{N \cdot \sum_{A_i \in A} colsize(A_i)}{m} \right\rceil \cdot C_{\text{r}} + \left\lceil \frac{N \cdot \sum_{A_i \in A} colsize(A_i)}{p} \right\rceil /\text{BW}$
$C^{\text{col}}_{\text{scan}}(N, S)$	Col SV scan cost	$\sum_{A_i \in S} \left(\left\lceil \frac{N \cdot \sum_{A_i \in S} colsize(A_i)}{m} \right\rceil \cdot C_{\text{r}} + \left\lceil \frac{N \cdot colsize(A_i)}{p} \right\rceil /\text{BW} \right)$
$C^{\text{index}}_{\text{lookup}}(N)$	Index lookup cost	$C_{\text{r}} \cdot \lceil log_F(N \cdot (colsize(key) + \text{pointerSize})/\text{p}) \rceil$
$C^{\text{row cl.index}}_{\text{scan}}(N, sel)$	Uncl. Index Row SV scan cost	$C^{\text{index}}_{\text{lookup}}(N) + C^{\text{row}}_{\text{scan}}(\lceil sel \cdot N \rceil)$
$C^{\text{col. cl.index}}_{\text{scan}}(N, S, sel)$	Uncl. Index Col SV scan cost	$C^{\text{index}}_{\text{lookup}}(N) + C^{\text{col}}_{\text{scan}}(\lceil sel \cdot N \rceil, S)$
$C^{\text{row uncl.index}}_{\text{scan}}(N, sel)$	Cl. Index Row SV scan cost	$C^{\text{index}}_{\text{lookup}} + \lceil sel \cdot N \rceil \cdot (C_{\text{r}} + \text{p/BW})$
$C^{\text{col. uncl.index}}_{\text{scan}}(N, S, sel)$	Uncl. Index Col SV scan cost	$C^{\text{index}}_{\text{lookup}} + \lceil sel \cdot N \rceil \cdot \lvert S \rvert \cdot (C_{\text{r}} + \text{p/BW})$

TABLE 2.1: Storage View Query Cost model

Symbol	Meaning	Model
$C^{\log}_{\text{update}}(N_u)$	Log SV update cost	$C^{\log}_{\text{scan}}(N_u)$
$C^{\text{row}}_{\text{update}}(N, N_u)$	Row SV update cost	$\min\left(C_{\text{r}} + \left\lceil \frac{N}{N_c} \right\rceil \cdot C^{\text{row}}_{\text{scan}}(2 \cdot N_c), \left\lceil \frac{N}{N_c} \right\rceil \cdot C^{\text{row}}_{\text{scan}}(N_c) + N_u \cdot (C_{\text{r}} + \text{p/BW})\right)$
$C^{\text{col}}_{\text{update}}(N, N_u, S)$	Col SV update cost	$\min\left(C_{\text{r}} + \left\lceil \frac{N}{N_c} \right\rceil \cdot C^{\text{col}}_{\text{scan}}(2 \cdot N_c), \left\lceil \frac{N}{N_c} \right\rceil \cdot C^{\text{col}}_{\text{scan}}(N_c) + N_u \cdot \lvert S \rvert \cdot (C_{\text{r}} + \text{p/BW})\right)$
$C^{\text{index}}_{\text{split}}(d)$	Index split cost	$\left(\sum_{i=1}^{d} (p_{\text{split}})^i\right) \cdot C_{\text{r}}$
$C^{\text{row cl.index}}_{\text{update}}(N, N_u, d)$	Cl. Index Row SV update cost	$C^{\text{index}}_{\text{lookup}}(N) + 2 \cdot C^{\text{row}}_{\text{scan}}(N_u) + C^{\text{index}}_{\text{split}}(d)$
$C^{\text{col. cl.index}}_{\text{update}}(N, N_u, S, d)$	Cl. Index Col SV update cost	$C^{\text{index}}_{\text{lookup}}(N) + 2 \cdot C^{\text{col}}_{\text{scan}}(N_u, S) + C^{\text{index}}_{\text{split}}(d)$
$C^{\text{row uncl.index}}_{\text{update}}(N, N_u, d)$	Uncl. Index Row SV update cost	$C^{\text{index}}_{\text{lookup}} + N_u \cdot (C_{\text{r}} + p/BW) + C^{\text{index}}_{\text{split}}(d)$
$C^{\text{col. uncl.index}}_{\text{update}}(N, N_u, S, d)$	Uncl. Index Col SV update cost	$C^{\text{index}}_{\text{lookup}} + N_u \cdot \lvert S \rvert \cdot (C_{\text{r}} + p/BW) + C^{\text{index}}_{\text{split}}(d)$

TABLE 2.2: Storage View Update Cost model

Symbol	Meaning	Unit
N	number of rows in table	
N_c	number of rows in a table chunk	
BW	sequential bandwidth of hard disk	Pages/sec
C_{r}	costs for a random access	sec
p	size of a page	Byte
$pointerSize$	size of a pointer in an index	Byte
F	fan-out of index node $= 1 + \left\lfloor \frac{\text{pageSize} - 2 \cdot \text{pointerSize}}{2 \cdot \text{colsize(key)}} \right\rfloor$	
d	depth of an index tree $= \left\lceil \log_F \left(\left\lceil \frac{N \cdot (\text{colsize(key)} + \text{pointerSize})}{\text{pageSize}} \right\rceil \right) \right\rceil$	
sel	query selectivity	
m	available main memory	Byte
A	set of all attributes in table	
key	indexed attribute	
$colsize(A_i)$	size of a value of attribute i	Byte
S	set of selected attributes	
p_{split}	probability of a leaf split	

TABLE 2.3: Symbols used in cost models

SV Transformation	Cost
Log SV → Row SV	$C^{\log}_{\text{scan}}(N) + C^{\text{row}}_{\text{scan}}(N)$
Log SV → Col SV	$C^{\log}_{\text{scan}}(N) + C^{\text{col}}_{\text{scan}}(N, A)$
Row SV ↔ Col SV	$C^{\text{row}}_{\text{scan}}(N) + C^{\text{col}}_{\text{scan}}(N, A)$
Row SV → Index SV	$C^{\text{row}}_{\text{scan}}(N) + \left(\frac{F^{d+1}-1}{F-1}\right) \cdot C_{\text{r}}$
Col SV → Index SV	$C^{\text{col}}_{\text{scan}}(N, \{\text{key,rowID}\}) + \left(\frac{F^{d+1}-1}{F-1}\right) \cdot C_{\text{r}}$

TABLE 2.4: Storage View Transformation Cost model

Update Cost. All updates to OctopusDB are done in the Primary Log and OctopusDB later propagates them recursively to the subsequent SVs using any appropriate maintenance algorithm. Therefore, update costs are a crucial factor when determining which SVs to keep. Table 2.2 shows our update cost model for Log, Row, Col, and Index SVs. For Row and Col SVs, we assume that either the tuples are scanned and updated in chunks of N_c; or each update triggers a random-I/O. We take the minimum cost among these two options as the update cost (as done by a cost-based optimizer). For updates in Index SVs, we also consider the costs to split leaves or nodes in the index structure ($C_{\text{split}}^{\text{index}}$). We model the probability of having a node/leaf split at a level as exponentially proportional to the depth of the level in the index tree.

For each call to **registerSV** or **registerQuery**, OctopusDB stores the reference to output SV or Query in the SV or Query Catalog respectively. Again, the holistic SV optimizer is responsible for propagating updates from the primary log to all SVs recursively. There are several ways, e.g. lazy updates, to do such SV maintenance. We believe that existing works from materialized views could be adapted in OctopusDB for SV maintenance. However, OctopusDB poses several new challenges, e.g. how to compute the optimal number of stores. This also has to consider the amount of *overlap* among stores, i.e. to avoid extensive update costs for redundant data representations.

Transformation Cost. Finally, we also model the costs to transform one type of SV to another in Table 2.4. We consider transformation as a query scan on the input SV followed by a update scan on the output SV. For Index SV, only the index attributes and rowID need to be read; the index tree needs to be built on those attributes only. The transformation cost model can be used by the holistic SV optimizer when considering to transform one SV to another, e.g. whether to transform a Row SV into a Col SV. Transformation cost is the price that OctopusDB has to pay while the benefit could be the reduced scan costs i.e. the difference between the iteration costs of the old and new SVs, or reduced update cost. As SVs are fully optional, OctopusDB may balance the two cost factors based on a given workload.

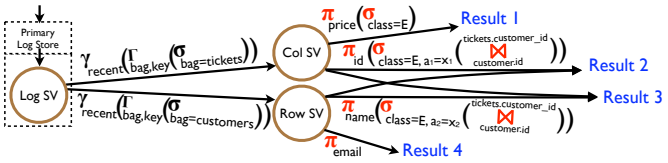
The three cost models discussed above form the backbone of the holistic SV optimizer. Based on these cost models the holistic SV optimizer can create, maintain, scan, transform, or delete any SV. Additionally, it can re-structure the SV lattice to derive maximum performance. We discuss several optimizations in the following.

2.4.3 Adaptive SV Optimization

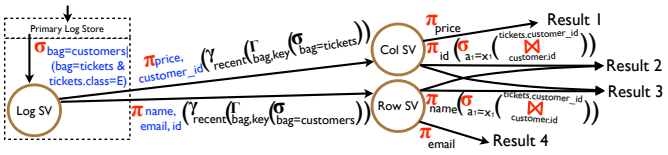
Here we discuss adaptive SV optimizations in OctopusDB.

SV Rearrangement. The holistic SV optimizer can rearrange the SV lattice in order to balance query and update costs. This implies that the SV optimizer decides how to connect the *tail* of an arrow to the existing SV lattice. One particular advantage of using a holistic optimizer is that the query operators can be pushed down through the entire SV store — even beyond the primary log.

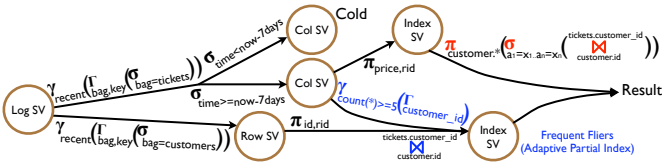
Operator Log-Pushdown. Figure 2.3(a) shows an example with four registered queries. All queries are computed based on two SVs only: a Col SV and Row SV. In this situation, the optimizer may decide to push down some of the selections and projections as follows: (1) we examine the projections of all registered queries and compute the union set of attributes, e.g. $\pi_{price,customer\_id}$ and $\pi_{name,email\_id}$. (2) we push these



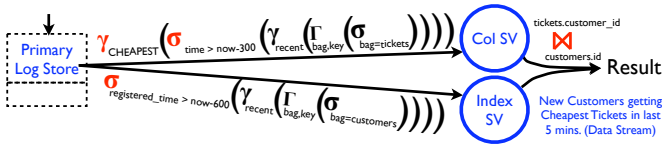
(a) Operator Log-Pushdown: before



(b) Operator Log-Pushdown: after



(c) Adaptive Partial SVs



(d) Stream Transformation

FIGURE 2.3: Workload adaption optimizations of OctopusDB for the Running Example

projections down the lattice until the primary log. Similarly, for selections we (1) compute a conjunctive selection, e.g. $\sigma_{bag=customers|(bag=tickets\&tickets.class=E)}$ and (2) push it even beyond the primary log. Figure 2.3(b) shows the resulting SV lattice. This means that any incoming log record will be checked even *before* putting it into the Log SV in the primary log. Tuples not matching will be discarded and thus we save time writing them to the primary log. Obviously, similarly to a store pushdown, as soon as a new query comes in, the conjunctive selection may have to be adapted. Otherwise we would discard too much. However in OLTP or reporting (not OLAP) workloads are often known [118] and thus a log and pushdown may be an option.

Adaptive Partial SVs. The holistic SV optimizer can inject additional SVs to speed-up query processing. Those SVs should be created for those parts of the data that is frequently queried. For instance, it does not make sense to build an index for an entire relation if only parts of that relation are queries. This observation lead to a technique called partial indexing [114]. However, that technique can be extended to create a partial store adapting *dynamically* to the current workload. Figure 2.2(e) already showed an example for *static* SV partitioning. Figure 2.3(c) shows an example for *adaptive* partial SVs. In this example a `Frequent Fliers` Index SV is used. We use a join query selecting those customers having at least five tickets over the past week. Index SV `Frequent Fliers` will only index those customers. As soon as a customer does not

Use-Case (traditional systems)	**Storage view definition**	
	type	**example query**
row store	Row SV	any
column store	Col SV	any
PAX	PAX SV	any
fractured mirrors	Row SV and Col SV	same query for both
column groups	Row SV and Col SV	$\pi_{a_1,\ldots,a_k}$ $\pi_{a_{k+1},\ldots,a_m}$
index	Index SV	any
indexed row store	Index SV(Row SV)	any
indexed column store	Index SV(Col SV)	any
read-optimized indexed column store + differential write-optimized row store	Index SV(Col SV) Row SV	$\sigma_{t<\text{now}()-1\text{day}}$ $\sigma_{t\geq\text{now}()-1\text{day}}$
partial index	Index SV	$\sigma_{420\leq a_k\leq 42000}$
projection index	Col SV	π_{a_k}
partial projection index	Index SV(Col SV)	$\pi_{a_k}(\sigma_{420\leq a_k\leq 42000})$
DSMS	Index SV	$\sigma_{t\geq\text{now}()-5\text{min}}$
DSMS + archive	Index SV and Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
snapshot	any	any
replicated row store	Row SV Row SV	same query for both
query	any	any
dynamic view	any	any
materialized view	any	any

Use-Case (new system)	**Storage view definition**	
	type	**example query**
OLTP + OLAP	Row SV Col SV	$\sigma_{t\geq\text{now}()-1\text{day}}$ $\sigma_{t<\text{now}()-1\text{day}}$
DSMS + OLTP	Index SV Row SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
DSMS + archive OLTP + archive OLAP	Index SV Row SV Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{\text{now}()-1\text{day}\leq t<\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-1\text{day}}$
other hybrid	any combination of the above	any

TABLE 2.5: Use-Cases of OctopusDB

qualify as a frequent flier anymore, its entry will be dropped from the index. Vice versa, if customers qualify, they will be added to the index dynamically.

Stream Transformation. In OctopusDB, any incoming log record corresponds to an event or item in a data stream system. For applications having continuous queries, we may only select a *window* of interest over the *unbounded* stream of log records i.e. the primary logical log in OctopusDB. This means the "database store" simply consists of several windows of interest. No other (older) data needs to be kept. OctopusDB can mimic this as follows: (1) do not use a Log SV for the Primary Log Store. (2) route all incoming log records to all relevant queries, (3) push possible updates up the SV lattice. In other words, we are reducing the stream processing problem to a *SV maintenance problem*. Figure 2.3(d) shows an example. In the Running Example, suppose the query workload changes to the following query: find new customers (registered within last 10 minutes) having booked the cheapest tickets in the last 5 minutes. For this we need to run a join on two windows. Whenever the contents of one of the windows changes, we may have to update the result to the join. However, this is nothing different from SV maintenance. Thus OctopusDB may use any known technique for updating query

results including push-based query DAGs or batch-oriented (out-of-order) updates [87]. Notice that any update to the join result is passed outside by providing an appropriate `callback` functon to `registerQuery` when registering the join. For any update on the join, `callback` will be called offering an iterator with the changed result tuples or an iterator of the complete new result.

Other Use-Cases. By creating the right SVs OctopusDB can mimic a variety of system. Furthermore, by combining different SVs OctopusDB can emulate newer hybrid systems, for instance combinations of continuous and store (=archival) queries which has been researched heavily in the past years [47]. Table 2.5 lists several use-cases for OctopusDB.

2.5 Purging and Checkpointing

Even given large amounts of main memory and external storage to keep the log, eventually the log may become too large to compute query results efficiently. This will only happen if the update rate is too high or the database has been up for a while and collected a long log of change operations. In this situation we need to shrink the size of the log. There are several options:

(1.) purge: Replace log records for data that is not of interest anymore. Here 'interest' is defined by the application, e.g. changes older than two years are irrelevant for most OLTP apps.
(2.) compress: Compress the log. This saves storage space using zip, dictionary compression, etc.. However, this does not decrease the number of log records. The log may still be too large.
(3.) checkpoint: Replace parts of the log by any storage view. This means, we write a *begin checkpoint log record* to the log. Then we create a storage view for all log records older than the begin checkpoint log record. We write the contents of the storage view to the log or other stable storage. When finished we write an *end checkpoint log record*. Then we purge all log records older than the begin checkpoint log record. The storage views used for checkpointing are never changed and therefore should be optimized for reading. Depending on the storage view we use for a checkpoint, we obtain the following different strategies:

(a) archive: Use a RowSV or ColSV. This means we keep all data but put it to a row store or col store for faster access.

(b) aggregate: Use an aggregated SV, i.e. aggregate part of the log. We may aggregate on any attribute. We could also aggregate over time, e.g. keep only one change per week could be of particular interest or even completely key-consolidate the checkpoint.

(c) re-checkpoint: Replace an existing checkpoint in the log with a derived checkpoint. For instance, assume we wrote an archive checkpoint using Row SV before. Later on we might decide that it consumes too much storage space, so we decide to replace it by an aggregated checkpoint. Like that we could devise a system that keeps an aggregated archive for everything older than 5 years, plus a detailed archive for everything younger than five years and older than one year, plus a log for everything younger than a year. If necessary, all incoming queries will be routed to all three SVs.

The final strategy is dependent on storage cost (which are low nowadays) and query cost. Note again that in this type of a system there is a gradual transition from a streaming system (use input log records for continuous query processing when they arrive and then purge them) to an OLTP system (keep fine-granular data forever and curate it) and then to an OLAP system (create aggregated snapshots of the data). The beauty of OctopusDB architecture is: first, OctopusDB could allow for all three types of analyses inside a single system without the need to stitch three different database systems together. Second, OctopusDB could allow for a gradual transition among the three types. This is hard to achieve with three separate systems.

2.6 Recovery

Logical Recovery. Recovery highly depends on the purging strategy used. So let's first assume that no data was purged or checkpointed. Recall that OctopusDB keeps at least one copy of the log on durable storage. Then, recovery is rather simple: we simply copy the log from durable storage to main memory. When that copy is finished, OctopusDB is fully recovered and may accept new queries. In the background OctopusDB will then re-create all storage views that existed before the crash. Note that the recovery process does not have to put any information on the progress information into the log, e.g. like compensation log records in ARIES [90]. This substantially simplifies the code base of our system. Now assume that OctopusDB crashes during recovery. There are two cases: (1) crash during log copying: OctopusDB simply reads the log at restart anyway, no extra logic required, (2) crash during storage view rebuilt: again we could simply rebuild all storage views. As an alternative we could keep track of storage views that were recreated. As soon as a storage view is persisted on durable media, we write an extra *SV created log record* to the log containing a checksum of the persisted storage view (following WAL). During recovery we may then skip recreation of this storage view if the checksum of the materialized storage view corresponds to the checksum in the log. Again, the latter option is just an add-on to speed-up recovery but not required for correctness. Given the bandwidth of current external storage, e.g. 100 of MB/s for single hard drives up to 500 MB/s for PCI-attached flash storage like FusionIO's iodrive [55], it might not pay off to implement this option.

So let's assume that the log is not complete anymore and was purged or checkpointed using any strategy as described in Section 2.5. In this case we need to analyze the log as follows: we read the log sequentially starting from the oldest entry. We collect all begin checkpoint log records and put them into a *checkpoint* set. If for any begin checkpoint log record, we find an end checkpoint log record, we remove the corresponding checkpoint from the set. If after reading the log *checkpoint* is empty, we proceed as if no log purging or checkpointing has ever happened. Otherwise we copy the log to main memory, however ignore all checkpoints missing an end checkpoint log record. After copying this partial log, OctopusDB is recovered. After that, in the background we re-create all checkpoints that did not have an end checkpoint log record.

ARIES-style Physiological Recovery. One might argue that OctopusDB's recovery algorithm gives away some performance by not using page-oriented (physiological) REDO as in ARIES[4]. One of the major performance advantages of ARIES stems from this idea. However, OctopusDB could easily be extended to keep physiological redo

[4]Note that UNDO is logical in ARIES anyway.

information as well. The trick is to write physiological REDO information for each SV separately. For instance, assume we want to implement a Row SV. Then we keep an extra log with physiological REDO information *inside* the implementation of Row SV. If we have to recover Row SV, we simply perform redo based on that internal log. UNDO may still be performed using the global logical log. Conceptually, this algorithm then does not differ from ARIES anymore. Hence, we expect the same performance characteristics. But clearly, we will collect more experimental evidence as soon as our system matures.

2.7 Transactions and Isolation

In the following we will discuss how to support concurrent execution of transactions in OctopusDB. Some recent work claimed that for a pure OLTP-main memory system concurrent transactions may not be required anyway. This is due to the fact that the useful amount of work of a transaction in an OLTP-system may typically be less than a millisecond. Then a serial execution of transactions might not hurt performance too much. However, the exact workload has to be known in advance. See Section 2.2 of [118] for details. Here, we do not make these assumptions. Concurrent execution of transactions could be supported in OctopusDB by extending its system interface with three methods:

beginTA()→ taID: starts a transaction and returns a system-generated transaction ID.
commitTA(taID)→ bool: commits transaction having taID. Returns true if successful, false otherwise.
abortTA(taID)→ bool: aborts transaction having taID. Returns true if successful, false otherwise.

Furthermore, we extend the methods of OctopusDB's system interface (Section 2.2.3) to receive an additional taID parameter. Thus, every call to the system (as well as its corresponding log record) is associated to a taID and we may define arbitrary transaction sequences. For instance, two concurrent transactions could call OctopusDB's system interface as shown in Table 2.6.

ts	Transaction 42	Transaction 43
1	beginTA()→ 42	
2	snapshot(myView, myQuery)	
3		beginTA()→ 43
4	query(INSERT <42, "students", 11,<"joe", 22>>)	
5	commitTA(42)→ OK	
6		query(INSERT <43, "students", 12, <"jim", 33>>)
7		abortTA(43)→ OK

Table 2.6: Sample execution of two concurrent transactions

Now let's discuss how to achieve ACID in OctopusDB. As in DBMSs, *Consistency* may be guaranteed by validating a set of integrity constraints at commit time[5]. The *Isolation* algorithm of OctopusDB is a variant of optimistic concurrency control. Its

[5]Note that one of the world's most popular open source databases, MySQL 5.1 based on the default MyISAM storage engine, completely ignores integrity constraints.

core idea is to append all changes (uncommitted or committed) to the log but only to propagate committed data to any derived storage view. Thus at all times derived storage views contain a consistent snapshot of committed data only. Uncommitted transactions are only allowed to *read* committed data either by retrieving it from the log or any derived storage view. All uncommitted transactions are allowed to *write* any data object they desire by adding log records to the log, but: the latter modifications are *not* yet propagated to the derived storage views. In addition, we also collect the unique IDs of databases objects read by transactions. This means, whenever a transaction issues a **snapshot** call, we retrieve the contents of that view, project it to (bag,key), and put it into the log associated with a read timestamp. Like this, we have all information at hand we require to run a textbook-style optimistic concurrency control algorithm, e.g. Section 17.6 in [105]. Now, let's assume we want to commit a transaction. To do so we call Algorithm 2.8.

Algorithm 2.8: commitTA

```
Input  : Integer taID
Output: bool
 1 LogRecord newLR = new WantsToCommitLR(taID,ts) ;
 2 this.getPrimaryLog().append(newLR);
 3 if this.getPrimaryLog().validate(taID) then
 4 |   LogRecord newLR = new CommitLR(taID,ts) ;
 5 |   this.getPrimaryLog().append(newLR);
 6 |   Iterator itSV = StorageViewCatalog.entries();
 7 |   itSV.open();
 8 |   while itSV.hasNext() do
 9 |   |   StorageView nextSV = itSV.next();
10 |   |   Iterator itLR = lthis.getPrimaryLog().iterate(P:= lr.taID=taID);
11 |   |   itLR.open();
12 |   |   while itLR.hasNext() do
13 |   |   |   LogRecord nextLR = itLR.next();
14 |   |   |   Source(nextSV, nextLR);
15 |   |   end
16 |   |   itLR.close();
17 |   end
18 |   itSV.close();
19 |   return true;
20 else
21 |   LogRecord newLR = new AbortLR(taID) ;
22 |   this.getPrimaryLog().append(newLR);
23 |   return false;
24 end
```

We first put a special WantsToCommitLR into the log (Lines 1–2). Then we run the validation phase of optimistic concurrency control (Line 3). If validation was successful, we insert a CommitLR into the log (Line 4). Then we propagate the changes introduced by this transaction to all storage views (Lines 6–18). If validation fails, we create an AbortLR and insert into the log (Lines 21–22).

Using this algorithm *Atomicity* is trivial as only transactions having a commit log record or are reflected in a storage view need to be considered by other operations. The same holds for *Durability*: as mentioned above, OctopusDB follows WAL anyway. Therefore all log records are durable. Note again that our log does neither need undo, redo, before or after images of pages, nor compensation log records to achieve idempotency. These issues only occur if log records are condensed into a store in the first place as in current DBMSs. In contrast, in OctopusDB storage view creation and maintenance is a *secondary* process.

Notice that the propagation process of Lines 6–18 is a possible synchronization bottleneck similar to standard view maintenance in current DBMSs, i.e. all (storage) views have to be brought to the *same* consistent state. It could be interesting to improve this to enable eventual or timeline consistency among storage views, i.e. trade consistency for performance. Several papers have argued for weaker forms of consistency, e.g. eventual consistency [102] or PNUTS' timeline consistency model [39]. We believe that these algorithms could also be adapted to improve storage view maintenance in a single database. Again, this chapter only opens the book for OctopusDB. We will explore this in more detail in future work.

2.8 Experimental Evidence

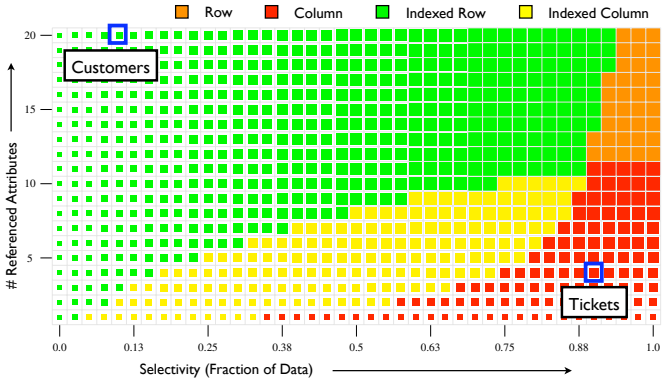
2.8.1 Workload-aware SV Selection

In this simulation we compare several SVs over varying workloads and show the effectiveness of each on a per relation basis. Furthermore, we demonstrate that it makes sense to have several SVs in OctopusDB as compared to a fixed store (row, column) in traditional databases. Figures 2.4(a) and 2.4(b) show the SVs with cheapest query and update costs respectively, in Picasso style diagrams, over varying selectivity and referenced attributes. We use the cost models shown in Tables 2.1 and 2.2 to compute the query and update costs for each SV. For each selectivity-attributes pair, we depict the cheapest SV by a square of size proportional to the fraction of its cost to the maximum cost in the entire space. In both the figures we can observe four distinct regions over the selectivity-attributes space, in which row, column, indexed row, and indexed column, respectively, are the cheapest SVs. Specifically, as expected, we observe that less number of referenced attributes favor column SVs compared to row SVs. In addition, high selectivity favors index SVs compared to unindexed ones.

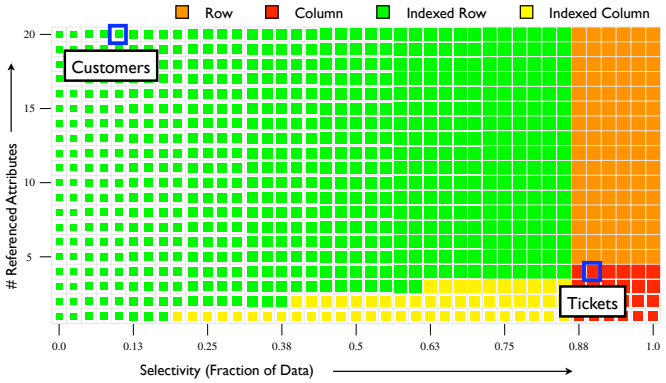
2.8.2 Outperforming Traditional Systems

The goal of this simulation is to compare OctopusDB with several existing database systems. We consider Row Store, Column Store, Index Row Store, Indexed Column Store, Fractured Mirrors and Indexed Fractured Mirrors to compare with OctopusDB. The workload consists of equal number of scan and update queries on Tickets and Customers relations in the Running Example. Table 2.7 shows the workload parameters. Figure 2.5 shows the overall performance of different systems in terms of the overall workload time. We can see from the figure that OctopusDB outperforms traditional database system by a factor of up to 5. Furthermore, we break down the workload time into query and update costs. We observe that OctopusDB outperforms all other systems in terms of update costs while only Indexed Fractured Mirrors match OctopusDB in terms of query costs. Indexed Fractured Mirrors, however, has prohibitive update costs.

The reason for better performance of OctopusDB compared to traditional systems is the flexible SV management in OctopusDB depending upon the workload. The Tickets and Customers relations in the above workload are marked in Figures 2.4(a) and2.4(b) for the scan and update queries respectively. From the figures, we observe that a column SV on Tickets and an Indexed Row SV on Customers provide the cheapest costs. Such



(a) Query Costs



(b) Update Costs

FIGURE 2.4: Query and Update Costs over varying Workload

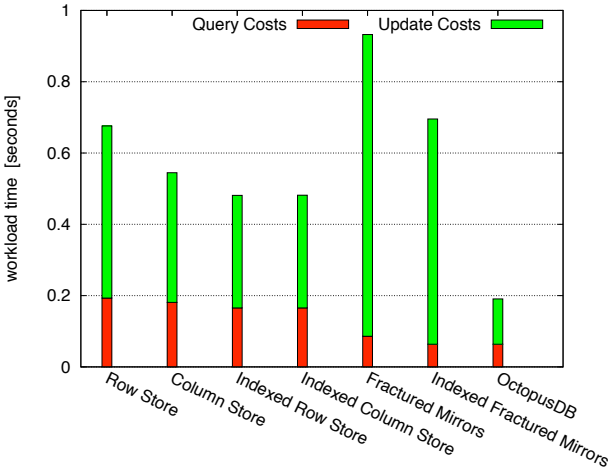


FIGURE 2.5: Workload costs for different systems

a SV configuration is impossible to achieve in a conventional database system, whereas OctopusDB can easily cope with it by creating the appropriate SVs.

Parameter	Tickets	Customers
N	100,000	20,000
N_c	50,000	50,000
BW	100 MB/sec	100 MB/sec
C_{random}	0.005 sec	0.005 sec
$pageSize$	4 KB	4 KB
$pointerSize$	4 Bytes	4 Bytes
sel	0.9	0.1
m	100 MB	100 MB
$\|A\|$	20	20
$keySize$	4 Bytes	4 Bytes
$colsize(A_i)$	8 Bytes	8 Bytes
$\|S\|$	4	20
p_{split}	1%	1%

TABLE 2.7: Parameters used in the Simulations

2.8.3 Automatic Adaptation

In this experiment we show how OctopusDB may automatically adapt SVs to a query response time requirement. We assume that the user specified that workload should not take longer than 0.1 sec.

System and Setup. Our current prototype of OctopusDB is implemented in Java 1.6. All experiments were executed on a medium-sized computing node. We used a single Intel Xeon Quadcore, 2.66Ghz (E5430) with 16GB of main memory. The operating system was Linux 2.6.27.7-9-xen. In order to have a realistic scenario, we assumed all data and storage views to fit into main memory. Note again, that OctopusDB is not limited to main memory scenarios but could also be run as an external memory system. We use Tickets and Customers records having 20 attributes each for our experiment. For each measurement, the workload contains a batch of 40 randomly picked scan and update queries in the ratio 1:3. We pick the search attribute for scan queries using zipfian distribution with a skewness factor of 4; scan queries have a selectivity of 0.01.

Experiment. We configured OctopusDB to use Log SV only and gradually increase the number of log records. We monitor a window of 5 latest measurements and take the average. If the average is above 0.1sec, we switch to a more efficient SV. Figure 2.6 shows the results. We observe that for up to 33,000 log records, the requirement is met by Log SV. Then, OctopusDB switches to Bag-partitioned Log SVs for Tickets and Customers relations. This works fine until 76,000 log records. Here, OctopusDB switches Tickets Log SV to Key-Consolidated Log SV. Again at 103,000 log records OctopusDB Key-Consolidates the Customer Log SV. This works fine until 127,000 log records where we switch Tickets to Column SV. At 200,000 log records, we switch Customers to Row SV. Finally, we switch Tickets and Customers to Indexed SVs at 206,000 and 211,000 log records respectively.

This experiment demonstrates that through a wide range of database sizes, scan-based methods, be it a log, a partitioned log, a key-consolidated log, a row store, or a column store may be enough to serve a workload. Only for larger database sizes we have to use indexing. Note that Indexed SV may also be used to create indexes only on a subset of attributes. This is useful if some attributes are queried more often than others.

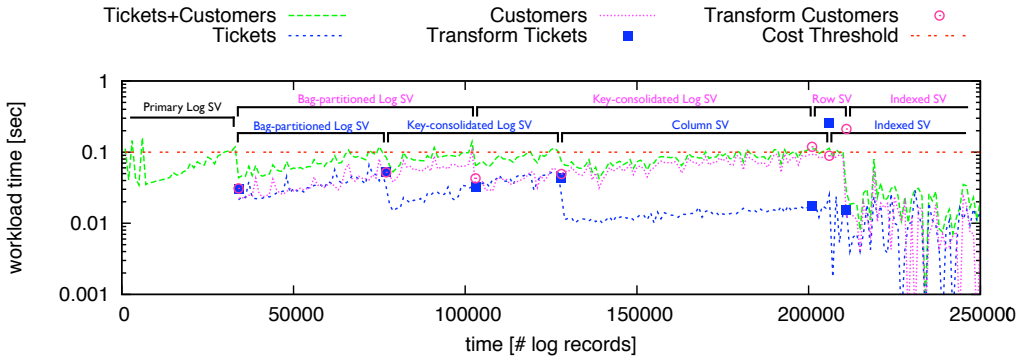


FIGURE 2.6: OctopusDB: automatic adaption of SVs for TICKETS and CUSTOMERS to query time requirement of 0.1sec when scaling database size. The figure shows the evolvement of workload time for both relations, the sum, as well as the SV transformation costs.

2.9 Related Work

Traditional DBMS Landscape. Several papers have claimed that one size does not fit all. It started with [53] who noted that DBMSs do not work well for DSS-type workloads. This work lead to one of the first column-oriented data warehouses: SybaseIQ. Later on, other authors supported the idea of different types of database systems for different markets as well [115, 117]. This split the landscape into at least four different systems: SearchEngines (read-only inverted index), OLTP (transactional row-store), OLAP (read-only column-store) and DSMS (continuous window queries on unbounded streams) which originated from append-only databases [122]. However, the major difference of OLTP and OLAP are different access patterns and missing ACID semantics in OLAP. Furthermore, it is somewhat easier to use compression in column stores, see [4] for a comprehensive tutorial. However, a recent paper has argued that column stores may be efficiently emulated on row stores as well [18]. In any case, note that OctopusDB may make use of the existing optimizations for column stores and/or row stores. The advantage of OctopusDB is that we are not restricted to a particular store and workload. In addition, as we show in this chapter, the boundary of OLTP/OLAP and DSMS may also be removed into a unified OctopusDB system. Furthermore, there have been several efficient approaches already to implementing search engines as an *application* on top of an OLTP database [69, 123]. Thus, given these and other recent advancements, it is questionable whether search engines will survive as a separate code base.

Lightweight Systems. Recently, a paper claimed that even in the traditional OLTP market existing DBMSs can be beaten by a large factor [81, 118]. This approach, HStore, is basically a stripped down version of an OLTP row-store. Again, this stripped down system could be emulated in OctopusDB as well. Thus, HStore is orthogonal to what we propose. Another line of systems has recently appeared exploring array-oriented systems for scientific applications [41]. Their major difference is an array data model with additional scientific operators. We believe that OctopusDB could also be extended to offer an Array Storage View. However, that discussion is beyond the scope of this chapter.

Scanning Systems. Due to the dramatic changes in hardware (random access is hardly becoming better, sequential access is improving by up to 50% per year), index access pays of less and less. Therefore several authors have proposed to drop indexed-based query plans entirely and resort to scanning. [70] proposes techniques to reduce the scanning costs on modern hardware architectures. [107] proposed per-tuple constant-time processing on a row-store. [136] examines shared-scans, i.e. multiple query results are computed using a single scan. This idea is extended in [128] to so-called clock-scans, i.e. continuously running shared scans. Finally, [23] proposed to compute multiple star queries simultaneously using a static shared operator pipeline and a clock scan as its input. All these techniques show the viability of scan-based plans and we believe that all those techniques may be integrated into OctopusDB. Still we believe that a system should be able to offer index access for highly selective queries as well. OctopusDB offers this option.

Rodent store [42] allows DBAs to declare the database store using an algebra. We agree with the authors of [42] that currently considerable functionality is duplicated as row-stores and column-stores are two separate lines of development that share however considerable common technology. However, Rodent Store still assumes that there has to be a store. Furthermore, it simply provides an abstract way to declare a store in an OLTP or OLAP-style system. No unified approach with streaming systems is provided as it is the case for OctopusDB. In addition, no updateable storage view mechanism is present in Rodent Store. Another key assumption in Rodent Store is that the database administrators will manually specify the storage algebra for each database. This is in sharp contrast to OctopusDB, which automatically adapts itself to the incoming query workload. Alternatively, Rodent Store also provides a storage design optimizer to recommend the storage representation for a given query workload. OctopusDB, on the other hand, does not assumes a static query workload. Rather, OctopusDB automatically adapts the storage views to continuously changing query workloads. Still we believe that the storage algebra concept in Rodent Store is an elegant way to express physical layouts. We could reuse this concept in OctopusDB to internally map a storage view to its on-disk representation.

On similar lines as Rodent Store, GMAP [127] presents a DDL for defining physical structures (similarly to [42]). However, in contrast to OctopusDB, GMAP does neither handle unification with streaming systems, automatic store selection and adaption, recovery, nor union queries.

Cracked databases, e.g. [71], break database tables into pieces by piggy-backing index-reorganization requests to individual queries, i.e. queries are interpreted as hints to break the database into pieces. Therefore cracked databases have similarities with partial indexing [114] and adaptive indexing [51], i.e. dynamic adjustment of index granularity to a given workload. Furthermore in contrast to OctopusDB, cracked databases assume a column-store (the authors mention that it could work on a row-store as well). In contrast, we do not assume a fixed store. In addition, in cracked databases the store may not be exchanged as it is the case in OctopusDB. Therefore cracked databases are orthogonal to what we propose. At the core, cracked databases maintain only a single adaptive storage structure (cracker columns) and adapt only to selection query predicates. On the other hand, OctopusDB can have several storage views and can adapt to any arbitrary query sub-expressions, such as selection, projections, joins etc. Furthermore, database cracking can take several thousand queries before it optimizes

the data fully, i.e. builds the cracker index fully [58]. OctopusDB does not have such a constraint. It can create an index storage view much more quickly. More recent work on cracking [74] has proposed to merge database cracking with adaptive merging [58, 59], in order to quickly achieve query performance comparable to a full index. However, database cracking still imposes a high overhead for the first few queries because of the large parts of the data being shuffled and reorganized. OctopusDB on the other hand keeps a light weight logical log to start with. Later it creates additional storage structures as and when required. Still, as part of future work it might be interesting to extend some of the cracking ideas to work on our more general storage view architecture.

Materialized views are arbitrary query results stored on disk [35]. There has been a lot of research on answering queries using materialized views [7, 36, 57, 103]. However, materialized view creation is an expensive one time process, typically with the help of a database administrator. Instead, OctopusDB automatically creates ad hoc storage views to continuously adapt to the query workload. Furthermore, with materialized views, the database administrator only decides *what* (query results) to store. However, with storage views, OctopusDB decides both *what* and *how* (layout) to store. A recent work, dynamic materialized views [134], materializes the frequently accessed rows dynamically. However, the data is still stored in the standard row layout.

Log-structured databases, such as Hyder [15] and Rose [113], store data as a log. Hyder runs on a cluster of servers with data-sharing architectures, i.e. having shared access to a large pool of storage. As a result, Hyder achieves scale-out without partitioning the data. Hyder also has an index structure, a search tree, which is marshaled into the log. It supports both read and write operations on indexed log records. Similarly, Rose uses log structured merged trees to provide better write and scan throughput than B-trees. However, both Hyder and Rose do not consider storage structures other than the log. OctopusDB goes much beyond these systems in the sense that it supports arbitrary physical representations of data, the log being just one instance of it.

Extensible databases such as Exodus [25], Shore [26], and Genesis [13] enable features to be added or removed from a general purpose DBMS in order to match the needs of a target application more closely. For example, Exodus (later succeeded as Shore) provided a flexible API for building an object-oriented database systems. However, it is limited to file, B+tree, and R*tree storage structures. OctopusDB, on the other hand, is not limited to a set of storage structures; it can create any arbitrary physical representations of data. The Genesis project recognized the need for supporting new storage structures in order to handle a variety of database applications. To do so, Genesis takes a building block approach to DBMS construction: it provides software modules which can be composed differently to produce different (customized) DBMSs. In contrast to Genesis, OctopusDB is not recomposed from scratch each time it sees a different workload. Rather, OctopusDB adapts to the workload over a period of time. On similar lines as Genesis, MySQL [94] has a pluggable storage engine architecture, which allows users to choose from the several available storage engines for their application. Each storage engine is optimized for a specific application, e.g. InnoDB for OLTP, MyISAM for read intensive web scale-out, NDB for high-availability clustering, and Archive storage engine for data archiving. However, at any given time, MySQL allows for only a single storage engine. In contrast, OctopusDB can make use of several storage views.

Finally, self-tuning DBMSs [30] have looked at ways to automate index selection. We believe that some of these techniques may be extended to support OctopusDB. At the same time we believe that automatic storage view selection in OctopusDB is a hard problem justifying a separate research study.

2.10 Conclusion

This chapter has proposed a unified database architecture which could support several types of workloads. The primary storage structure of OctopusDB is a logical log. All other storage structures are just secondary *storage views* on this log. Storage views include row stores, column stores, indexes, but also windows on unbounded streams. With OctopusDB we are inverting the traditional development of a DBMS: previously there always was a specific store which was an irrevocable design-decision, built-in into the DBMS. On top of that, an ARIES-style [90] log-based recovery was implemented to guarantee atomicity and durability. In this chapter, we took exactly the opposite approach: we started with the log (which is totally disconnected from any store) and if necessary, we define optional SVs on that log suited for a particular workload.

We have presented several use-cases for OctopusDB. We have also provided a cost model for querying, updating, and transforming SVs. A simulation and experiment with a prototype of OctopusDB demonstrated the feasibility and efficiency of our idea. That experiment also showed that OctopusDB may pick an SV adapting automatically to a given workload to fulfill a query response time guarantee. OctopusDB has several advantages: it unifies existing architectures into a single system, it has a simple interface, and it reduces administration overhead. Several things may be explored in more detail including some algorithmic details.

Chapter 3

Indexing and Join Techniques for Large Scale Data Management

MapReduce is a computing paradigm that has gained a lot of attention in recent years from industry and research. Unlike parallel DBMSs, MapReduce allows non-expert users to run complex analytical tasks over very large data sets on very large clusters and clouds. However, this comes at a price: MapReduce processes tasks in a scan-oriented fashion. Hence, the performance of Hadoop — an open-source implementation of MapReduce — often does not match the one of a well-configured parallel DBMS. In this chapter we propose a new type of system named Hadoop++: it boosts task performance without changing the Hadoop framework at all (Hadoop does not even 'notice it'). To reach this goal, rather than changing a working system (Hadoop), we *inject* our technology at the right places through UDFs only and affect Hadoop *from inside*. This has three important consequences: First, Hadoop++ significantly outperforms Hadoop. Second, any future changes of Hadoop may directly be used with Hadoop++ without rewriting any glue code. Third, Hadoop++ does not need to change the Hadoop interface. Our experiments show the superiority of Hadoop++ over both Hadoop and HadoopDB for tasks related to indexing and join processing.

3.1 Introduction

3.1.1 Background

Over the past three years MapReduce has attained considerable interest from both the database and systems research community [5, 6, 22, 27, 34, 37, 38, 56, 76, 92, 99, 101, 119, 124, 132].

There is an ongoing debate on the advantages and disadvantages of MapReduce versus parallel DBMSs [44, 46]. Especially, the slow task execution times of MapReduce are frequently criticized. For instance, [101] showed that shared-nothing DBMSs outperform MapReduce by a large factor in a variety of tasks.

Recently, some DBMS vendors have started to integrate MapReduce front-ends into their systems including Aster, Greenplum, and Vertica. However, these systems do not change the underlying execution system: they simply provide a MapReduce front-end to

a DBMS. Thus these systems are still databases. The same holds for a recent proposal from VLDB 2009 [5]: *HadoopDB*. It combines techniques from DBMSs, Hive [124], and Hadoop. In summary, HadoopDB can be viewed as a data distribution framework to combine local DBMSs to form a *shared-nothing DBMS*. The results in [5] however show that HadoopDB improves task processing times of Hadoop by a large factor to match the ones of a shared-nothing DBMS.

3.1.2 Research Challenge

The approach followed by HadoopDB has severe drawbacks. First, it forces users to use DBMSs. Installing and configuring a parallel DBMS, however, is a complex process and a reason why users moved away from DBMS in the first place [101]. Second, HadoopDB changes the interface to SQL. Again, one of the reasons of the popularity of MapReduce/Hadoop is the simplicity of its programming model. This is not true for HadoopDB. In fact, HadoopDB can be viewed as just another parallel DBMS. Third, HadoopDB locally uses ACID-compliant DBMS engines. However, only the indexing and join processing techniques of the local DBMSs are useful for read-only, MapReduce-style analysis. Fourth, HadoopDB requires deep changes to glue together the Hadoop and Hive frameworks. For instance, in HadoopDB local stores are replaced by local DBMSs. Furthermore, these DBMSs are created outside Hadoop's distributed file system thus superseding the distribution mechanism of Hadoop. We believe that managing these changes is non-trivial if any of the underlying Hadoop or Hive changes[1].

Consequently, the research challenge we tackle in this chapter is as follows: is it possible to build a system that: (1) keeps the interface of MapReduce/Hadoop, (2) approaches parallel DBMSs in performance, and (3) does not change the underlying Hadoop framework?

3.1.3 Our Solution

Overview. Our solution to this problem is a new type of system: *Hadoop++*. Hadoop++ operates exactly as MapReduce by passing the same key-value tuples to the map and reduce functions. Similar to HadoopDB, Hadoop++ also allows us: (i) to perform index accesses whenever a MapReduce job can exploit the use of indexes, and (ii) to co-partition data so as to allow map tasks to compute joins results locally at query time. However, we show that in terms of query processing Hadoop++ matches and sometimes improves the query runtimes of HadoopDB. The beauty of our approach is that we achieve this *without changing the underlying Hadoop framework at all*, i.e. without using a SQL interface and without using local DBMSs as underlying engines. We believe that this non-intrusive approach fits well with the simplicity philosophy of Hadoop.

Hadoop++ changes the internal layout of a *split* — a large horizontal partition of the data — and/or feeds Hadoop with appropriate UDFs. However, Hadoop++ does not change anything in the Hadoop framework.

[1] A simple example of this was the upgrade from Hadoop 0.19 to 0.20 which affected principal Hadoop APIs.

3.1.4 Contributions

In this chapter we make the following contributions:

(1.) MapReduce Expressiveness. We show that MapReduce and parallel DBMS have the same expressiveness. To demonstrate this, we extend standard relational algebra by a multimap operator mapping an input item to a set of output items. We show that any task specified in MapReduce can be expressed by a query in extended relational algebra. After that, we also show that any relational algebra plan may be expressed in MapReduce. As a consequence, we conclude that both technologies have the same expressiveness. (Section 3.2)
(2.) The Hadoop Plan. We demonstrate that Hadoop is nothing but a hard-coded, operator-free, physical query execution plan where ten *User Defined Functions* `block`, `split`, `itemize`, `mem`, `map`, `sh`, `cmp`, `grp`, `combine`, and `reduce` are injected at predetermined places. We make Hadoop's hard-coded query processing pipeline explicit and represent it as a DB-style physical query execution plan (*The Hadoop Plan*). As a consequence, we are then able to reason on that plan. (Section 3.3)
(3.) Trojan Index. We provide a non-invasive, DBMS-independent indexing technique coined *Trojan Index*. A Trojan Index enriches logical input splits by bulkloaded read-optimized indexes. Trojan Indexes are created at data load time and thus have no penalty at query time. Notice, that in contrast to HadoopDB we neither change nor replace the Hadoop framework *at all* to integrate our index, i.e. the Hadoop framework is not aware of the Trojan Index. We achieve this by providing appropriate UDFs. (Section 3.4)
(4.) Trojan Join. We provide a non-invasive, DBMS-independent join technique coined *Trojan Join*. Trojan join allows us to co-partition the data at data load time. Similarly to Trojan Indexes, Trojan Joins do neither require a DBMS nor SQL to do so. Trojan Index and Trojan Join may be combined to create arbitrarily indexed and co-partitioned data inside the same split. (Section 3.5)
(5.) Experimental comparison. To provide a fair experimental comparison, we implemented all Trojan-techniques on top of Hadoop and coin the result Hadoop++. We benchmark Hadoop++ against Hadoop as well as HadoopDB as proposed at VLDB 2009 [5]. As in [5] we used the benchmark from SIGMOD 2009 [101]. All experiments are run on Amazon's EC2 Cloud. Our results confirm that Hadoop++ outperforms Hadoop and even HadoopDB for index and join-based tasks. (Section 5.6)

3.2 From Relational Algebra to MapReduce and Back

The goal of this section is to show that MapReduce and DBMSs have the same expressiveness. We show that any relational algebra expression can be expressed in MapReduce. Vice versa any MapReduce task may be expressed in extended relational algebra. We extend standard relational algebra by a multimap operator mapping an input item to a set of output items. As a consequence, we conclude that both technologies have the same expressiveness.

This is a formal argument and does *not* imply that plans have to be created *physically* like this. First, we show how to map relational algebra operators to MapReduce (§ 3.2.1 to 3.2.5). Then, we show how to map any MapReduce program to relational algebra (§ 3.2.6).

3.2.1 Mapping Relational Operators to MapReduce

We assume as inputs two relational input data sets T and S containing items that are termed *records*. The schema of T is denoted $\text{sch}(T) = (a_1, .., a_{ct})$, $\text{sch}(S) = (b_1, .., b_{cs})$ respectively where $a_1, .., a_{ct}$ and $b_1, .., b_{cs}$ are attributes of any domain. In case no schema is known for the data, the schema simply consists of a single attribute containing the byte content of the item to process. In the remainder of this chapter we assume that input data sets are split into records according to the above definition. The subset of attributes in $\text{sch}(T)$ representing the key is named $k_T \subseteq \text{sch}(T)$. The remaining attributes $\text{sch}(T) \setminus k_T$ representing the value are named v_T, hence $\text{sch}(T) = k_T \oplus v_T$. This also holds for S and we use v_S and k_S accordingly. Inputs and outputs to relational operators are assumed to be duplicate-free sequences, i.e. duplicates are removed unless specified otherwise (e.g. `unionall`). `map` is called for each input record. Key and value are passed as separate parameters and a sequence of intermediate (key, value)-pairs is returned:

$$\texttt{map}(\text{key } k, \text{value } v) \mapsto [(ik_1, iv_1), \ldots, (ik_{m(k,v)}, iv_{m(k,v)})].$$

The number of intermediate output records $m(k, v) \geq 0$ may vary for different k and v. Similarly, `reduce` is called for each distinct intermediate key ik. The set of intermediate values ivs having that intermediate key is passed to `reduce`:

$$\texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [ov_1, .., ov_{r(ik,ivs)}]$$

Thus each `reduce` function produces a sequence of *output* values $ov_1, .., ov_{r(ik,ivs)}$ Again the number of output values $r(ik, ivs) \geq 0$ may vary for different inputs. In many applications the output contains a single value only, i.e. $r(ik, ivs) = 1 \; \forall \; ik, ivs$.

3.2.2 Unary operators

In the following we will show how to express relational algebra operators using MapReduce. We use $\Mapsto$ to denote how to map the left-hand side operator to a MapReduce job. The most simple operator is π. It can be expressed in MapReduce as follows:

Projection (π).

$$\pi_{a_{i_1},..,a_{i_n}}(T) \Mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Here $\oplus$ denotes that two attributes sets are concatenated to a new schema. `prj`() projects a *single* record to attributes $a_{i_1}, .., a_{i_n}$. Thus π is realized in `map` by concatenating the attributes of the key and the value, projecting to the desired attributes, and outputting the resulting records as the intermediate key. As value we output "1". `reduce` then simply outputs the intermediate key. Recall, that `reduce` is only called once for each intermediate key. Thus our definition of reduce removes all duplicates. Note that the **rename** operator ρ may be defined analogously to π.

The selection operator may be expressed as follows.

Selection (σ).

$$\sigma_P(T) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \begin{cases} [(k \oplus v, 1)] & \text{if } P(k \oplus v), \\ \text{none} & \text{else.} \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Here `map` examines each input record and passes it to the selection predicate P. If P holds, $[(k \oplus v, 1)]$ is output. Otherwise nothing is output. `reduce` simply outputs the intermediate key.

Grouping (Γ). We differentiate between grouping and aggregation. Grouping forms groups of records belonging together. For instance, assume an input T with $\text{sch}(T) = \{a_1, a_2\}$ and $T = \{(3,2),(2,1),(1,3),(2,2),(3,4),(1,7)\}$. If we group T, we obtain $\Gamma_{a_1}(T) = \{(3,\{2,4\}),(1,\{3,7\}),(2,\{2,1\})\}$. Only an additional aggregation would transform each group in a_2 into a new value. Hence, grouping may be applied without aggregation.

$$\Gamma_{a_1,..,a_n}(T) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T)\setminus\{a_{i_1},..,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus ivs)] \end{cases}$$

This means `map` concatenates attributes of the key and the value and projects them to the grouping attributes. These attributes are used as the intermediate key. All remaining attributes form the intermediate value. `reduce` then outputs a single record for each distinct intermediate key plus the set of values having that key.

Aggregation (γ). Aggregation can be done by applying an aggregation function $\text{agg}([iv_1, \ldots, iv_n]) \mapsto v$ in `reduce`. If those values were formed by a previous grouping operator, we obtain the desired result:

$$\gamma_{\text{agg}}(T) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k, v)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus \text{agg}(ivs))] \end{cases}$$

Another alternative is to combine both grouping and aggregation into a single MapReduce task:

$$\gamma_{\text{agg}}\left(\Gamma_{a_{i_1},..,a_{i_n}}(T)\right) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},..,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T)\setminus\{a_{i_1},..,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik \oplus \text{agg}(ivs))] \end{cases}$$

This means, we simply modify `reduce` to apply agg() to the output valueset. Note that agg() may be any aggregate including trivial ones such as `MIN`, `MAX`, `SUM`, `AVG`, and `DISTINCT`.

3.2.3 Binary Operators

MapReduce operates on a single input only. This means that a binary operator cannot be modeled by considering two input files. Therefore, in the following we consider the two inputs to be contained in a single file and denote this as $T|S$. When discussing individual operators we denote this as $T|S$, i.e., T is processed before S. We use a function $\text{input}(k \oplus v) \mapsto \{T|S\}$ to determine whether $k \oplus v$ belongs to T or S. Technically, this function may be implemented by attaching some metadata bit signaling its input to

each record. Recall that the precondition for union, intersect, and difference is $\text{sch}(T) = \text{sch}(S)$.

Union ($\cup$). $T \cup S \Rightarrow \gamma_{\texttt{distinct}}\left(\Gamma_{\texttt{sch}(T)}(T|S)\right)$.

This means, we express union as a grouping plus a following duplicate removal on intermediate values. This works as both input sets are already contained in the same input file.

Difference ($\setminus$).

$$T \setminus S \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k \oplus v, 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \begin{cases} [(ik)] & \text{if } |ivs| = 1 \wedge \text{input}(ik) = T, \\ \text{none} & \text{else.} \end{cases} \end{cases}$$

This means, in `map` we consider all attributes to be intermediate keys. `reduce` tests whether the size of the intermediate valueset ivs contains only a single "1" and ik belongs to input T. Only if this holds, we output ik. Otherwise nothing is output.

Intersection ($\cap$). Obviously intersection may be expressed as $T\setminus(T\setminus S)$ resulting in two MapReduce tasks. However, intersection can also be expressed in a single MapReduce job:

$$T \cap S \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(k \oplus v, 1)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \begin{cases} [(ik)] & \text{if } |ivs| = 2, \\ \text{none} & \text{else.} \end{cases} \end{cases}$$

This mapping is similar to difference, however we only output a record, if ivs contains two "1"s. As both input sets are duplicate free, this may only hold if the record is contained in both input sets.

Cross Product ($\times$). Let $\text{h}_T()$ be a hash-function defined on the key k_T of $\text{sch}(T)$. Let $D > 0$ be a constant. Then the cross product is defined as

$$T \times S \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\text{h}_T(k) \bmod D, k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(0, k \oplus v), .., (D-1, k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \Big[\texttt{crossproduct}(T_{ik}, S) \mid \\ \quad T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\ \quad S = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\}\Big] \end{cases}$$

Here `map` creates a disjoint partitioning on input T by assigning each record from T a number in $0, .., D-1$. The records from S are replicated by outputting D intermediate records covering all intermediate keys from $0, .., D-1$. The purpose of this partitioning is to allow for D reduce function calls and thus a concurrent execution on different nodes[2]. Inside a `reduce` call the input set ivs is split into two subsets T_{ik} and S. On these sets we then compute the cross product using the *local* function `crossproduct`.

[2]Note that h_T is orthogonal to the partitioning function `sh` (§3.3). The former hashes records to `reduce` functions, the latter hashes `reduce` functions to reduce tasks.

Join ($\bowtie$). Joins may be expressed as $\sigma_{PJ}(T \times S)$ resulting in two MapReduce tasks where one is based on a cross product. Obviously this does not scale for large input sets. We therefore show below a more efficient variant.

$$T \bowtie_{PJ(T,S)} S \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\texttt{prj}_{a_i}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(\texttt{prj}_{b_j}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ \Big[\texttt{crossproduct}(T_{ik}, S_{ik}) \mid \\ \quad T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\ \quad S_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\}\Big] \end{cases}$$

This means that `map` re-partitions both inputs T and S into co-partitions T_{ik} and S_{ik} where the join attributes inside a copartition have the same value. It then suffices to call `crossproduct` for these copartitions. Note that this is similar to a standard relational sort-merge join in the following way: it has to perform a nested-loop, i.e. cross product, on the records having the same value for their join attribute. Also note that for those cases where the join attribute is skewed in a way that the input becomes too large to fit into main memory, the call to `crossproduct` may perform a block-based nested-loop join similar to DBMSs.

3.2.4 Extended Operators

We discuss an additional operator that does not effect the expressiveness of MapReduce but is useful in the following discussion.

Sort.

$$\texttt{sort}_{a_{i_1},\ldots,a_{i_n}}(T) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{prj}_{a_{i_1},\ldots,a_{i_n}}(k \oplus v), \texttt{prj}_{\text{sch}(T) \setminus \{a_{i_1},\ldots,a_{i_n}\}}(k \oplus v))] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [\{ik\} \times ivs] \end{cases}$$

This mapping rule is somewhat surprising as neither `map` nor `reduce` perform an actual sort operation. The correctness of this mapping rule is guaranteed as the MapReduce and Hadoop frameworks preserve interesting orders [45]. Finally, let us stress that all other operators (e.g. division, and outer-joins) may be composed by the above operators.

3.2.5 Relational DAGs

So far we have considered single operators and provided rules to map them to MapReduce jobs. However, relational algebra expressions typically consist of multiple operators forming a *Directed Acyclic Graph* (DAG). These DAGs may be mapped to a cascade of MapReduce jobs by applying our rewrite rules recursively. Outputs of subplans are simply considered input files to the next operator. Thus each operator triggers a separate MapReduce job. For instance, $T \bowtie (S \bowtie U)$ is computed by executing a MapReduce job of §3.5 for S and U and then executing another MapReduce job on the result and T. Obviously these plans are far from optimal and may be improved in several ways. An upcoming paper discusses how to compute multi-way joins [6]. However those joins do not use co-partitioning as Trojan Join. As discussed above this has severe performance penalties. Therefore it would be interesting to extend our Trojan Join to multiway-joins. We will research this idea as part of future work.

3.2.6 Mapping MapReduce to Relational Algebra

The main idea of MapReduce is to perform an aggregation based on two user-defined functions. The purpose of the first function (`map`) is to define the items and grouping key, the purpose of the second function (`reduce`) is to define the aggregation function and the output format. One peculiarity here is that both functions may return a *sequence* of records. To express this we require a special operator that however is straightforward to integrate into an existing relational algebra: a multimap operator.

multimap operator. $\texttt{mmap}_\text{f}(T) \mapsto T'$. For each input item $t \in T$ this operator applies a function $f(t)$ generating zero, one, or multiple output items. All output items have the same schema $\text{sch}(T')$. Function f takes as its argument the entire tuple t, however the attributes of t may be passed as different arguments.

Using this operator we are able to define the backmapping rule for any MapReduce job as follows:

MapReduce Given an input set T and two UDFs, `map` and `reduce`, any MapReduce task can be expressed in relational algebra as:

$$\texttt{MR}_{\texttt{map},\texttt{reduce}}(T) \Rrightarrow \texttt{mmap}_{\texttt{reduce}}\Big(\Gamma_{ik}\big(\texttt{mmap}_{\texttt{map}}(T)\big)\Big).$$

This means, logically any MapReduce job can be expressed as a multimap operator $\texttt{mmap}_{\texttt{map}}$ followed by a grouping on the intermediate key ik and a $\texttt{mmap}_{\texttt{map}}$ using `reduce`. Notice that the main difference of γ and $\texttt{mmap}_{\texttt{reduce}}$ is that the former creates exactly one output value for an input group whereas the latter may create 0, 1, or multiple output values for an input group. Thus, in general $\texttt{mmap}_{\texttt{reduce}}$ *may* aggregate the input value set similarly to γ, but it does not have to.

$$\begin{array}{c} (3,3) \\ (2,1) \\ (1,6) \\ (2,0) \\ (3,1) \\ (1,2) \end{array} \quad \begin{array}{c} \\ \\ \texttt{mmap}_{\texttt{map}} \\ \longrightarrow \\ \\ \end{array} \quad \begin{array}{c} (3,4) \\ (2,2) \\ (1,7) \\ (2,1) \\ (3,2) \\ (1,3) \end{array} \quad \begin{array}{c} \\ \\ \Gamma_{a_1} \\ \longrightarrow \\ \\ \end{array} \quad \begin{array}{c} \\ \\ (2,\{2,1\}) \\ (1,\{7,3\}) \\ (3,\{2,4\}) \end{array} \quad \begin{array}{c} \\ \\ \texttt{mmap}_{\texttt{reduce}} \\ \longrightarrow \\ \\ \end{array} \quad \begin{array}{c} \\ \\ (1 \oplus \{7\} \oplus 7) \\ (1 \oplus \{7,3\} \oplus 10) \\ (3 \oplus \{4\} \oplus 4) \\ (3 \oplus \{2,4\} \oplus 6) \end{array}$$

FIGURE 3.1: MapReduce processing in relational algebra

Figure 3.1 shows an example for an input set T having six tuples, $\text{sch}(T) = (a_1, a_2)$ where $k_T = [a_1]$ and $v_T = [a_2]$. The map function increases all values by one, i.e. $\texttt{map} := [k, v{+}1]$. The results are then grouped (Γ_{a_1}) and fed into `reduce` which creates all subsets of value sets having a sum greater three, i.e. $\texttt{reduce} := [ik \oplus vset' \oplus \texttt{sum}(vset') \mid vset' \subseteq vset \wedge \texttt{sum}(vset') > 3]$. Each output value is the concatenation of the intermediate key, the subset and its sum. For instance, for tuple $(2, \{2, 1\})$ all subsets have a sum smaller or equal three. Thus no output is produced. For $(1, \{7, 3\})$ two subsets $\{7\}$ and $\{7, 3\}$ have a sum greater than three. Thus two output tuples $(1 \oplus \{7\} \oplus 7)$ and $(1 \oplus \{7, 3\} \oplus 10)$ are produced. Similarly for $(3, \{2, 4\})$ two output tuples are produced.

3.3 Hadoop as a Physical Query Execution Plan

In this section we examine how Hadoop computes a MapReduce task. We have analyzed Yahoo!'s Hadoop version 0.19. Note that Hadoop uses a *hard-coded* execution pipeline. No operator-model is used. However Hadoop's query execution strategy may be expressed as a physical operator DAG. To our knowledge, this thesis is the first to do so in that detail and we term it *The Hadoop Plan*. Based on this we then reason on The Hadoop Plan.

3.3.1 The Hadoop Plan

The Hadoop Plan is shaped by three user-defined parameters M, R, and P setting the number of mappers, reducers, and data nodes, respectively [45]. An example for a plan with four mappers ($M = 4$), two reducers ($R = 2$), and four data nodes ($P = 4$) is shown in Figure 3.2. We observe, that The Hadoop Plan consists of a subplan L ( ) and P subplans H1–H4 ( ) which correspond to the inital *load phase (HDFS)* into Hadoop's distributed file system. M determines the number of mapper subplans ( ), whereas R determines the number of reducer subplans ( ).

Let's analyze The Hadoop Plan in more detail:

Data Load Phase. To be able to run a MapReduce job, we first load the data into the distributed file system. This is done by partitioning the input T horizontally into disjoint subsets $T_1, \ldots, T_b$. See the physical partitioning operator `PPart` in subplan L. In the example $b = 6$, i.e. we obtain subsets $T_1, \ldots, T_6$. These subsets are called *blocks*. The partitioning function `block` partitions the input T based on the block size. Each block is then replicated (`Replicate`). The default number of replicas used by Hadoop is 3, but this may be configured. For presentation reasons, in the example we replicate each block only once. The figure shows 4 different data nodes with subplans H1–H4. Replicas are stored on different nodes in the network (`Fetch` and `Store`). Hadoop tries to store replicas of the same block on different nodes.

Map Phase. In the map phase each map subplan M1–M4 reads a subset of the data called a *split*[3] from HDFS. A *split* is a logical concept typically comprising one or more blocks. This assignment is defined by UDF `split`. In the example, the split assigned to M1 consists of two blocks which may both be retrieved from subplan H1. Subplan M1 unions the input blocks T_1 and T_5 and breaks them into records (`RecRead`). The latter operator uses a UDF `itemize` that defines how a split is divided into items. Then subplan M1 calls `map` on each item and passes the output to a `PPart` operator. This operator divides the output into so-called *spills* based on a partitioning UDF `mem`. By default `mem` creates spills of size 80% of the available main memory. Each spill is logically partitioned (`LPart`) into different regions containing data belonging to different reducers. For each tuple a shuffle UDF `sh` determines its reducer[4]. We use $\twoheadrightarrow$ to visualize the logically partitioned stream. In the example — as we have only two reducers — the stream is partitioned into two substreams only. Each logical partition is then sorted (`Sort`) respecting the sort order defined by UDF `cmp`. After that the data is grouped

[3] Not to be confused with *spills*. See below. We use the terminology introduced in [45].

[4] By default MapReduce (and also Hadoop) implement UDF `sh` using a hash partitioning on the intermediate key [45].

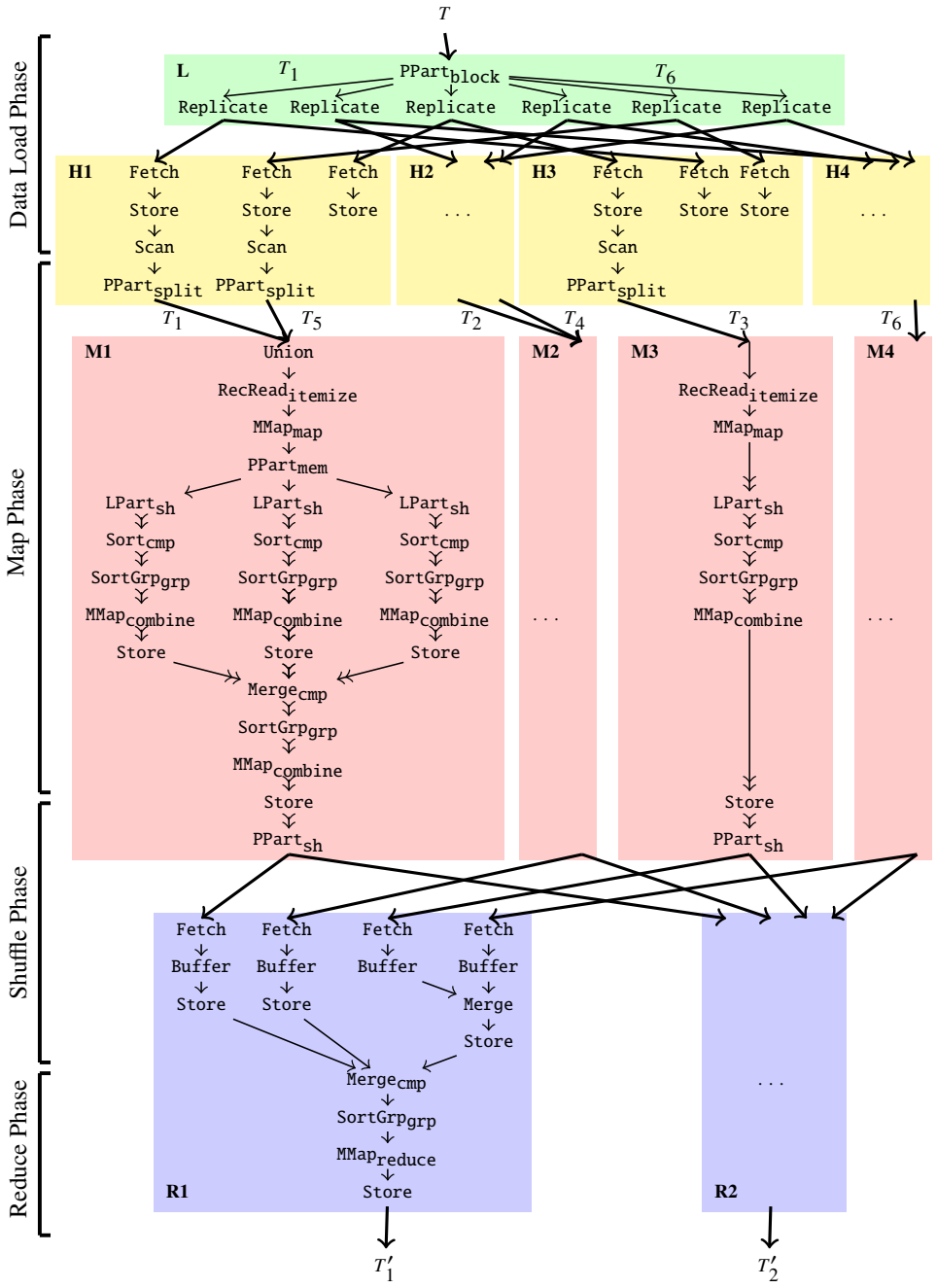


FIGURE 3.2: The Hadoop Plan: Hadoop's processing pipeline expressed as a physical query execution plan

(`SortGrp`) building groups as defined by UDF `grp`. For each group `MMap`[5] calls UDF `combine` which pre-reduces the data [45]. The output is materialized on disk (`Store`). M1 shows a subplan processing three spill files. These spill files are then retrieved from disk and merged (`Merge`). Again, we apply `SortGrp`, `MMap` and `combine`. The result is stored back on disk (`Store`). Subplan M3 shows a variant where only a single spill file that fits into main memory is created. When compared to M1, M3 looks different. This asymmetry may occur if a mapper subplan (here: M3) consumes less input data and/or

[5]A multimap operator `MMap` maps one input item to zero, one, or many output item(s). See Section 3.2.6 for details.

creates less output data than other subplans (here: M1). In this case all intermediate data may be kept in main memory in that subplan. In any case all output data will be completely materialized on local disk (`Store`).

Shuffle Phase. The shuffle phase redistributes data using a partitioning UDF `sh`. This is done as follows: each reducer subplan (R1 and R2 in the example) fetches the data from the mapper subplans, i.e. each reduce subplan has a `Fetch` operator for each mapper subplan. Hence, in this example we have $2 \times 4 = 8$ `Fetch` operators (see for instance R1). For each mapper subplan there is a `PPart` operator with R outgoing arrows $\rightarrow$. This means, the streams do not represent logical partitions anymore but are physically partitioned (see for instance M1). The reducer subplans retrieve the input files entirely from the mapper subplans and try to store them in main memory in a `Buffer` before continuing the plan. Note that the retrieval of the input to the reduce phase is *entirely blocking*. If the input data does not fit into main memory, those files will be stored on disk in the reducer subplans. For instance, in R1, the input data from M1 and M2 is buffered on disk, whereas the input from M3 and M4 is directly merged (`Merge`) and then stored. After that the input from M1 and M2 and the merged input from M3 and M4 is read from disk and merged. Note that if the input to a reducer is already locally available at the reducer node, `Fetch` may be skipped. This may only happen if the previous mapper subplan was executed on the same node. Also notice that `PPart` uses the same shuffle UDF `sh` as used inside a mapper subplan.

Reduce Phase. Only after a single output stream can be produced, the actual reduce phase starts. The result of the `Merge` is grouped (`SortGrp`) and for each group `MMap` calls `reduce`. Finally, the result is stored on disk (`Store`). The MapReduce framework does not provide a single result output file but keeps one output file per reducer. Thus the result of MapReduce is the union of those files. Notice that all UDFs are optional except `map`. In case `reduce` was not specified, the reduce and shuffle phases may be skipped.

3.3.2 Discussion

(1.) In general, by using a hard-coded, operator-free, query-execution pipeline, Hadoop makes it impossible to use other more efficient plans (possibly computed depending on current workload, data distribution, etc.)

(2.) At the mapper side, a full-table scan is used as the only access method on the input data. No index access is provided.

(3.) Grouping is implemented by sorting.

(4.) Several `MMap` operators executing `combine()` functions (which usually perform the same as a `reduce()` function [45]) are inserted into the merge tree. This is an implementation of *early duplicate removal and aggregation* [16, 131]. For merges with less than three input spills no early aggregation is performed.

(5.) The Hadoop Plan is highly customizable by exchanging one of the ten UDFs `block`, `split`, `itemize`, `mem`, `map`, `sh`, `cmp`, `grp`, `combine`, and `reduce`.

In summary, one could consider The Hadoop Plan a distributed external merge sort where the run (=spill) generation and first level merge is executed in the mapper subplan. Higher level and final merges are executed in the reducer subplans. The sort operation is mainly performed to be able to do a sort-based grouping — but this *interesting order* may also be exploited for applications bulkloading indexes (e.g. inverted

lists or B^+-trees). The initial horizontal partitioning into disjoint, equally-sized subsets resembles the strategy followed by shared-nothing DBMSs: in a first phase, the different subsets can be processed fully independently. In a second phase, intermediate results are horizontally repartitioned among the different reducers and then merged into the final result sets.

3.4 Trojan Index

The Hadoop Plan as shown in Figure 3.2 uses a `Scan` operator to read data from disk. Currently, Hadoop does not provide index access due to the lack of a priori knowledge of schema and the MapReduce jobs being executed. In contrast, DBMSs require users to specify the schema; indexes may then be added on demand. However, if we know the schema and the anticipated MapReduce jobs, we may create appropriate indexes in Hadoop as well.

Trojan Index is our solution to integrate indexing capability into Hadoop. The salient features of our approach are as follows:

(1.) *No External Library or Engine*: Trojan Index integrates indexing capability natively into Hadoop without imposing a distributed SQL-query engine on top of it.
(2.) *Non-Invasive*: We do not change the existing Hadoop framework. Our index structure is implemented by providing the right UDFs.
(3.) *Optional Access Path*: Trojan Index provides an optional index access path which can be used for selective MapReduce jobs. The scan access path can still be used for other MapReduce jobs.
(4.) *Seamless Splitting*: Data indexing adds an index overhead ($\sim$ 8MB for 1GB of indexed data) for each data split. The new logical split includes the data as well as the index. Our approach takes care of automatically splitting indexed data at *logical* split boundaries. Still data and indexes may be kept in different physical objects, e.g. if the index is not required for a particular task.
(5.) *Partial Index*: Trojan Index need not be built on the entire split; it can be built on any contiguous subset of the split as well. This is helpful when indexing one out of several relations, co-grouped in the same split.
(6.) *Multiple Indexes*: Several Trojan Indexes can be built on the same split. However, only one of them can be the primary index. During query processing, an appropriate index can be chosen for data access.

We illustrate the core idea of Trojan Index in Figure 3.3. For each split of data (SData T) a covering index (Trojan Index) is built. Additionally, a header (H) is added. It contains indexed data size, index size, first key, last key and number of records. Finally, a split footer (F) is used to identify the split boundary. A user can configure the split size (SData T) while loading the data. We discuss the Trojan Index creation and subsequent query processing below.

3.4.1 Index Creation

Trojan Index is a covering index consisting of a sparse directory over the sorted split data. This directory is represented using a cache-conscious CSS-tree [108] with the leaf

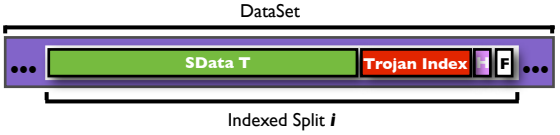


FIGURE 3.3: Indexed Data Layout

pointers pointing to pages inside the split. In MapReduce we can express our index creation operation for relation T over an attribute a_i as follows:

Index.

$$Index_{a_i}(T) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ [(\texttt{getSplitID()} \oplus \texttt{prj}_{a_i}(k \oplus v), k \oplus v)] \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\ [(ivs \oplus \texttt{indexBuilder}_{a_i}(ivs))] \end{cases}$$

Here, $\texttt{prj}_{a_i}$ denotes a projection to attribute a_i and $\oplus$ denotes that two attribute sets are concatenated to a new schema. Figure 3.4(a) shows the MapReduce plan corresponding to the indexing operation defined above. The distributed file system stores the data for Relation T. The MapReduce client partitions the Relation T into splits as shown in the figure. The `itemize`.next() function reads {`offset,record`}-pairs and the `map` emits {`splitID+a,record`} as the intermediate key-value pair. Here `splitID+a` is a composite key concatenating the split ID (function `getSplitID()`) and the index attribute; `record` is a value containing all attributes of the record.

We need to re-partition the composite keys emitted from the mappers such that the reducers receive almost the same amount of data. We do this by supplying a hash partitioning function (UDF `sh` in The Hadoop Plan) that re-partitions records by hashing only on the split identifier portion of the composite key.

$$\texttt{sh}(\text{key } k, \text{value } v, \text{int } numPartitions) \mapsto k.splitID \ \% \ numPartitions \tag{3.1}$$

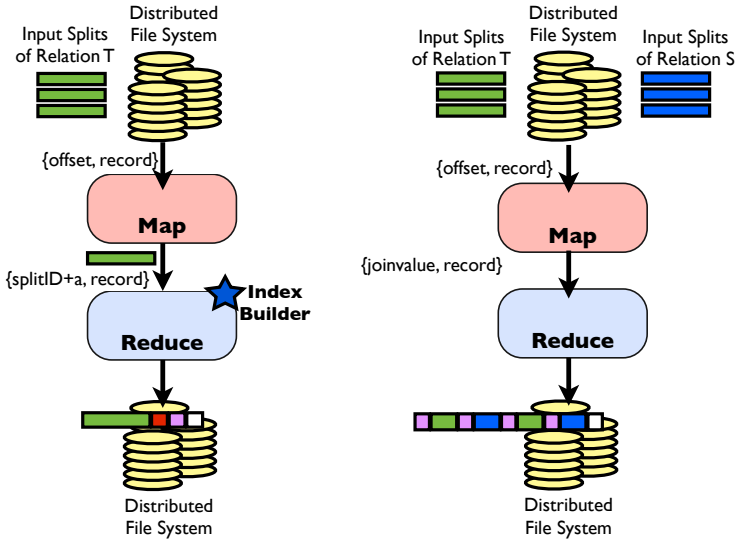
To construct a clustered Trojan Index, the data needs to be sorted on the index attribute a. For this we exploit the interesting orders created by the MapReduce framework [45]. This is faster than performing a local sort at the reducers. To do so, we provide a UDF `cmp` instructing MapReduce to sort records by considering only the second part (the index attribute a) of the composite key.

$$\texttt{cmp}(\text{key } k1, \text{key } k2) \mapsto \texttt{compare}(k1.a \ , \ k2.a)$$

Since we are building Trojan Index per split, we need to preserve the split in each reducer call. For this we provide a grouping function (UDF `grp`) that groups tuples based on the split identifier portion of the composite key.

$$\texttt{grp}(\text{key } k1, \text{key } k2) \mapsto \texttt{compare}(k1.splitID \ , \ k2.splitID) \tag{3.2}$$

`reduce`, shown in Figure 3.4(a), has a local `indexBuilder` function. which builds the Trojan Index on the index attribute of the sorted data. `reduce` emits the set of values concatenated with the Trojan Index, index header, and split footer. The output data is stored on the distributed file system.



(a) Indexing (b) Co-partitioning

FIGURE 3.4: MapReduce Plans

Algorithm 3.1: Trojan Index/Trojan Join `split` UDF

Input : JobConf job, Int numSplits
Output: logical data splits

```
1  FileSplit [] splits;
2  File [] files = GetFiles(job);
3  foreach file in files do
4      Path path = file.getPath();
5      InputStream in = GetInputStream(path);
6      Long offset = file.getLength();
7      while offset > 0 do
8          in.seek(offset-FOOTER_SIZE);
9          Footer footer = ReadFooter(in);
10         Long splitSize = footer.getSplitSize();
11         offset -= (splitSize + FOOTER_SIZE);
12         BlockLocations blocks = GetBlockLocations(path,offset);
13         FileSplit newSplit = CreateSplit(path,offset,splitSize,blocks);
14         splits.add(newSplit);
15     end
16 end
17 return splits;
```

3.4.2 Query Processing

Consider a query q referencing an indexed dataset T. We identify the split boundaries using *footer* F and create a map task for each split. Algorithm 3.1 shows the `split` UDF that we provide for creating the splits. For a given job, we retrieve and iterate over all data files (Lines 2–3). For each file we retrieve its path and the input stream (Lines 4–5). The input stream is used to seek and read the split footers, i.e. we do not scan the entire data here. We start looking for footers from the end (Lines 6–8) and retrieve the split size from them (Lines 9–10). We set the offset to the beginning of the split (Line 11) and use it to retrieve block locations (Line 12) and to create a logical split (Line 13). We add the newly created split to the list of logical splits (Line 14) and repeat the process until all footers in all files have been read. Finally, we return the list of logical splits (Line 17).

Algorithm 3.2: Trojan Index `itemize`.initialize UDF

Input: FileSplit split, JobConf job

```
1  Global FileSplit split = split;
2  Key lowKey = job.getLowKey();
3  Global Key highKey = job.getHighKey();
4  Int splitStart = split.getStart();
5  Global Int splitEnd = split.getEnd();
6  Header h = ReadHeader(split);
7  Overlap type = h.getOverlapType(lowKey,highKey);
8  Global Int offset;
9  if type == LEFT_CONTAINED or type == FULL_CONTAINED or type == POINT_CONTAINED then
10 |  Index i = ReadIndex(split);
11 |  offset = splitStart + i.lookup(lowKey);
12 else if type == RIGHT_CONTAINED or type == SPAN then
13 |  offset = splitStart;
14 else
15 |  // NOT_CONTAINED, skip the split;
16 |  offset = splitEnd;
17 end
18 Seek(offset);
```

Algorithm 3.3: Trojan Index `itemize`.next UDF

Input : KeyType key, ValueType value
Output: has more records

```
1  if offset < splitEnd then
2  |  Record nextRecord = ReadNextRecord(split);
3  |  offset += nextRecord.size();
4  |  if nextRecord.key < highKey then
5  |  |  SetKeyValue(key, value, nextRecord);
6  |  |  return true;
7  |  end
8  end
9  return false;
```

Algorithm 3.2 shows the `itemize` UDF that we provide for index scan. We read the low and the high selection keys (Lines 1–2) from the job configuration and the split boundary offsets (Lines 3–4) from the split configuration. Thereafter, we first read the index header (Line 5) and evaluate the overlap type (Line 6) i.e. the portion of the split data relevant to the query. Only if the split contains the low key (Line 8), we read the index (Line 9) and compute the low key offset within the split (Line 10). Otherwise, if the split contains the high key or the selection range spans the split (Line 11), we set the offset to the beginning of the split (Line 12); else we skip the split entirely (Lines 13–15). Finally, we seek the offset within the split (Line 17) to start reading data record by record. Algorithm 3.3 shows the method to get the next record from the data split. We check if the split offset is within the end of split (Line 1) and index key value of the next record is less than the high key (Line 3). If yes, we set the key and the value to be fed to the mapper and return true (Lines 4-5), indicating there could be more records. Else, we return false (Line 8).

Note that the use of the Trojan Index is optional and depends upon the query predicate. Thus, both full and index scan are possible over the same data. In addition, indexes and data may be kept in separate physical blocks, i.e. UDF `split` may compose physical blocks into logical splits suited for a particular task.

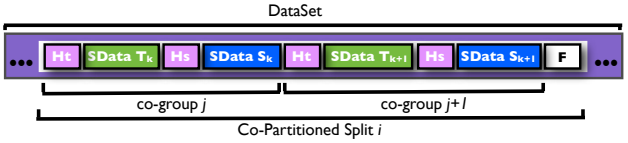


FIGURE 3.5: Co-partitioned Data Layout

3.5 Trojan Join

Efficient join processing is one of the most important features of DBMSs. In MapReduce, two datasets are usually joined using *re-partitioning*: partitioning records by join key in the map phase and grouping records with the same key in the reduce phase. The reducer joins the records in each key-based group. This re-partitioned join corresponds to the join detailed in Section 3.2.3. Yang et al. [34] proposed to extend MapReduce by a third *Merge* phase. The Merge phase is a join operator which follows the reduce phase and gets sorted results from it. Afrate and Ullman [6] proposed techniques to perform multiway joins in a single MapReduce job. However, all of the above approaches perform the join operation in the reduce phase and hence transfer a large amount of data trough the network — which is a potential bottleneck. Moreover, these approaches do not exploit any schema-knowledge, which is often available in advance for many relational-style tasks. Furthermore, join conditions in a schema are very unlikely to change — the set of tables requested in a join query may however change.

Trojan Join is our solution to support more effective join processing in Hadoop. We assume that we know the schema and the expected workload, similar to DBMS and HadoopDB. The core idea is to *co-partition* the data at load time — i.e. given two input relations, we apply the same partitioning function on the join attributes of both the relations at data loading time — and place the *co-group* pairs, having the same join key from the two relations, on the same split and hence on the same node. As a result, joins are now processed locally within each node at query time — a feature that is also explored by SQL-DBMSs. Moreover, we are free to group the data on any attribute other than the join attribute in the same MapReduce job. The salient features of Trojan Join are as follows:

(1.) *Non-Invasive.* We do not change the existing Hadoop framework. We only change the internal representation of a data split.

(2.) *Seamless Splitting.* When co-grouping the data, we create three headers per data split: two for indicating the boundaries of data belonging to different relations; one for indicating the boundaries of the logical split. Trojan Join automatically splits data at logical split boundaries that are opaque to the user.

(3.) *Mapper-side Co-partitioned Join.* Trojan Join allows users to join relations in the map phase itself exploiting *co-partitioned data*. This avoids the shuffle phase, which is typically quite costly from the network traffic perspective.

(4.) *Trojan Index Compatibility.* Trojan indexes may freely be combined with Trojan Joins. We detail this aspect in Section 3.5.3.

We illustrate the data layout for Trojan Join in Figure 3.5. Each split is separated by split footer (F) and contains data from two relations T S (depicted green and blue in Figure 3.5). We use two headers H_t and H_s, one for each relation, to indicate the size of

each co-partition[6]. Given an equi-join predicate $PJ(T, S) = (T.a_i = S.b_j)$, the Trojan Join proceeds in two phases: the *data co-partitioning* and *query processing* phases.

3.5.1 Data Co-Partitioning

Trojan Join co-partitions two relations in order to perform join queries using map tasks only. Formally, we can express co-partitioning as:

$$CoPartition_{a_i,b_j}(T,S) \Rightarrow \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \\ \begin{cases} [(\texttt{prj}_{a_i}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(\texttt{prj}_{b_j}(k \oplus v), k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(\{ik\} \times ivs)] \end{cases}$$

Here, the helper input() function identifies whether an input record belongs to T or S. Figure 3.4(b) shows the MapReduce plan for co-partitioning the data. This works as follows. The MapReduce client partitions the data of both relations into splits as shown in the figure. For each record in an input split, `itemize`.next() receives the `offset` as key and the record as value and `map` emits {`joinvalue, record`} as key-value pairs. Here `joinvalue` is the key having value either a_i or b_j depending on the lineage; `record` contains all attributes of the record. For re-partitioning, sorting and grouping the key-value pairs we use the entire key i.e. we use the default `sh`, `cmp`, and `grp` UDFs. As a result, each call to `reduce` receives the set of records having the same join attribute value. The final output of `reduce` is a virtual split containing several co-groups as shown in Figure 4.

3.5.2 Query Processing

A Trojan Join between relations T and S can be expressed as the re-partitioned join operator shown in Section 3.2.3 replacing `map` with an identity function. Though join processing in this manner is a considerable improvement, we still need to shuffle the data. However, we actually do not need the shuffle phase as relations T and S were already co-partitioned. Therefore, we present an optimized variant of this join which requires only a single `map` without a `reduce`. Hence, Hadoop++ may skip both the

[6]Notice that one can also store each relation in separate physical blocks just like a DBMS. Extending our approach to this is straightforward: we simply need to provide a UDF `split`. This also holds for our Trojan Index proposal.

Algorithm 3.4: Trojan Join `itemize`.next UDF

Input : KeyType key, ValueType value
Output: has more records

```
1  if offset < splitEnd then
2      if offset == nextHeaderOffset then
3          Header header = ReadHeader(split);
4          offset += header.size();
5          nextHeaderOffset = offset + header.getSplitSize();
6      end
7      Record nextRecord = ReadNextRecord(split);
8      offset += nextRecord.size();
9      SetKeyValue(key, value, nextRecord);
10     return true;
11 end
12 return false;
```

shuffle and the reduce phase. The map function in Trojan Join is shown below:

$$
T \bowtie_{PJ(T,S)} S \Rightarrow \left\{
\begin{array}{l}
set\ \beta = \emptyset; key\ lk = null; \\
\texttt{map}(\text{key } k, \text{value } v) \mapsto \\
\left\{
\begin{array}{l}
\text{if } (lk \mathrel{!}= k)\ \{ \\
\quad \text{if } !first\_incoming\_pair(k, v)\ \{ \\
\quad\quad output : \Big[\texttt{crossproduct}(T'_{lk}, S'_{lk}) \mid \\
\quad\quad\ T'_{lk} = \{(\texttt{prj}_{a_i}(k \oplus v), k \oplus v) \mid (k \oplus v) \in \beta \wedge \text{if input}(k \oplus v) = T\}, \\
\quad\quad\ S'_{lk} = \{(\texttt{prj}_{b_j}(k \oplus v), k \oplus v) \mid (k \oplus v) \in \beta \wedge \text{if input}(k \oplus v) = S\}\Big] \\
\quad \} \\
\quad \beta = \{k \oplus v\},\ \ lk = k \\
\} \\
\text{else } \{ \\
\quad output : \text{none} \\
\quad \beta = \beta \cup \{k \oplus v\} \\
\}
\end{array}
\right.
\end{array}
\right.
$$

To process a join query, our approach automatically splits the required data by identifying the split boundaries — using the *footer* F — and creates a map task for each split. For this, we supply a `split` UDF that identifies such boundaries (see Algorithm 3.1). We also supply a UDF `itemize` that allows mappers to skip headers in input splits. Algorithm 3.4 shows how UDF `itemize` computes the next key-value pairs ('items'). Here `offset`, `splitEnd`, and `header` are global variables defined in the `itemize`.initialize function (similar to Algorithm 3.2). We check if the split `offset` is contained in this split (Line 1). If yes, we check if the current `offset` points to a header (Line 2) so as to skip the header (Lines 3–5). We then set the key and the value to be fed to `map` and return true (Lines 7–10), indicating there could be more records. In case the `offset` is not within the end of split, we return false (Line 12). This indicates that there are no more records.

The `map` function shown before starts by initializing a co-group with the first (k, v)-pair. Thereafter, it keeps collecting in β the records belonging to the same co-group i.e. the same join attribute values. A different join attribute value indicates the beginning of the next co-group in the data. Here, we make two assumptions: first, records with the same join attribute value arrive contiguously, which is realistic since the relations are co-grouped; second, in contrast to previous MapReduce jobs, the map function maintains a state (β, lk) to identify the co-group boundaries within a split. When a new co-group starts, the map function classifies the records in β into relations T' and S' based on their lineage and performs the cross product between them by calling the local `crossproduct`

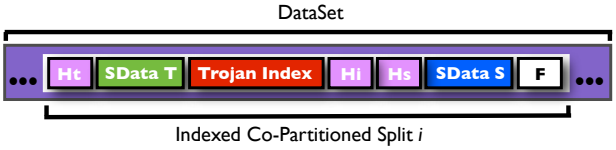


FIGURE 3.6: Indexed Co-partitioned Data Layout

function. The result is emitted and β is reset to start collecting records for the next co-group. This process is repeated until there is no more incoming (k, v)-pair. To perform the cross product on the last co-group, the `map` injects an `end-of-split record` after the last record in each data split marking the end of that split. The `reduce` may then output the join result over the last co-group. Notice that the final result of all of these co-partitioned joins is exactly the same as the result produced by the re-partitioned join.

3.5.3 Trojan Index over Co-Partitioned Data

We can also build indexes on co-partitioned data. Trojan Join may be combined with both unclustered and clustered Trojan Indexes. For instance, we can build an unclustered Trojan Index over any attribute without changing the co-grouped data layout. Alternatively, we can build a clustered Trojan Index by internally sorting the co-partitioned data based on the index attribute. The internal sorting process is required only when the index attribute is different from the join attribute. For example, assume relations T and S are co-partitioned and suppose we want to build a clustered Trojan Index over a given attribute of relation T. To achieve this, we run the indexing MapReduce job as described in Section 3.4.1. This job sorts the records from T based on the index attribute and stores them contiguously within the split. The resulting data layout is illustrated in Figure 3.6. Each split is separated by a split footer (F) and has a header per relation (H_t and H_s), indicating the size of each co-partition. In addition, a clustered Trojan Index and its header (H_i) is stored after the indexed relation (T) in the split. At query time, we supply the UDF `itemize` function as before. However, we set the constructor of `itemize` function as in Algorithm 3.3 in order to provide index scan. Adapting Trojan Join processing for indexed data is straightforward.

3.6 Experiments

We evaluate the performance of Hadoop++ (i.e. Hadoop including Trojan Index and Trojan Join) and compare it with Hadoop and HadoopDB. Our main goal in the experiments is to show that we can reach similar or better performance than Hadoop and HadoopDB without relying on local DBMSs. We also show in §3.6.4 that Hadoop++ still inherits Hadoop's fault-tolerance performance.

3.6.1 System Setup

Hadoop. For our experiments, we realized the following changes to the default configuration settings: (1) we stored data into the *Hadoop Distributed File Systems* (HDFS) using 256MB data blocks, (2) we allowed each task tracker to run with a maximum heap

size of 1024MB, (3) we increased the sort buffer to 200MB, (4) Hadoop was allowed to reuse the task JVM executor instead of restarting a new process per task, (5) we used 100 concurrent threads for merging intermediate results, (6) we allowed a node to concurrently run two map tasks and a single reduce task, and (7) we set HDFS replication to 1 as done in [5].

Hadoop++ is an improved version of Hadoop that incorporates support for index-scans and co-partitioning as discussed in §3.4 and §3.5. We use the same configuration settings as for Hadoop, but we allow it to read data in binary format, except for one of our benchmarks (*UDF task*). A binary record is composed of the data itself and a header containing the lineage of the record and the offset of each attribute.

HadoopDB is a hybrid system combining Hadoop, HBase, and single instance DBMSs, e.g. Postgres, into a system somewhat similar to a PDBMS. We use PostgreSQL 8.4 as local database and increase the memory for shared buffers to 512 MB and the working memory to 1 GB. As in [5], we do not use PostgreSQL's data compression feature, we set data replication to 1 as done in [5].

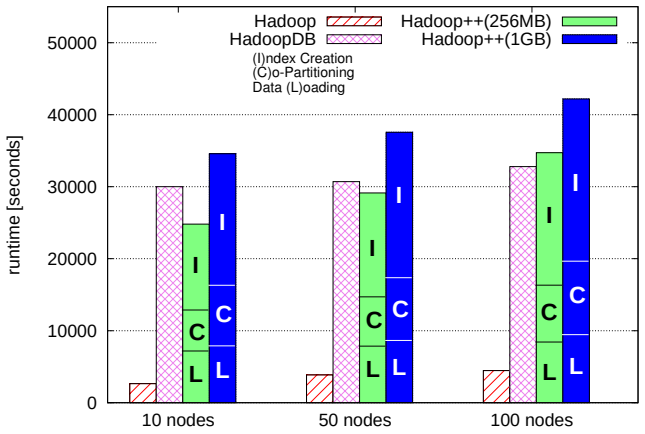
3.6.2 Benchmark Setup

We ran all our experiments on Amazon EC2 large instances in *US-east* location. Each large instance has 4 EC2 compute units (2 virtual cores), 7.5 GB of main memory, 850 GB of disk storage and runs 64-bit platform Linux Fedora 8 OS. Throughout our performance study we realized that performance on EC2 may vary. We analyse this variance in detail in an accompanying paper [110]. Here we executed each of the tasks three times and report the average of the trials. We discard these assumptions to evaluate fault-tolerance in §3.6.4. We report only those trial results where all nodes are available and operating correctly. To factor out variance, we also ran the benchmark on a physical 10-node cluster where we obtained comparable results[7]. On EC2 we scale the number of virtual nodes: 10, 50, and 100. We compared the performance of Hadoop++ against Hadoop and HadoopDB. We used Hadoop 0.19.1 running on Java 1.6 for all these three systems. We evaluated two variants of Hadoop++ that only differ in the size of the input splits (256 MB and 1 GB[8]). For HadoopDB, we created databases exactly as in [5].

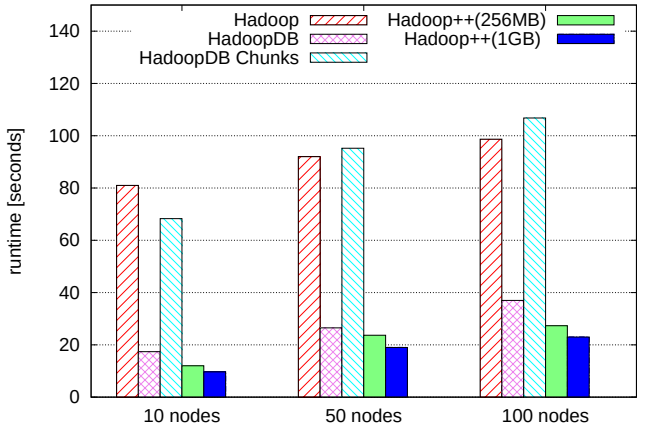
We used the benchmark and data generator proposed in [101] and used in the HadoopDB paper [5]. We selected those tasks relevant to indexing and join processing. For completeness, we also report results of the other tasks in Section 3.6.5. The benchmark creates three tables: (1) *Documents* containing HTML documents, each of them having links to other pages following a Zipfian distribution. (2) *Rankings* containing references to *Documents*, (3) *UserVisits* referencing *Rankings*. Both *Rankings* and *UserVisits* contain several randomly generated attribute values. The sizes of *Rankings* and *UserVisits* are 1 GB (18M tuples) and 20 GB (155M tuples) per node, respectively. Please refer to [101] for details.

[7] With a single exception: on the physical cluster for the selection task Hadoop++(1GB) was still faster than HadoopDB, but Hadoop++(256MB) was slightly slower than HadoopDB.

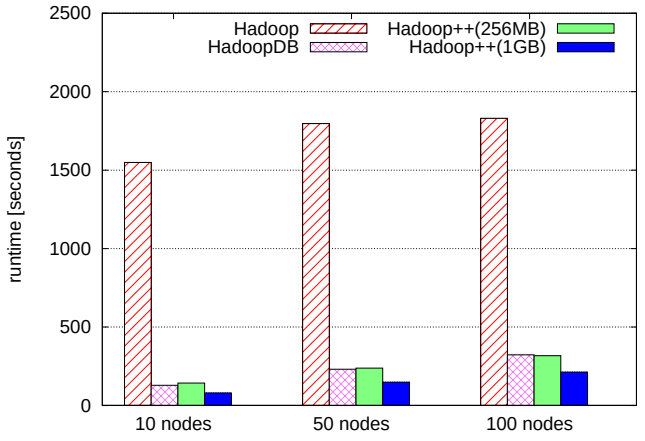
[8] Unfortunately, we could not use split sizes beyond 1GB due to a bug [67] in Hadoop's distributed file system. We believe however that runtimes using Trojan Join would improve even further.



(a) Data loading, partitioning, and indexing



(b) Selection Task



(c) Join Task

FIGURE 3.7: Benchmark Results related to Indexing and Join Processing

3.6.3 Analytical Tasks

3.6.3.1 Data Loading

As in [5] we show the times for loading UserVisits only; the time to load the small Rankings is negligible. Hadoop just copies *UserVisits* (20GB per node) from local hard disks into HDFS, while Hadoop++ and HadoopDB partition it by *destinationURL* and index it on *visitDate*. Figure 3.7(a) shows the load times for *UserVisits*. For Hadoop++ we show the different loading phases: The data loading into HDFS including conversion from textual to binary representation, followed by the co-partioning phase (§ 3.5.1), and index creation (§ 3.4.1). We observe that Hadoop++(256MB) has similar performance as HadoopDB; Hadoop++(1GB), however, is slightly slower. We believe this is because the loading process is CPU-bound, thereby causing map tasks to slow down when processing large input splits. However, this difference is negligible, as these costs happen at data load time. This means these costs have to be paid only once. Users may then run an unlimited number of tasks against the data. The **trade-off** we observe is similar to the one seen in any DBMS: the more we invest at data load time, the more we might gain at query time. Thus, the more queries benefit from that initial investment, the higher the overall gain. Overall, we conclude that Hadoop++ scales well with the number of nodes.

3.6.3.2 Selection Task

This task performs a selection predicate on *pageRank* in *Rankings*. We use the same selectivity as in [5, 101], i.e. 36,000 tuples per node by setting the *pageRank* threshold to 10. We describe the `SQL` queries and MapReduce jobs used for the selection task below.

SQL query. HadoopDB performs the selection task by executing this `SQL` statement:

```
SELECT pageURL, pageRank FROM Rankings WHERE pageRank>10;
```

MapReduce jobs. Hadoop performs the same MapReduce job as in [5]. In contrast to Hadoop, Hadoop++ uses a MapReduce job composed of a single `map` function receiving only those $(key, value)$-pairs whose *pageRank* is above 10. This is because Hadoop++ makes use of the Trojan Index.

For this task, we run two variants of HadoopDB similar to the authors of HadoopDB [5]. In the first variant, each node contains the entire 1 GB *Rankings* in a single local database. In the second variant each node contains twenty partitions of 50 MB each in separate local databases (HadoopDB Chunks). Figure 3.7(b) illustrates the selection task results for all systems. We observe that Hadoop++ outperforms Hadoop and HadoopDB Chunks by up to factor 7, and HadoopDB by up to factor 1.5. We also observe that Hadoop++(1GB) performs better than Hadoop++(256MB). This is because Hadoop++(1GB) has much fewer map tasks to execute and hence less scheduling overhead. Furthermore, its index coverage is greater. This allows it to get more data at once. These results demonstrate the superiority of Hadoop++ over the other systems for selection tasks.

3.6.3.3 Join Task

This task computes the average *pageRank* of those pages visited by the *sourceIP* address that has generated the most *adRevenue* during the week of January 15-22, 2000. This task requires each system to read two different data sets (*Rankings* and *UserVisits*) and join them. The number of records in *UserVisits* that satisfy the selection predicate is ∼134,000. We describe the `SQL` queries and MapReduce jobs used to perform the join tasks below.

SQL query. HadoopDB pushes the following `SQL` statement to the local databases. It computes partial aggregates in the local DBMSs and then requires a single `reduce` task for the final aggregation:

```
SELECT sourceIP, COUNT(pageRank),
                 SUM(pageRank), SUM(adRevenue)
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN
Date('2000-01-15') AND Date('2000-01-22')
GROUP BY UV.sourceIP;
```

MapReduce jobs. While Hadoop uses three MapReduce jobs as explained in [101], Hadoop++ uses a single MapReduce job that implements the co-partitioned join operator explained in §3.5. This works as follows. First, the selection predicate on *visitDate* is applied and only matching *UserVisits* records are passed to the `map` function. For *Rankings*, all records are passed to the `map` function. The `map` function, in turn, performs the join operation and outputs only those results that satisfy the join predicate. Notice that the `map` function can perform the join operation locally because data in input splits is composed of co-groups from *Rankings* and *UserVisits*. Then, a `combine` performs pre-aggregation before shuffling. Finally, a single `reduce` task performs the final aggregation.

Figure 3.7(c) illustrates results for each system when performing this join task. Again, we observe that Hadoop++ outperforms Hadoop by up to factor 20. This is because Hadoop++ performs an index-scan over *UserVisits* to speed up the selection predicate and because *Rankings* and *UserVisits* were co-grouped at loading time. More importantly, our results show that Hadoop++(1GB) outperforms HadoopDB by up to factor 1.6. This is not the case for Hadoop++(256MB), because it has less relevant data per input split to join and more `map` tasks to process. Again, as discussed in §3.6.3.1, these gains are possible, as we trade query performance with additional effort at data load time, see Figure 3.7(a).

3.6.4 Fault-Tolerance

In this section we show results of two fault-tolerance experiment which are similar to the one done in [5]. We perform the *node failures* experiment as follows: we set the expiry interval, i.e. the maximum time between two heartbeats, to 60 seconds. We chose a node randomly and kill it after 50% percent of work progress. We perform the *straggler nodes* experiment as follows: we run a concurrent I/O-intensive process on a randomly chosen node so as to make it a straggler node. We define the slowdown as in [5], $slowdown = \frac{(n-f)}{n} * 100$, where n is the query execution time without failures

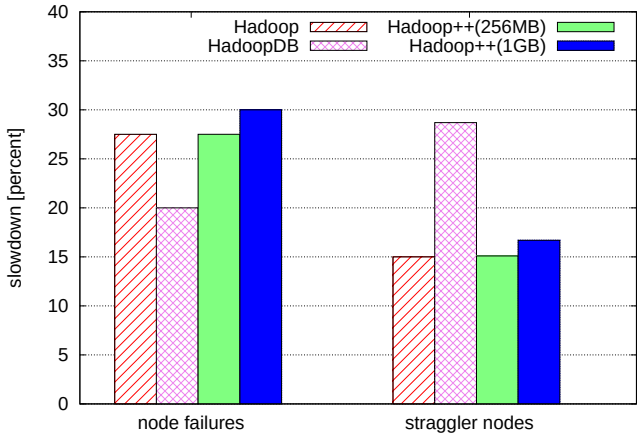


FIGURE 3.8: Fault Tolerance.

and f is the execution time with a node failure. For both series of tests, we set HDFS replication to 2.

Figure 3.8 shows the results. As expected, we observe that Hadoop++(256MB) has the same performance as Hadoop. However, we can see that while increasing the size of input splits from 256 MB to 1 GB, Hadoop++ slows down. This is because Hadoop++(1GB) has 4 times more data to process per input split, and hence it takes more time to finish any lost task. Hence, we observe a natural trade-off between performance and fault-tolerance: By increasing the input split size, Hadoop++ has better performance but it is less fault-tolerant and vice-versa. We observe that Hadoop++ is slower than HadoopDB for the node failures experiments. This is because Hadoop++ needs to copy data from replica nodes while HadoopDB pushes work to replica nodes and thus requires less network traffic. For the straggler nodes experiment however, Hadoop++ significantly outperforms HadoopDB. This is because HadoopDB sometimes pushes tasks to straggler nodes rather than replica nodes. This slows down its speculative execution.

3.6.5 Additional Benchmark Results

Here we list results for the other tasks defined in the bechmark of [101]. For the grep task, Hadoop++ executes exactly the same code as Hadoop. Therefore runtimes are not effected (and not shown). For the other tasks — even though they are neither related to indexing nor join processing — we still see an improvement of Hadoop++ over Hadoop. We discuss this briefly in the following.

3.6.5.1 Large and Small Aggregation Task

These tasks computes the total sum of *adRevenue* grouped by *sourceIP* in *UserVisits*. *Large Aggregation Task* uses all characters of *sourceIP* as grouping key; it computes 2.5 millions groups. In contrast, *Small Aggregation Task* uses the first seven characters of *sourceIP* as grouping key; it computes $2,000$ groups. We describe the `SQL` queries and MapReduce jobs used to perform the aggregation tasks below.

SQL queries. The `SQL` queries used by HadoopDB to perform the large (`Q1`) and small (`Q2`) aggregation tasks are as follows:

```
(Q1) SELECT sourceIP, SUM(adRevenue)
     FROM UserVisits GROUP BY sourceIP;
(Q2) SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
     FROM UserVisits GROUP BY SUBSTR(sourceIP, 1, 7);
```

MapReduce jobs. For Hadoop and Hadoop++, we use a similar MapReduce job as introduced in [101], but without the final MapReduce job to produce a single file. The only difference between Hadoop's and Hadoop++'s plan is that Hadoop reads data in text format while Hadoop++ reads it in binary format.

Figures 3.9(a) and 3.9(b) summarize the results for this task. We observe that for both tasks (small and large and aggregation) Hadoop++ outperforms Hadoop because it reads data in binary format and, hence, it can read *sourceIP* and *adRevenue* without reading other attributes. Furthermore, we can observe that Hadoop++ is slower than HadoopDB, because Postgres applies a hash aggregation while Hadoop++ uses a sort-based-aggregation. However, the performance of Hadoop++ and HadoopDB is in the same ballpark even though HadoopDB emulates a non-compressed PDBMS and even though the improvements of Hadoop++ are not related to aggregate computation.

3.6.5.2 UDF Aggregation Task

We also consider an aggregation query that parses each HTML document in *Documents*, using a UDF[9], which extracts the inlinks and counts the number of unique pages referencing a URL.

When loading HTML documents into HDFS for Hadoop and Hadoop++, we proceed as in [5, 101], i.e. we concatenate several documents into larger ones in order to avoid memory problems with the HDFS' server when dealing with a large number of documents. In contrast, HadoopDB stores each HTML document separately in relation *Documents*. We describe the `SQL` queries and MapReduce jobs used to perform the UDF aggregation task below.

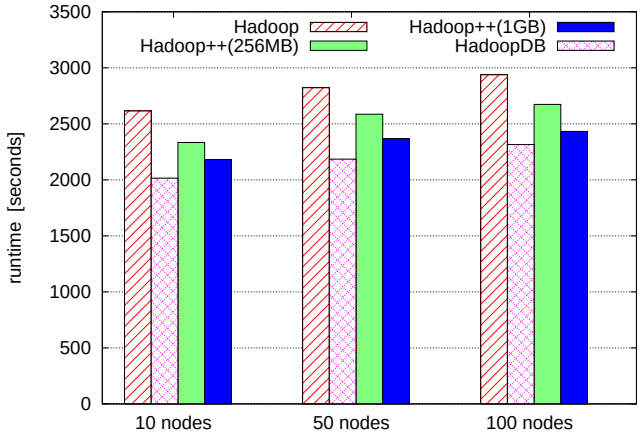
SQL queries. HadoopDB uses: `SELECT url, contents FROM Documents;`

Notice that this query allows HadoopDB to get the contents of HTML documents. Thereafter, it has to perform the UDF part using an identical MapReduce job as Hadoop.

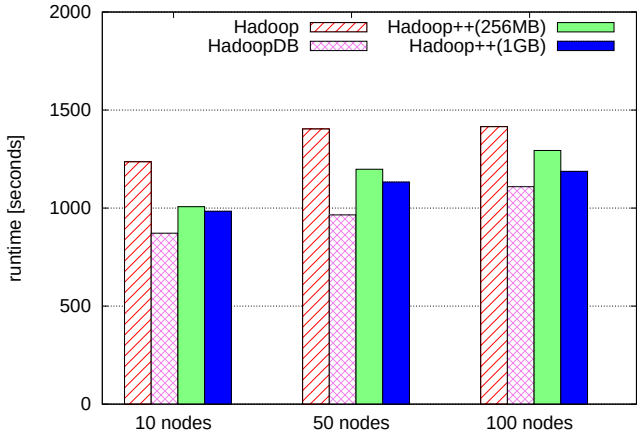
MapReduce jobs. Hadoop and Hadoop++ use a MapReduce job whose `map` function receives HTML documents split by their lines. For each incoming line, the `map` function uses a regular expression to find all URLs and outputs an integer 1 for each URL found. Finally, a `reduce` function aggregates all values having the same URL. As a result, this MapReduce job computes the number of references for each URL.

Figure 3.9(c) shows results for this task. We see the best variant of Hadoop++ is at least as good or better than HadoopDB. This is because Hadoop++ processes concatenated HTML documents as described above. Compared to Hadoop, Hadoop++(256MB) has similar performance. However, overall, neither HadoopDB nor Hadoop++ can improve over the Hadoop for this particular task.

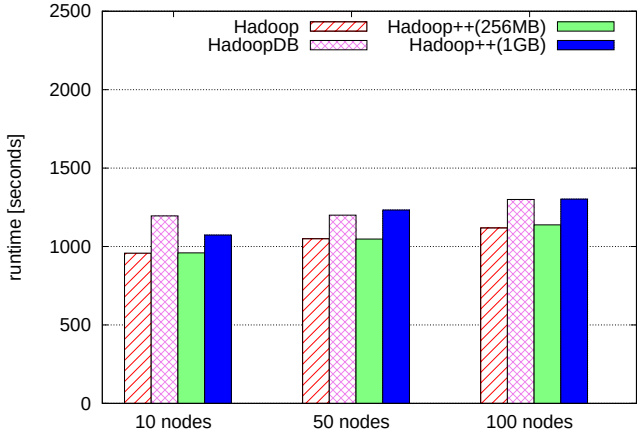
[9]Not to be confused with the ten UDFs provided by Hadoop as explained above.



(a) Large Aggregation Task



(b) Small Aggregation Task



(c) UDF Aggregation Task

FIGURE 3.9: Additional Task Results not related to Indexing and Join Processing

3.7 Discussion & Conclusion

This chapter has proposed new index and join techniques: Trojan Index and Trojan Join, to improve runtimes of MapReduce jobs. Our techniques are non-invasive, i.e. they do to require us to change the underlying Hadoop framework. We simply need to provide appropriate user-defined functions (and not only the two functions `map` and `reduce`). The beauty of this approach is that we can incorporate such techniques to any Hadoop version with no effort. We exploited this during our experiments when moving from Hadoop 0.20.1 to Hadoop 0.19.0 (used by HadoopDB) for fairness reasons. We implemented our Trojan techniques on top of Hadoop and named the resulting system Hadoop++.

The experimental results demonstrate that Hadoop++ outperforms Hadoop. Furthermore, for tasks related to indexing and join processing Hadoop++ outperforms HadoopDB – without requiring a DBMS or deep changes in Hadoop's execution framework or interface. We also observe that as we increase the split size, Hadoop++ further improves for both selection and join tasks. This is because the index coverage also increases. Performance of fault-tolerance, however, decreases with larger splits as it requires more time to recompute lost tasks. This symbolizes a tradeoff between runtime and fault tolerance of MapReduce jobs.

An important lesson learned from this chapter is that most of the performance benefits stem from exploiting schema knowledge on the dataset and anticipating the query workload at *data load time*. Only if this schema knowledge is available, DBMSs, HadoopDB as well as Hadoop++ may improve over Hadoop. But again: there is no need to use a DBMS for this. Schema knowledge and anticipated query workload may be exploited in *any* data processing system.

In terms of Hadoop++'s interface we believe that we do not have to change the programming interface to SQL: standard MapReduce jobs — unaware of possible indexes and join conditions — may be analyzed [22] and then rewritten to use the Trojan techniques proposed in this chapter.

With Trojan Index and Trojan Join, Hadoop++ introduces indexed and co-partitioned storage views, apart from the default row storage view, in Hadoop MapReduce. As a result of this greater flexibility in the storage layer, Hadoop++ allows for efficient index and join processing, in addition to standard scan-oriented processing.

Chapter 4

Data Layouts for Large Scale Data Management

MapReduce is becoming ubiquitous in large-scale data analysis. Several recent works have shown that the performance of Hadoop MapReduce could be improved, for instance, by creating indexes in a non-invasive manner. However, they ignore the impact of the data layout used inside data blocks of Hadoop Distributed File System (HDFS). In this chapter, we analyze different data layouts in detail in the context of MapReduce and argue that Row, Column, and PAX layouts can lead to poor system performance. We propose a new data layout, coined Trojan Layout, that internally organizes data blocks into attribute groups according to the workload in order to improve data access times. A salient feature of Trojan Layout is that it fully preserves the fault-tolerance properties of MapReduce. We implement our Trojan Layout idea in HDFS 0.20.3 and call the resulting system Trojan HDFS. We exploit the fact that HDFS stores multiple replicas of each data block on different computing nodes. Trojan HDFS automatically creates a different Trojan Layout per replica to better fit the workload. As a result, we are able to schedule incoming MapReduce jobs to data block replicas with the most suitable Trojan Layout. We evaluate our approach using three real-world workloads. We compare Trojan Layouts against Hadoop using Row and PAX layouts. The results demonstrate that Trojan Layout allows MapReduce jobs to read their input data up to 4.8 times faster than Row layout (3.3 times faster on average and 1.1 times slower in the worst case); and up to 3.5 times faster than PAX layout (1.6 times faster on average and no improvement in the worst case).

4.1 Introduction

Analyzing terabytes of data on a daily basis is a common task for many enterprises such as Google, Facebook, and Yahoo!. With this trend, MapReduce [46] is quickly becoming the *de facto* standard for large-scale analysis in industry. However, it has been shown that MapReduce suffers from very slow execution times in analytical queries compared to DBMSs [101]. Several recent works have improved the performance of MapReduce. Figure 4.1 illustrates the research focus of some of these research works. For instance, Pig Latin by Olston et al. proposed a new interface to execute MapReduce jobs [99] (top-left box in Figure 4.1); other researchers proposed HadoopToSQL [77]

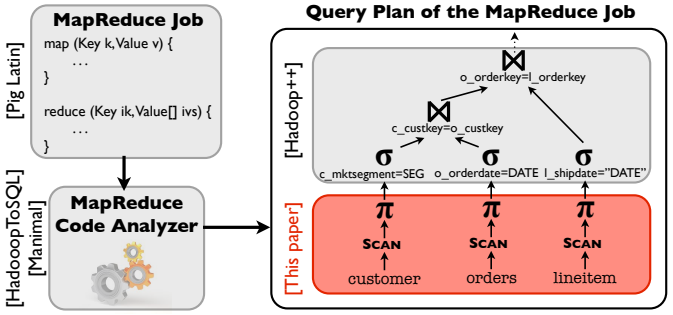


FIGURE 4.1: Example of some research works in MapReduce.

and Manimal [22] to automatically analyze the code of MapReduce jobs in order to produce more efficient query plans (bottom-left box); we recently proposed Hadoop++, a new system that improves the performance of MapReduce jobs by injecting code into the MapReduce query plans [49] (top-right box). Many other works have focused on improving MapReduce in other aspects [75, 85, 93, 97, 104]. However, none of these works considers the impact of data layouts, per distributed file system (DFS) data block, on the data read performance of MapReduce jobs. This chapter fills this gap: we analyze the different data layouts in detail. The red part in Figure 4.1 illustrates the focus of this chapter.

4.1.1 Background and Motivation

Traditional Layouts in MapReduce. Currently, MapReduce processes input data blocks in a strictly row-oriented fashion. Thus, all tuple attributes have to be read from disk even if only some of them are relevant to process a given task. The disadvantage of a row-oriented layout has been thoroughly researched in the context of column stores [3, 4, 28, 106, 116]. However, in a distributed system a column store has severe drawbacks as the data blocks for different columns may reside on different nodes. Thus, whenever a query references more than one attribute, columns have to be sent through the network in order to merge different attributes values into a row (*tuple reconstruction*). For instance, consider a table `AccessLog` containing access log-records of a web server. Assume the following simple SQL-query:

```
SELECT url, sourceip FROM AccessLog
WHERE url LIKE '%.edu%';
```

To process this query MapReduce needs to either: (1) fetch and scan all columns, join them to reconstruct tuples containing attribute values for both `url` and `sourceip`, and then filter those tuples to only return the ones containing ".edu" in their `url`; or (2) scan column `url`, collect the tuple-IDs of matching tuples and match them to retrieve the `sourceip` value of each tuple. The latter process is called late materialization [2]. In either case, the problem is that the attributes referenced after the `SELECT` clause have to be fetched over the network in many cases. This can significantly decrease the performance of MapReduce jobs.

Hybrid Layout in MapReduce. For these reasons a clever optimization is to use a hybrid layout of columns and rows. The idea is to keep the same data on a block as we

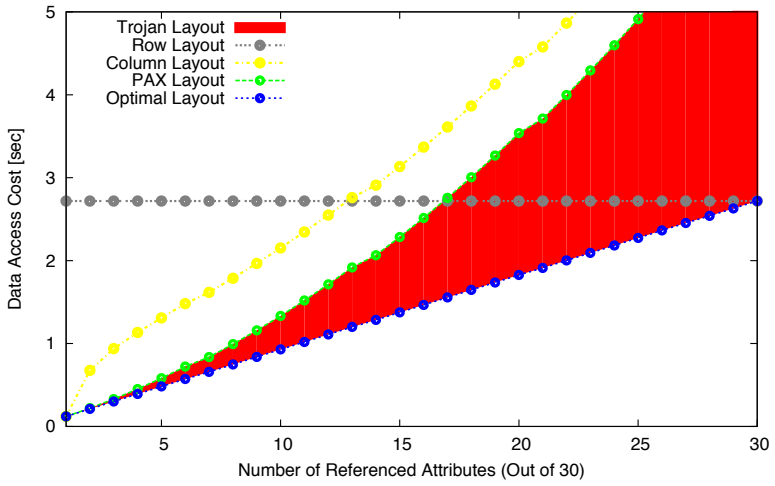


FIGURE 4.2: Data access costs for different data layouts in Hadoop.

would keep in a row layout. However, in contrast to a row layout, *inside* a block data is organized into a column layout. This approach is termed PAX (partition across) and was first proposed in the context of page organization in a DBMS [11]. Recently, it was also introduced in MapReduce [33, 125]. Using PAX in MapReduce has big advantages: (1) tuple reconstruction does not trigger expensive fetches over the network, as all data values belonging to a tuple are locally available inside a block, (2) the execution pipeline does not have to be changed at all to implement complex tuple reconstructing joins, and (3) as data blocks are typically large, about 256 MB, the subpage containing data for a specific attribute is very large. For instance, assuming a table having 30 attributes of equal size, each subpage still contains 8.5 MB of data! As a result, reading a subpage amounts to a sequential scan of 8.5 MB of data, which is typically very fast on disk. To process the SQL query mentioned above, it is then sufficient to read the subpage containing data for attribute `url` only. Then, for the qualifying tuples, we only need to access data from the subpages containing the `sourceip`. For this, we may scan that subpage entirely or elevator-scan the subpage skipping some parts in the scan. In either case, we do not trigger any network requests to fetch missing attributes. All data is local *within* the data block.

Different Layout Performance in MapReduce. Figure 4.2 shows the comparison of the estimated access cost of running a MapReduce job on each of the Row, Column, and PAX Layouts. Additionally, we consider the Optimal Layout, which co-locates all attributes referenced by any incoming query into a single column group. We use a query referencing a single input table of 30 attributes. Our cost model considers random and sequential I/O, network, and even the scheduling decisions made by the MapReduce scheduler (see Tables 4.1 and 4.2 for details of the cost model and this simulation). The results in Figure 4.2 show that Column Layout is not competitive compared to PAX Layout in MapReduce in a distributed setting. This is due to the high costs for fetching missing attribute values over the network as explained above. We also observe that PAX Layout is better than Row Layout for up to 17 (out of 30) referenced attributes. Beyond that, PAX Layout is worse than Row Layout, because the number of individual seeks in PAX adds considerable random I/O due to buffered reads of the individual subpages. In addition, tuple reconstruction in PAX adds considerable CPU costs. Even if PAX Layout seems to perform well in many cases, we observe that there exists a big gap between PAX Layout and Optimal Layout. This is because the Optimal Layout always

groups all referenced attributes together, thereby requiring fewer seeks and no tuple reconstruction. Therefore, it is quite important for the performance of applications to pack as many referenced attributes as possible co-located together.

4.1.2 Our Approach and Research Challenges

In this chapter, we propose a new approach coined *Trojan Layout*. Like PAX, Trojan Layout keeps the same data inside a block. However, in contrast to PAX, we allow for any internal data layout inside a block. The possible improvement of Trojan Layouts over PAX is depicted by the red space in Figure 4.2. Interestingly, we already see an improvement of ∼270% over Column Layout and of ∼20% over Pax Layout for five referenced attributes.

Additionally, we exploit the existing data block replication in Hadoop DFS (HDFS) to create different Trojan Layouts on a per-replica basis. This means that rather than keeping all data block replicas in the same layout, we use different Trojan Layouts for each replica. Each replica is optimized for a different subclass of queries. As a result, every incoming query can be scheduled to the most suitable data block replica. In a special case this would efficiently mimic fractured mirrors [106], which maintain two copies of the data: one in Row Layout and other in Column Layout. The reader may think that this is also possible in HDFS by using pure Row and Column Layouts. However, doing so would significantly impact the fault-tolerance properties of HDFS, because a data block replica would not contain the same data in Row Layout as in Column Layout. Therefore, complex mechanisms would be required to identify, track, and reconstruct lost data block replicas.

The idea of Trojan Layouts triggers a number of interesting research challenges. First, we have to cluster a given workload into query groups based on their access pattern in order to better exploit different data block replicas. Second, we need to invent efficient algorithms to determine the right Trojan Layout for each data block replica. Although some existing work from vertical partitioning may be leveraged [9, 60, 64], these algorithms have issues. They have to be extended to (i) improve the quality of vertical partitioning, and (ii) support replicas of the same block in different layouts. Third, we should not force users into manually defining data block layouts. If we did that, we might eventually end up turning MapReduce into yet another DBMS, with a few hundred different knobs to be properly set by a skilled (and expensive) database administrator. However, the ease-of-use and the low administration costs of MapReduce are some of its biggest advantages over DBMSs.

Therefore, the main problem we tackle in this chapter is as follows. Given an incoming query workload, we have to determine the right Trojan Layout for each data block replica that: (i) approaches to optimal layouts in performance, (ii) keeps the interface of MapReduce intact, and (iii) is zero-admin, which is extremely important for future distributed systems as emphasized in the conclusion section of the ten-year best paper award of Surajit Chaudhuri [30].

4.1.3 Contributions

Trojan Layouts are inspired by PAX in the sense that we only change the internal organization of a data block and not among data blocks. However, we considerably depart from PAX as we can: (i) co-locate attributes together according to query workloads, (ii) use different Trojan Layouts for different data block replicas, and (iii) in a special case, mimic fractured mirrors: having the best from both PAX and Row Layouts. In summary, we make the following key contributions:

1. We propose a column grouping algorithm in which we first (i) determine column groups using a novel interestingness measure, which denotes how well a set of attributes speeds up most or all queries in a workload; and then (ii) pack the column groups in order to maximize the total interestingness of data blocks. We use this algorithm as a basis to determine the Trojan Layout of data blocks in HDFS. It is worth noting that even if we focus on MapReduce in this chapter, one can use our column grouping algorithm in other domains as well.

2. We exploit default HDFS data replication to create a different Trojan Layout per data block replica. For this, we first show how to apply our column grouping algorithm for query grouping as well, i.e. for clustering queries in a workload according to their access patterns. We then map each resulting query group to one data block replica so as to compute the Trojan Layout for such a replica.

3. We present Trojan HDFS, a (per-replica) Trojan Layout aware HDFS. At data upload time, Trojan HDFS automatically transforms data block replicas into their corresponding Trojan Layouts; it hides all messy details from the user. Thereafter, Trojan HDFS keeps track of Trojan Layouts for each data block replica. With Trojan HDFS, neither the MapReduce processing pipeline nor the MapReduce interface are changed at all.

4. We evaluate Trojan Layouts using three real-world workloads: TPC-H, Star Schema Benchmark (SSB), and Sloan Digital Sky Survey (SDSS). The results demonstrate that Trojan Layouts allow MapReduce jobs to read data up to a factor of 4.8 faster than Row Layout and up to a factor of 3.5 faster than PAX Layout.

4.2 Overview

We propose Trojan Layouts as our solution to decrease the waiting time of data-intensive jobs when accessing data from HDFS. The core idea of Trojan Layouts is to internally organize data blocks into column groups according to the workload. Our approach has three phases: (1) *compute* the Trojan Layout for each data block replica, (2) *create* the computed Trojan Layouts in HDFS, and (3) *access* the existing Trojan Layouts. From the user perspective, the data analysis workflow remains the same: upload the input data and run the query workload exactly as before.

Given a query workload W, at upload time we determine the Trojan Layout for each data block replica. We then store each data block replica in its respective Trojan Layout. We illustrate the core idea of Trojan Layouts in Figure 4.3. A data block using a Trojan Layout is composed of `Header` metadata and a set of column groups (see Replica 1 in

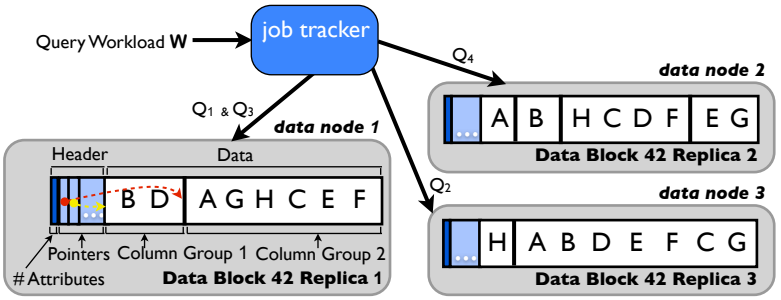


FIGURE 4.3: Per-replica Trojan Layouts in HDFS.

data node 1). The header contains the number of attributes stored in a data block and attribute pointers. An attribute pointer points to the beginning of the column group that contains that attribute. For instance, in Replica 1, the first attribute pointer (the red arrow) points to Column Group 2, which contains attribute A. The second attribute pointer would then point to Column Group 1, and so on. Each column group in turn contains a set of attributes in row-fashion, e.g. Column Group 1 in Replica 1 has tuples containing attributes B and D.

At query time, we transparently adapt an incoming MapReduce job to query the data block replica that minimizes the data access time. Then, we route the map tasks of the MapReduce job to the data nodes storing such data block replicas. For example, in Figure 4.3, the map tasks of Q_4 are routed to data node 1, while the map tasks of Q_1 and Q_3 are routed to data node 2 and those of Q_2 are routed to data node 3. Notice that in case the scheduler cannot route a map task to the best data block replica, e.g. because the data node storing such a replica is busy, the scheduler transparently fallbacks to other Trojan Layouts. An important feature of our approach is that we keep the Hadoop MapReduce interface intact by providing the right *itemize* function in the Hadoop plan [49]. As in normal MapReduce, users only care about `map` and `reduce` functions.

The salient features of our approach are as follows:

- *Invisibility.* We create and access Trojan Layouts in a way that is invisible to users.
- *Non-Invasive.* We do not change the outside HDFS unit, i.e. the data block, but rather change the internal representation of a data block. As a result, we do not require any change in the MapReduce processing pipeline for accessing Trojan Layouts.
- *Seamless Query Processing.* We seamlessly wrap the input data to a given MapReduce job. Obviously, we do this for the data block replica that minimizes the time for reading required data.
- *Rich DBMS features support.* One can easily enrich Trojan Layouts with standard DBMS optimizations, such as indexing, partitioning, and co-partitioning.
- *Per-Replica Trojan Layout*: We exploit existing data block replication in HDFS to create a different Trojan Layout for each replica so as to better fit to the workload.

We provide the details on how we compute Trojan Layouts in Section 4.3. Then, in Section 4.4, we describe how we enable HDFS to store each data block replica in a different Trojan Layout in TROJAN HDFS. In the same Section 4.4, we discuss how we physically create Trojan Layouts. In addition, we discuss the query processing and scheduling aspects of our approach.

4.3 Interestingness-based Column Grouping Algorithm

The core idea of Trojan layouts is to adapt the internal representation of data blocks, while the outside view of data for the rest of the data processing pipeline remains the same. This speeds up query execution. Since we focus on scan and projection operators in the query plan, we restrict ourselves to attribute level data adaptation, i.e. group sets of attributes together. Given a relation with attribute set $\mathbb{A}$, we consider a column group $G \subseteq \mathbb{A}$ as any subset of $\mathbb{A}$. To understand the intuition behind column grouping, let us consider the queries and their access pattern in Example 4.1 below.

Example 4.1. *Access pattern of attributes A,B,C,D in queries Q_1–Q_{10}.*

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
A	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
B	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
C	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*
D	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

In the above table, if a query accesses an attribute, then the corresponding cell has value 1, otherwise it has value 0. Notice that attributes A and B are co-accessed in queries Q_1 to Q_4 (■ in Q_1 to Q_4 of Example 4.1). Thus, column group {A,B} is interesting as it can speedup queries Q_1 to Q_4. As another example, consider the queries and their access pattern in Example 4.2 below.

Example 4.2. *Access pattern of attributes M,N,O,P in queries Q_{11}–Q_{20}.*

	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}
M	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*0*
N	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*1*	*1*
O	*0*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*1*	*1*
P	*1*	*1*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*

Observe that attributes M,N and N,O are co-accessed respectively in queries $\{Q_{11},Q_{12},Q_{18},Q_{19}\}$ and $\{Q_{12},Q_{13},Q_{19},Q_{20}\}$ (■ in Example 4.2). This makes column group {M,N,O} an interesting one. Thus, generally speaking, a column group is interesting if pairs of attributes in the column group are co-accessed (e.g. M,N and N,O in Example 4.2), even though all attributes in the column group may not be co-accessed. Consequently, in order to find the most suitable internal representation of data in data blocks, we focus on two core operations: (i) determining the interesting column groups, and (ii) packing them within a data block such that the total *interestingness* of the data block is maximized. Note that column group interestingness is directly linked to query workload performance, since we consider a column group as interesting based on whether or not its attributes are co-accessed in a query workload. Denoting a set of complete and disjoint column

groups as $\mathbb{G}'$, where $\mathbb{G}'$ is a subset of the set of all possible column groups $\mathbb{G}$, we can now describe the problem we address as follows:

Problem Statement. *Given a column group interestingness function* $Intg(G) \rightarrow [0,1]$, *find the complete and disjoint column group set* $\mathbb{G}'$ *that maximizes the total interestingness of a data block*, i.e. max $\left(\sum_{\forall G \in \mathbb{G}'} Intg(G)\right)$.

To approach the above problem, we first describe our novel *column group interestingness* function and compare its effectiveness with prior approaches below. Thereafter, we map the packing of column groups, which is an NP-hard problem [109], to a 0-1 Knapsack problem and solve it using a *branch and bound* technique.

4.3.1 Column Group Interestingness

Intuitively, a column group is highly interesting if it speeds up most or all of the queries in the workload. Thus, to formally define the interestingness of a column group, we first consider the access costs of queries in query workload W. Let Path($\texttt{Opt}, Q$) denote the access path chosen by optimizer `Opt` for query Q and let $B_A(Q, \text{Path}(\texttt{Opt}, Q))$ denote the number of bytes of attribute A read by query Q when using access path Path($\texttt{Opt}, Q$)[1]. We denote the total bytes consumed by a query Q as its *footprint* F_Q:

$$F_Q = \sum_{A \in \mathbb{A}} B_A(Q, \text{Path}(\texttt{Opt}, Q)).$$

Let us now understand which attributes contribute to the query footprint. For this, traditionally e.g. [9], one would use an attribute usage matrix $U(Q, A)$ to indicate whether or not an attribute A is referenced by query Q, i.e U(Q,A)=1, if Q references A, and 0 otherwise. However, $U(Q, A)$ considers only the attribute occurrences (■ in Example 4.1), even though attribute *non-occurrences* give equally important information: they are crucial in determining whether one attribute should co-occur with another or not. For instance, in Example 4.1, attributes C and D have common non-occurrence only in query Q_1 whereas for queries Q_2–Q_4 column group {C,D} will have redundant access of C (■). In contrast, attributes A and B have all non-occurrence in common (queries Q_5–Q_{10}) and therefore column group {A,B} is more interesting. To capture this, we generalize $U(Q, A)$ using a binary variable x, which denotes the occurrence ($x = 1$) and the non-occurrence ($x = 0$) of an attribute.

$$U_x(Q, A) = \begin{cases} U(Q, A) & \text{if x=1,} \\ 1 - U(Q, A) & \text{if x=0.} \end{cases}$$

Notice that the above usage matrix does not take into account the footprints (total byte access) of queries in which they occur.

[1]The choice of the access path depends on the optimizer, which can choose either the index access path or the table scan access path. For instance, the optimizer can come up with the access path reading lesser number of bytes. Without any loss of generality, one can supply other Path($\texttt{Opt}, Q$) functions to our algorithm.

Example 4.3. *footprint and attribute usage in queries Q_1 to Q_4.*

	Query footprint
$\mathbf{Q}_1$	*10*
$\mathbf{Q}_2$	*20*
$\mathbf{Q}_3$	*30*
$\mathbf{Q}_4$	*40*

	Q_1	Q_2	Q_3	Q_4
A	*1*	*1*	*0*	*0*
B	*0*	*0*	*1*	*1*
C	*0*	*1*	*0*	*1*

For instance, in Example 4.3, attributes A,B,C have the same frequency in the workload. However, attributes A,C are co-referenced by the cheaper query Q_2 (i.e. having smaller footprint) whereas attributes B,C are co-referenced by more expensive query Q_4 (i.e. having bigger footprint), thereby making B,C more likely to be together. Therefore, we introduce the *relative importance* (RI) of attributes, which takes query footprints into account. Intuitively, RI_A is the fractional reading cost in the events when an attribute A occurs as well as when it does not. We define RI_A as follows:

$$RI_A(x) = \frac{\sum_{Q \in W} F_Q \cdot U_x(Q, A)}{\sum_{Q \in W} F_Q}.$$

RI_A is normalized by the total workload costs to make it comparable. Since we want to co-locate attributes inside data blocks, we need to determine whether two attributes should be stored together. Thus, we also define $RI_{A,B}(x, y)$ as the *relative importance of an attribute pair* A, B in terms of the query workload cost.

$$RI_{A,B}(x, y) = \frac{\sum_{Q \in W} F_Q \cdot U_x(Q, A) \cdot U_y(Q, B)}{\sum_{Q \in W} F_Q}.$$

Now, to estimate the similarity between two attributes A and B over the range of values of x and y, we measure their mutual dependence using the mutual information [89] between them. We can compute the *mutual information between two attributes* using their relative importances as follows:

$$MI(A, B) = \sum_{x \in \{0,1\}} \sum_{y \in \{0,1\}} RI_{A,B}(x, y) \cdot \log \left(\frac{RI_{A,B}(x, y)}{RI_A(x) \cdot RI_B(y)} \right).$$

Essentially, $MI(A, B)$ measures the information (data access patterns) that attributes A and B share. We normalize $MI(A, B)$ by the minimum entropies of the two attributes to normalize its range between 0 and 1, i.e. $nMI(A, B) = \frac{MI(A,B)}{min(H(A),H(B))}$. Here, $H(A)$ and $H(B)$ denote the entropy of attributes A and B. For an attribute A, we compute its entropy as: $H(A) = \sum_{x \in \{0,1\}} RI_A(x) \cdot \log \left(\frac{1}{RI_A(x)} \right)$. Finally, we can define column group interestingness.

Definition 4.1. Column Group Interestingness of a column group G is the average normalized mutual information of any given attribute pair in G. Formally,

$$Intg(G) = \begin{cases} \frac{1}{\binom{|G|}{2}} \cdot \sum\limits_{\{A,B\} \in G, A \neq B} nMI(A, B) & |G| > 1, \\ \frac{1}{|\mathbb{A}|-1} \cdot \sum\limits_{A \in G, B \in \mathbb{A} \setminus G} 1 - nMI(A, B) & |G| = 1. \end{cases}$$

■

Note that for column groups having a single attribute, we take the inverse of the mutual information with any other attribute in $\mathbb{A}$. In other words, we measure the benefit of the attribute in the column group not occurring with any other attribute in $\mathbb{A}$. $Intg(G)$ has values between 0 and 1. Higher interestingness indicates higher mutual dependence within a column group.

By default, we would have to consider all column groups ($O(2^{|\mathbb{A}|})$) within a data block. In practice, we use the similar pruning method as in [9] in order to reduce the search space. We experimentally determine the threshold interestingness value and discard all column groups having interestingness below that threshold. A higher interestingness threshold produces a smaller set of candidate column groups. This has two consequences: (i) the search space for finding the best combination of column groups (introduced as column group packing in Section 4.3.2) becomes smaller, and (ii) only the attributes appearing in highly interesting column groups remain in the candidate set and are thus likely to be grouped. All remaining attributes which do not appear in any of the highly interesting column groups will end up in row layout. Apart from threshold based pruning, we can perform further aggressive pruning, for column groups having same interestingness value, in two ways: (i) keep the smallest column group to reduce redundant data read, or (ii) keep the largest column group to reduce tuple reconstruction costs.

Comparison with CG-Cost [9]. It is important to note that, in contrast to [9], our definition of interestingness produces superior interestingness measure, which we illustrate as follows. The algorithm in [9] computes the interestingness (CG-Cost) for column groups {A,B} and {C,D} in Example 4.1 as 0.4 and 0.6 respectively. Our algorithm computes interestingness (Intg) as 1.0 and 0.23 respectively, which makes much more sense since A and B always occur/not-occur together. Likewise, the algorithm in [9] computes the interestingness for both column groups {M,N,O} and {M,P} in Example 4.2 as 0.2. Our algorithm computes interestingness (Intg) as 0.278 and 0.005 respectively. Again, this makes more sense since {M,N} and {N,O} are pairwise similar making group {M,N,O} more interesting.

4.3.2 Column Group Packing as 0-1 Knapsack Problem

Once we have the candidate column groups along with their interestingness values, our goal now is to pack these column groups into a data block such that the total interestingness of all column groups in the data block is maximized. As mentioned before, this is an NP-hard problem [109]. Thus, we map it to a 0-1 knapsack problem, with an extra disjointness constraint, to solve it.

For a given column group G, let $id(G)$ denote the group identifier (a numeric in binary representation) such that its i^{th} bit is set to 1 if G contains attribute i, it is set to 0 otherwise. Given m column groups, we have to find 0-1 variables $x_1, x_2, ..., x_m$ — where x_i is 1 if column group G_i is selected and 0 otherwise — such that the total interestingness is maximized. Additionally, the sum of the group identifiers should be at most $id(\mathbb{A})$ and each of the groups should be *disjoint*. Formally, $max \sum_{i=1}^{m} Intg(G_i) \cdot x_i$ subject to:

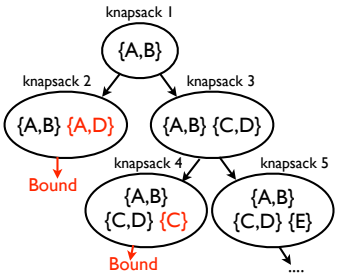


FIGURE 4.4: Branch and Bound

Algorithm 4.1: Branch And Bound Knapsack:`CGA.bbKnapsack`

Input : item, benefit, weight, weightVector, itemBitMap
Output: Max benefit item vectors, each for #column-groups from 1 to $\mathbb{A}$

```
1  if EndOfItemList(item) then
2  |   k = NumItems(itemBitMap);
3  |   if weight < MaxWeight then
4  |   |   k = k+1;
5  |   end
6  |   if k > 0 and benefit > MaxBenefit(k) then
7  |   |   CGA.SetMaxBenefit(k,benefit);
8  |   |   CGA.SetMaxBenefitItemBitMap(k,itemBitMap);
9  |   end
10 else
11 |   CGA.bbKnapsack(NextItemInList(item), benefit, weight, weightVector, itemBitMap);
12 |   if (weight & ItemWeight(item))==0 and (weight + ItemWeight(item) ≤ MaxWeight) then
13 |   |   CGA.bbKnapsack(NextItemInList(item), benefit+ItemBenefit(item), weight+ItemWeight(item),
   |   |   weightVector | ItemWeight(item), itemBitMap | ItemVector(item));
14 |   end
15 end
```

$$\sum_{i=1}^{m} id(G_i) \cdot x_i \leq id(\mathbb{A}) \tag{4.1}$$

$$x_i + x_j \leq 1, \quad \forall i,j \text{ s.t. } i \neq j \wedge G_i \cap G_j \neq \varnothing. \tag{4.2}$$

Here, (4.2) is an extra constraint to the standard 0-1 knapsack problem. Due to this additional constraint we cannot reduce the problem to a sub-problem. This is because the solution to the sub-problem may contain items which are not disjoint in the main problem. Thus, we cannot use a dynamic programming algorithm to solve this problem. However, constraint (4.2) allows us to pre-filter non-disjoint column groups. Therefore, we can apply a *branch and bound* technique. The idea is to consider a column group and its subsequent combinations with other column groups, only if it is disjoint with the column groups currently in the knapsack. Figure 4.4 illustrates this idea. We observe that column groups {A,D} and {C} bound any further branching of knapsack iterations. Algorithm 4.1 shows the pseudo-code of this technique. The algorithm denotes a column group as a knapsack item, its interestingness as the benefit, and its group identifier as weight. In case we have explored all knapsack items, we check if we have a knapsack with greater benefit than before (Lines 1-10). Else, we recursively call `CG.bbKnapsack` in two cases: (i) without taking the current item into the knapsack (Line 12), and (ii) taking the current item if it satisfies constraints (4.1) and (4.2) (Lines 13–15).

It is worth noting that our interestingness function does not consider the size of the column group. However, for operators such as joins, the number and sizes of column

Algorithm 4.2: `EnumerateAndGroup`

Input : Items items
Output: Group[][] itemGroupings

```
1  Group [] candidates = GetSubsets(items);
2  CGA.SetItemList(candidates);
3  maxWeight = 0;
4  for i=1 to size(candidates) do
5  |   maxWeight = maxWeight | (1 << i);
6  |   CGA.SetItemWeight(candidates[i], v(candidates[i]));
7  |   CGA.SetItemBenefit(candidates[i], I(candidates[i]));
8  end
9  CGA.SetMaxWeight(maxWeight);
10 Group [][] groupings = CGA.bbKnapsack(0,0,0,0,0);
11 return groupings;
```

groups would be quite important. Thus, we solve the above problem each for the number of column groups ranging from 1 to $|\mathbb{A}|$, as shown in Algorithm 4.2. We first generate the column groups (Line 1) and add them to the item list (Line 2), then we set the weight (group identifier) and benefit (interestingness) of each item (Lines 4–8). We set `maxWeight` to the maximum item weight and call `CG.bbKnapsack`, which returns a column group set each for the number of groups ranging from 1 to $|\mathbb{A}|$. As the number of solutions is equal to the number of attributes in the relation, it is now feasible to compare and pick the best partitioning using a cost model, which we describe below.

Cost Model. We model the costs for full table scan access over four different layouts: (i) Row Layout, (ii) Column Layout, (iii) PAX Layout, and (iv) Optimal Layout (which contains, for each query, perfect column groupings within a data block). Table 4.1 shows the cost model for these layouts and Table 4.2 lists the symbols used in our cost model. Our cost model considers random and sequential I/Os to read data from HDFS, network costs to transfer data blocks across data nodes, and even the scheduling decisions made by MapReduce scheduler. For network costs, we compute the probability of not finding any local data block copy and estimate the costs of transferring data from another node. Note that the Row and PAX layouts have same network transfer costs. On the other hand Column and PAX layouts have same random and sequential I/O costs, as Hadoop performs a buffered read anyways. However, these two layouts have different network costs. Table 4.2 also shows the default parameter values which we used in our simulation.

Our column grouping algorithm, along with column group pruning, works well for several realistic datasets, e.g. for TPC-H tables (having a maximum of 16 attributes) and for SSB tables (having a maximum of 17 attributes). However, finding the right Trojan Layouts for scientific data sets (having hundreds of attributes), like SDSS, becomes a difficult task to achieve. Luckily, HDFS replicates data blocks three times by default to ensure the availability of data blocks. Thus, instead of using the same data layout for all the three replicas, we create a different Trojan Layout per replica. This divide-and-conquer approach significantly reduces the complexity of our column grouping algorithm. We describe per-replica Trojan Layout in the following section.

4.4 Per-Replica Trojan Layout

In this section, we describe our novel per-replica Trojan Layouts. The core idea of per-replica Trojan Layout is to first create query groups (using the *same* column grouping

Symbol	Meaning	Model
w	# map phases (waves)	$\left\lceil \frac{N \cdot B}{S \cdot m \cdot n} \right\rceil$
$C_{\text{tr}}^{\text{row}}(S)$	transfer cost for row layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{row}}(S)$	rand I/O cost for row layout	$C_{\text{rand}} \cdot \frac{S}{b}$
$C_{\text{seq}}^{\text{row}}(S)$	seq I/O cost for row layout	$\frac{S}{\text{BW}_{\text{disk}}}$
$C_{\text{tr}}^{\text{opt}}(S)$	transfer cost for optimal layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{opt}}(S)$	rand I/O cost for optimal layout	$C_{\text{rand}} \cdot \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert}$
$C_{\text{seq}}^{\text{opt}}(S)$	seq I/O cost for optimal layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{tr}}^{\text{pax}}(S)$	transfer cost for PAX layout	$(1 - p_{1\text{r}}) \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{pax}}(S)$	rand I/O cost for PAX layout	$C_{\text{rand}} \cdot \left\lceil \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert} \right\rceil \cdot \lvert A' \rvert$
$C_{\text{seq}}^{\text{pax}}(S)$	seq I/O cost for PAX layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{tr}}^{\text{col}}(S)$	transfer cost for column layout	$\left[1 - p_{1\text{r}} + \left(1 - \frac{R}{n}\right) \cdot (\lvert A' \rvert - 1)\right] \frac{S}{\text{BW}_{\text{net}}}$
$C_{\text{rand}}^{\text{col}}(S)$	rand I/O cost for column layout	$C_{\text{rand}} \cdot \left\lceil \frac{S \cdot \lvert A' \rvert}{b \cdot \lvert A \rvert} \right\rceil \cdot \lvert A' \rvert$
$C_{\text{seq}}^{\text{col}}(S)$	seq I/O cost for column layout	$\frac{S \cdot \lvert A' \rvert}{\text{BW}_{\text{disk}} \cdot \lvert A \rvert}$
$C_{\text{scan}}^{\text{row}}$	scan cost for row layout	$\left(C_{\text{tr}}^{\text{row}}(S) + C_{\text{rand}}^{\text{row}}(S) + C_{\text{seq}}^{\text{row}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{opt}}$	scan cost for optimal layout	$\left(C_{\text{tr}}^{\text{opt}}(S) + C_{\text{rand}}^{\text{opt}}(S) + C_{\text{seq}}^{\text{opt}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{pax}}$	scan cost for PAX layout	$\left(C_{\text{tr}}^{\text{pax}}(S) + C_{\text{rand}}^{\text{pax}}(S) + C_{\text{seq}}^{\text{pax}}(S) + C_{\text{init}}^{m}\right) \cdot w$
$C_{\text{scan}}^{\text{col}}$	scan cost for column layout	$\left(C_{\text{tr}}^{\text{col}}(S) + C_{\text{rand}}^{\text{col}}(S) + C_{\text{seq}}^{\text{col}}(S) + C_{\text{init}}^{m}\right) \cdot w$

TABLE 4.1: Full table scan access cost model for different layouts in Hadoop

Symbol	Meaning	Unit	Default Value
N	number of blocks		400
B	block size	bytes	256 MB
S	split size	bytes	256 MB
R	replication factor		3
n	number of nodes		50
m	number of concurrent map tasks	2	
C_{init}^{m}	map initialization cost	seconds	0.1 sec
C_{rand}	random seek cost	seconds	0.005 sec
BW_{disk}	disk bandwidth	bytes/s	100 MB/s
BW_{net}	network bandwidth	bits/s	1 GBits/s
b	buffer size	bytes	512 KB
$p_{1\text{r}}$	probability of first replica being local		0.97
A	attribute set		{1,..,30}
A'	referenced attribute set		{1},{1,2}..

TABLE 4.2: Cost Model Symbols

algorithm) and then create column groups for each query group separately. This serves two purposes: (i) instead of creating a single layout for the entire workload, we create multiple layouts, each specialized for a part of the workload, and (ii) query grouping can significantly decrease the number of referenced attributes for each query group, which, in turn, reduces the complexity of our column grouping algorithm. Algorithm 4.3 shows the pseudo-code to compute per-replica Trojan Layouts in two steps:

(1.) Query Grouping. We first group queries in the workload based on their access pattern. Notice that column grouping is orthogonal to query grouping. However, two queries are similar if they access similar attributes just as two attributes are similar if they are accessed by similar queries. In that respect, query grouping, or rather partitioning, is very similar to column grouping. Therefore, we use our column grouping algorithm (Algorithm 4.2) for query grouping as well: we just interchange attributes with queries (Line 1). To illustrate, in the attribute usage matrix of Example 4.1 in Section 4.3, query group $\{Q_1, Q_2, Q_3\}$ has an interestingness of 0.49 whereas query group $\{Q_2, Q_3, Q_4\}$, having queries with more similar access pattern, has an interestingness of

Algorithm 4.3: `PerReplicaEnumerateAndGroup`

Input : Query[] queries, Attribute[] attributes, Int replicationFactor
Output: Group[][] perReplicaGroupings

```
1 Group [][] queryGroupings = EnumerateAndGroup(queries);
2 Group [] queryGrouping = queryGroupings[replicationFactor];
3 Group [][] perReplicaGroups;
4 for i=1 to size(queryGrouping) do
5 |   Attribute [] refAttributes = GetRef(queryGrouping[i]);
6 |   Group [][] attrGroupings = EnumerateAndGroup(refAttributes);
7 |   perReplicaGroups[i] = PickBestUsingCostModel(attrGroupings);
8 end
9 return perReplicaGroups;
```

1.0. As a result of running Algorithm 4.2 for query grouping, we receive a collection of query group sets that are complete and disjoint. Each query group set in the collection contains a different number of query groups. We pick the query group set having as many query groups as the replication factor (Line 2), thus mapping one query group to one data block replica. However, we can as well map one query group to multiple replicas, depending on the workload.

Recall that we perform query grouping in order to reduce the complexity of our column grouping algorithm. However, if the number of queries increase then the complexity of query grouping will increase as well. To deal with this, for large workloads, we apply query grouping recursively as follows: (i) first we (independently) group consecutive sets of p queries, (ii) then we XOR the queries in a query group to represent each query group as a single combined query, (iii) now we again (independently) group consecutive sets of p combined queries, (iv) we repeat this process till we have a single set of p or less queries. Here p denotes the maximum number of queries which can be grouped in reasonable time. Experimentally, we determine p to be less than or equal to 20.

(2.) Query Routing. Finally, for each query group, we get the referenced attributes and build column groups on them (Lines 5–6 in Algorithm 4.2). We pick the best column grouping among groupings of different size using a cost model[2] (Line 7).

In the remainder of this section, we discuss how we support per-replica Trojan Layouts in HDFS (Section 4.4.1); how we transform data blocks to a given Trojan Layout (Section 4.4.2); how we access Trojan Layouts in Hadoop MapReduce jobs (Section 4.4.3); and what scheduling policies we consider (Section 4.4.4).

4.4.1 Layout Aware Replication

We implemented a variant of HDFS, called Trojan HDFS, to introduce per-replica Trojan Layouts into HDFS. Trojan HDFS differs from HDFS in two aspects:

(1.) The name node in Trojan HDFS keeps a catalog of the Trojan Layouts of all data block replicas. Trojan HDFS exploits the fact that the name node maintains a triplet of pointers for each data block replica[3]. It adds a fourth pointer to this structure, which points to the Trojan Layout descriptor of the data block replica. Figure 4.5 illustrates

[2]See see Tables 4.1 and 4.2 for details of the cost model.

[3]In this triplet (i) the first pointer points to the metadata of the node storing the data block replica, (ii) the second pointer points to the previous data block stored on the same data node, and (iii) the third pointer points to the next data block stored on the same data node.

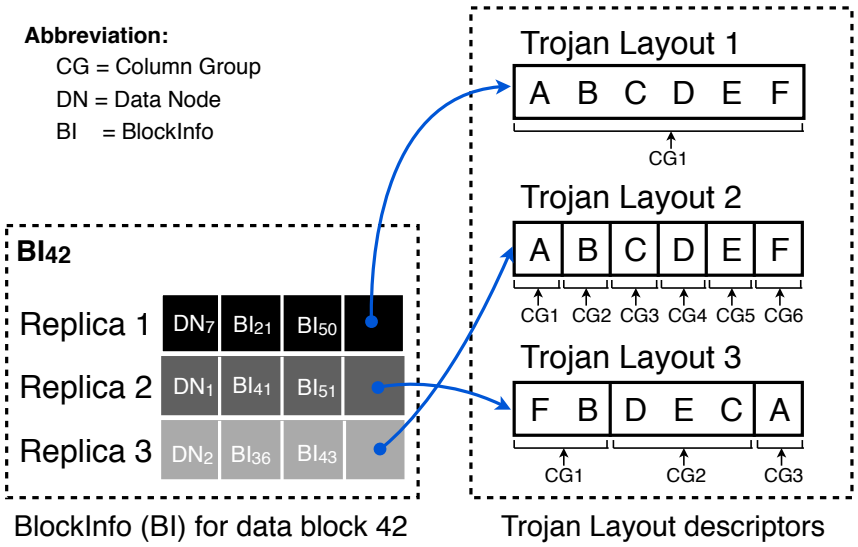


FIGURE 4.5: Quadruplets for a data block in TROJAN HDFS stored at the name node. This structured is composed: (i) of a pointer to the data node (e.g. DN 7) storing a replica (e.g. the first replica) of a data block (e.g. data block 42), (ii) of a pointer to the previous data block (e.g. data block 21) stored on DN 7, (iii) of a pointer to the next block (e.g. data block 51) stored on DN 7, and (iv) of a pointer to the Trojan Layout descriptor for that data block replica (e.g. row-layout).

this quadruplet of pointers associated to a data block replica. Note that more than one data block replica could point to the same Trojan Layout descriptor.

(2.) A data node in TROJAN HDFS asks the name node for the Trojan Layout of each data block replica stored locally. After receiving the Trojan Layout for a given data block replica, a data node internally reorganizes the data of the data block replica according to the received layout. There are two ways, for a data node, to do so: (i) reorganize a data block as soon as the data block replica is copied locally, or (ii) reorganize a data block after all replicas of the data block are copied to relevant data nodes. The reader might think the first strategy to be better, since data nodes do not have to wait for other replicas to be copied. However, this strategy generates contention between data nodes for accessing data block replicas. This is because a data node would be accessing a given local data block replica for transformation while another data node would be trying to remotely copy the same data block replica for replication. This contention will, in turn, significantly increase the data upload time. Therefore, in TROJAN HDFS, we apply the second strategy, i.e. data nodes start data block reorganizations after all replicas are copied. We showed an example of the resulting internal organization of a data block in Figure 4.3.

4.4.2 Layout Creation

We now focus on the process of uploading a file to TROJAN HDFS. In a spirit similar to databases physical design wizards, users have to run our *Trojan Layout Wizard* (TLW) to come up with the Trojan Layouts for their data sets. For this, users feed the TLW with the query workload, the schema of their data sets, and the replication factor they will use to store their data sets. Given these inputs, TLW computes the per-replica Trojan Layouts and returns a *layout configuration* file. The layout configuration file contains, for each data set, a row having data set name and per-replica Trojan Layouts ids. As

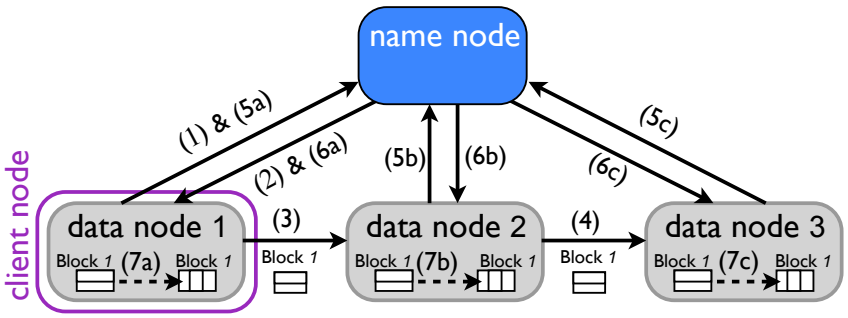


FIGURE 4.6: Process to upload a file to TROJAN HDFS

an example, the layout configuration file for TPC-H `Customers`, TPC-H `Lineitem`, SSB `LineOrder`, and SDSS `PhotoObj` can be as follows:

```
TPC-H_Customers: Row | Column | Customer_column_grouped
TPC-H_Lineitem: Row | Column | Lineitem_column_grouped
SSB_LineOrder: Row | Column | LineOrder_column_grouped
SDSS_PhotoObj: Row | Column | PhotoObj_column_grouped
```

The layout ids (e.g. Customer_column_grouped) in the above layout configuration file is mapped to actual attributes in a separate file. Users simply upload the layout configuration file into a predefined directory in TROJAN HDFS. At start up, the name node loads the layout configuration file into main memory. After this, the users can upload their data files into TROJAN HDFS exactly as in standard HDFS. Internally, TROJAN HDFS takes care of storing data block replicas in their respective Trojan Layouts, hiding all messy details from the users.

Figure 4.6 depicts the upload process with a replication factor of 3. For simplicity, we assume in Figure 4.6 that the data set to upload contains only a single data block. The idea is that as soon as all replicas of a data block are copied, the data nodes internally reorganize replicas according to their assigned Trojan Layout. In detail, the uploading process has the following steps: (1) the client node (e.g. data node 1) asks the name node to register a data block (e.g. data block 1); (2) the name node returns the set of data nodes to hold the three data block replicas (e.g. data nodes 1, 2, and 3); (3) after storing data block 1 locally, data node 1 sends a replica to data node 2; (4) data node 2 stores data block 1 locally and sends a replica to data node 3, which in turn also stores data block 1 locally; (5a)–(5c) each data node then informs the name node of the newly received data block 1; (6a)–(6c) the name node returns the Trojan Layout corresponding to the data block replica stored by each data node; (7a)–(7c) finally, each data node transforms data block 1 into its respective Trojan Layout. In case a user uploads a data set that is not in the layout configuration file, the name node asks the data nodes to keep the data layout unchanged (typically row layout).

4.4.3 Query Processing

To process an incoming MapReduce job, we first identify which attributes need to be read. We then use a cost model (Table 4.1) to automatically pick the best Trojan Layout in TROJAN HDFS for the MapReduce job. Then, we schedule map tasks to those data

Algorithm 4.4: Trojan Layout `itemize`.initialize UDF

Input: FileSplit split, Configuration job

```
1  Set ReferencedAttributes = job.getRefAtts();
2  Global FileSplit split = split;
3  Header h = ReadHeader(split);
4  Set GroupedData, ReadGroups = ∅;
5  Global StringBuilder attributeOrder = new StringBuilder ();
6  foreach attribute in h.getAttributes() do
7      if ReferencedAttributes.contains(attribute) then
8          if !ReadGroups.contains(attribute.getStartOffset()) then
9              GroupedData.add(new Group (split.readfully(attribute.getStartOffset(),
               attribute.getEndOffset())));
10             ReadGroups = attribute.getStartOffset() ;
11         end
12         attributeOrder.append (attribute.getPosition() + ",");
13     end
14 end
```

nodes storing data block replicas in the best Trojan Layout (given by the cost model). We provide an *itemize* `UDF` [49] to the map tasks so that they can read only the referenced attributes and reconstruct tuples, all invisible to the users.

Algorithm 4.4 shows the `itemize.initialize` method for enabling a map task to transparently read referenced attributes from data blocks and automatically reconstruct tuples as expected by applications (MapReduce jobs). To do so, we first get the required attributes from the job configuration (Line 1). Then, we read the header of a data block (Lines 2–3). The header information allows us to know the column groups that contains the referenced attributes (relevant column groups). We upload such relevant column groups into main memory (Line 6–10). Additionally, we keep track of the position of each referenced attribute so as to allow a map function to know how attributes are ordered in a tuple (Line 12). Now, to feed tuples to the `map` function, we simply iterate over each column group and check if a column group has more tuples. If so, we reconstruct the tuple from relevant column groups and pass it to the `map` function. Otherwise, we signal the end of tuples.

4.4.4 Scheduling Policies

By default, the Hadoop MapReduce scheduler tries to allocate map tasks to those data nodes having *any* replicas of the requested data block locally. However, with per-replica Trojan Layouts, scheduling map tasks to data nodes having different data block replicas may have quite different performance. For example, a MapReduce job requiring one attribute out of 16 would be a way faster to complete if the input data set is in column-layout. Therefore, to query per-replica Trojan Layouts, we always schedule map tasks to those nodes storing the best Trojan Layout (`Best-Layout` policy, for short). This policy is reasonable because in practice, if map tasks are slightly delayed [133], only 1% of map tasks need to fetch the best layout anyways. Still, in case of contention, we might end up delaying several map tasks with the `Best-Layout` scheduling policy. To avoid this delay, there could be two more scheduling policies to allocate these map tasks: (i) schedule map tasks to available nodes even if they do not store the best Trojan Layout; later, map tasks fetch data blocks with the best layout (`Fetch Best-Layout` policy), and (ii) schedule map tasks to those nodes storing the second best Trojan Layout; later, map tasks read the local data blocks (`2nd Best-Layout` policy). Both `Fetch Best-Layout` and `2nd Best-Layout` scheduling policies avoid delaying map tasks. However, `Fetch`

`Best-Layout` policy now incurs networks costs to fetch the best layout whereas `2nd Best-Layout` policy affects the data access performance.

We experimentally compare these three scheduling policies in Section 4.5.7.

4.5 Experimental Evaluation

We implemented our ideas on top of HDFS 0.20.3. We evaluate the performance of Trojan Layouts and compare it with Hadoop MapReduce 0.20.3 using Row Layout (Hadoop-Row) and PAX Layout (Hadoop-PAX). We ran our experiments with two main objectives in mind: (i) to show that the use of Trojan Layouts allows us to significantly improve data access performance, and (ii) to evaluate the effectiveness of our column grouping algorithm.

4.5.1 Testbed

We ran all our experiments on a physical 10-node cluster where each physical node runs five virtual nodes, using Xen virtualization, i.e., resulting in a total of 50 virtual nodes. Node virtualization is also used by Amazon to scale up its clusters. However, we showed that Amazon EC2 suffers from high variance in performance [110]. Therefore, running the experiments on our cluster allows us to get more stable results. Each physical node in our cluster has one 2.66 GHz Quad Core Xeon running 64-bit platform Linux openSuse 11.1 OS, 4x4 GB main memory, 6x750 GB SATA hard disks, and three Gigabit network cards. We set each virtual node to have a physical 750 GB hard disk and physical 3.2 GB main memory. The physical nodes are connected with a Cisco Catalyst 3750E-48PD, which uses two Gigabit Ethernet ports for each node in channel bonding mode. From now on, we refer to virtual nodes as *nodes* for clarity. We used Hadoop 0.20.3 running on Java 1.6 for all our experiments. We used Trojan HDFS to store input datasets, recall that Trojan HDFS is a variant of HDFS that supports different data layouts per-replica (see Section 4.4.2). We made the following three changes to the default HDFS settings: (i) we store data into Trojan HDFS using 256MB data blocks as in [49], (ii) we allow Hadoop to reuse the task JVM executor instead of restarting a new process per task, and (iii) we allow a node to concurrently run two map tasks and a single reduce task.

4.5.2 Datasets and Benchmarks

To better validate Trojan Layouts, we used three real-world datasets and benchmarks: TPC-H, Star Schema Benchmark (SSB), and the Sloan Digital Sky Survey[4] (SDSS).

TPC-H. We generated data for the `Customers` and `LineItem` tables using the TPC-H `DBGEN` data generator tool. We used a scale factor of $1,000$ to generate 50 files of data, which results into a total of 23.74 GB for the `Customers` table and into a total of 759 GB for the `LineItem` table. Since TPC-H `Customers` table appears in only eight queries of the TPC-H benchmark, we consider only the first eight queries of all other tables as well.

[4]For further details on SDSS visit: http://www.sdss.org/

SSB. We generated data for `LineOrder` table using the SSB `DBGEN` tool. We used a scale factor of 1,000 to generate 50 files of data, which results into a total of 600 GB in total. We consider the first eight SSB queries, i.e. we use all three variants of the first two queries and the first two variants of the third query.

SDSS. We used ∼50 GB of real-world data provided by SDSS for the `PhotoObj` table. As for TPC-H and SSB, we consider the first eight relevant SDSS queries. Notice that the `PhotoObj` table has 446 attributes in total. However, to be fair to HADOOP ROW, for all three systems we consider only those 46 attributes which are referenced by our SDSS benchmark queries.

4.5.3 Benchmarks Queries

In this section we enumerate the workload queries used in our experiments (see Section 5.6). For each of the four tables — TPC-H `Customer`, TPC-H `Linitem`, SSB `LineOrder`, and SDSS `PhotoObj` — we pick those first eight queries, in their respective benchmarks, which touch at least one attribute from them. The reason for doing this was that only eight TPC-H queries access any of the attributes in Customer table. Hence, in order to be fair and have equal-sized workload for all datasets, we picked just the first eight queries over the four datasets. Tables 4.3, 4.4, 4.5,and 4.6 below list the queries which we use over the three layouts — Row Layout, PAX Layout, and Trojan Layout — in our experiments.

Query Number	Query	Referenced Attributes
Q_1	TPC-H Query 3	0,6
Q_2	TPC-H Query 5	0,3
Q_3	TPC-H Query 7	0,3
Q_4	TPC-H Query 8	0,3
Q_5	TPC-H Query 10	0,1,2,3,4,5,7
Q_6	TPC-H Query 13	0
Q_7	TPC-H Query 18	0,1
Q_8	TPC-H Query 22	0,4,5

TABLE 4.3: TPC-H `Customers` Queries (#total attributes = 8)

Query Number	Query	Referenced Attributes
Q_1	TPC-H Query 1	4,5,6,7,8,9,10
Q_2	TPC-H Query 3	0,5,6,10
Q_3	TPC-H Query 4	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
Q_4	TPC-H Query 5	0,2,5,6
Q_5	TPC-H Query 6	4,5,6,10
Q_6	TPC-H Query 7	0,2,5,6,10
Q_7	TPC-H Query 8	0,1,2,5,6
Q_8	TPC-H Query 9	0,1,2,4,5,6

TABLE 4.4: TPC-H `Lineitem` Queries (#total attributes = 16)

4.5.4 Layout Details

In this section, we show and discuss the Trojan Layouts that we obtained from our column grouping algorithm. Since we create Trojan data layouts per replica of a given data block, let us first look at the query groupings generated for our experimental datasets. Recall that we use the same algorithm for generating query groups as we use for generating column groups i.e. we simply invert the attribute usage matrix. Figure 4.7

Query Number	Query	Referenced Attributes
Q_1	SSB Query 1.1	5,8,9,11
Q_2	SSB Query 1.2	5,8,9,11
Q_3	SSB Query 1.3	5,8,9,11
Q_4	SSB Query 2.1	3,4,5,12
Q_5	SSB Query 2.2	3,4,5,12
Q_6	SSB Query 2.3	3,4,5,12
Q_7	SSB Query 3.1	2,4,5,12
Q_8	SSB Query 3.2	2,4,5,12

TABLE 4.5: SSB `LineOrder` Queries (#total attributes = 17)

Query Number	Query	Referenced Attributes
Q_1	Basic SELECT-FROM-WHERE	0,1,2,3
Q_2	Moving Asteroids	0,4,5
Q_3	Using three tables	0,2,3,6,7,8,9,10,11,12,17 18,19,20,21,22,23,28,29
Q_4	Selected neighbors in run	0,1,6,7,23,24,25,26,27,30 31,32,33,34,35,36,37,38
Q_5	Gridded galaxy counts	2,3,37,39
Q_6	Stars multiply measured	0,8,9,10,11,12,30,37 40,41,42,43,44
Q_7	Spatial Queries with HTM functions	0,2,3
Q_8	Checking if objects are in SDSS footprint	0,2,3

TABLE 4.6: SDSS `PhotoObj` Queries (#total attributes = 46)

shows the query groups for the first relevant eight queries on TPC-H `Customer`, TPC-H `Lineitem`, SSB `LineOrder`, and SDSS `PhotoObj` datasets. Here, we assume three replicas per data block and generate three query groups, one for each replica.

Dataset	Replica 1	Replica 2	Replica 3
TPC-H `Customer`	Q_2, Q_3, Q_4	Q_5	Q_1, Q_6, Q_7, Q_8
TPC-H `Lineitem`	Q_1	Q_5	Q_2, Q_3, Q_4, Q_6, Q_7, Q_8
SSB `LineOrder`	$Q1$, $Q2$, $Q3$	$Q4$, $Q5$, $Q6$	Q_7, Q_8
SDSS `PhotoObj`	Q_4	Q_6	Q_1, Q_2, Q_3, Q_5, Q_7, Q_8

TABLE 4.7: Query Grouping

Note that two query groups in TPC-H `Customer`, TPC-H `Lineitem`, and SDSS `PhotoObj` datasets as well as all three query groups in SSB `LineOrder` dataset match perfectly with the attribute access pattern. This means there is no redundant attribute accessed and there are no joins in tuple reconstruction. We map each query group to a replica and then compute the column grouping for each replica separately.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	1,2,4,5,6,7	6	2,3,7
CG_2	0,3	0,1,2,3,4,5,7	0
CG_3			1
CG_4			4,5
CG_5			6

TABLE 4.8: TPC-H `Customer` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Tables 4.8, 4.9, 4.10, and 4.11 show the column groupings for the three data block replicas of TPC-H `Customer`, TPC-H `Lineitem`, SSB `LineOrder`, and SDS `PhotoObj` tables respectively. In these tables, we represent the attributes as integer (starting from 0) attribute IDs in the same sequence as they appear in their benchmark datasets.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	0,1,2,3,11, 12,13,14,15	0,1,2,3,7,8,9 11,12,13,14,15	0,2,5,6,10
CG_2	4,5,6,7,8,9,10	4,5,6,10	1,4
CG_3			8,9,11
CG_4			14,15
CG_5			7,12
CG_6			3,13

TABLE 4.9: TPC-H `Lineitem` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	0,1,2,3,4,6 7,10,12,13 14,15,16	0,1,2,6,7 8,9,10,11 13,14,15,16	0,1,3,6,7 8,9,10,11 13,14,15,16
CG_2	5,8,9,11	3,4,5,12	2,4,5,12

TABLE 4.10: SSB `LineOrder` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

Column Groups	Replica 1	Replica 2	Replica 3
CG_1	2,3,4,5,8,9 10,11,12,13,14 15,16,17,18,19 20,21,22,28,29 39,40,41,42,43 44,45	1,2,3,4,5,6,7,13 14,15,16,17,18 19,20,21,22,23 24,25,26,27,28 29,31,32,33,34 35,36,38,39,45	13,14,15,16,24 25,26,27,30,31 32,33,34,35,36 38,40,41,42,43 44,45
CG_2	0,1,6,7,23,24 25,26,27,30,31 32,33,34,35,36 37,38	0,8,9,10,11,12 30,37,40,41,42 43,44	0,1,2,3
CG_3			4,5
CG_4			37,39
CG_5			11,19
CG_6			7,28
CG_7			8,23
CG_8			12,18
CG_9			10,20
CG_{10}			17,21
CG_{11}			6,9
CG_{12}			22,29

TABLE 4.11: SDSS `PhotoObj` Column Groups. Green replicas have perfect Trojan Layouts for the queries routed to them.

For instance, in Table 4.8, attribute IDs 0 to 7 denote the eight attributes of TPC-H `Customer` table. Note that each replica may have a different number of column groups, e.g. SDSS `PhotoObj` has two column groups each in the first two replicas whereas it has twelve column groups in the last replica (see Table 4.11). However, the union of columns groups in each of the replicas contains all attributes present in the dataset. It is worth to notice that at least two replicas (shown as green in Tables 4.8, 4.9, 4.10, and 4.11) of each dataset perfectly fit the queries routed to them i.e. the queries do not access any redundant attributes nor need any joins for tuple reconstruction.

Finally, observe that increasing the number of replicas allows us to create more variety of column groupings and thus hence fit a heterogenous query workload better. In the extreme case, we could maintain one replica, with perfect column grouping, for each

query in the workload. However, the downside is that such an arrangement incurs exorbitant storage costs. Nevertheless, the beauty of Trojan Layouts is that it exploits the default replication in parallel data processing systems without touching the distributed data storage configurations.

4.5.5 Experiment Methodology

We first evaluate how well Trojan Layouts allow MapReduce jobs to improve their performance. In particular, we evaluate how well our approach exploits different data replicas to fit a given workload. As this chapter focuses on scan and projection operators of MapReduce jobs (see Figure 4.1), we implement map-phase-only MapReduce jobs for all our benchmark queries. The reason to do so is that Trojan Layouts improve the performance of MapReduce jobs by improving the way they read data from HDFS, which is done in the map phase of MapReduce jobs. Furthermore, we do not analyze the MapReduce job and assume that we know the data access pattern, i.e. the attributes accessed by each query. Recent works in other aspects of MapReduce (shown in Figure 4.1) have described how to extract these data access patterns from MapReduce jobs [22, 77]. We run each benchmark three times, measure the time it takes to read the required data from disk — i.e. the elapsed time between the initialization and finalization of the `itemize` UDF — and report the improvement factor of our approach based on the average reading time of the trials.

4.5.6 Per-Replica Trojan Layout Performance

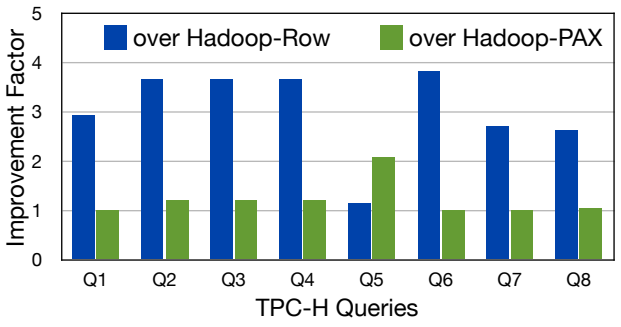
In this section, we evaluate the data access time improvement of Trojan Layouts over Hadoop-Row and Hadoop-PAX. Let us first evaluate these three data layouts in terms of redundant attributes reads and attribute joins for tuple reconstruction. For this, we analyzed the query groupings and their Trojan Layouts (see Section 4.5.4 for layout details) and we observed that in all datasets at least two query groups fit perfectly to its corresponding Trojan Layout. Hence, per-replica Trojan Layouts significantly reduce redundant attribute access as well as tuple reconstruction overhead. Table 4.12 summarizes this observation.

	#Redundant Attributes Read	*#Joins in Tuple Reconstruction*
Hadoop-Row	525	0
Hadoop-PAX	0	139
Trojan Layout	14	20

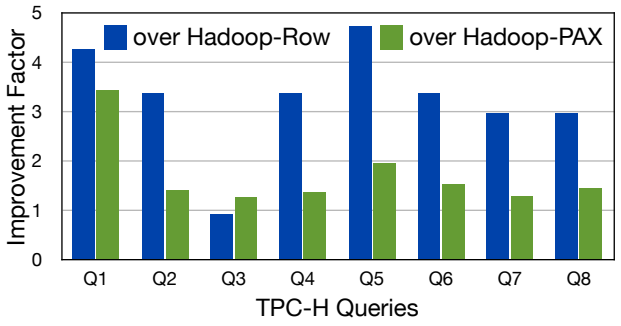
Table 4.12: Per-replica Trojan Layout analysis

We observe that Trojan Layouts allow us to read ∼37 times less redundant attributes than Hadoop-Row and to perform ∼7 times less attribute joins for reconstructing tuples than Hadoop-PAX. Thus, Trojan Layouts provide for a good trade-off between the number of redundant attributes and the number of joins in tuple reconstruction (green cells). This is in contrast to Hadoop-Row and Hadoop-PAX, which are at the two extremes (red cells).

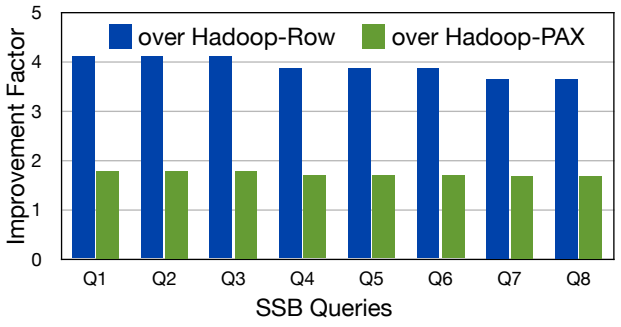
Figure 4.7 illustrates the improvement of data access time when using Trojan Layouts over Hadoop-Row and Hadoop-PAX. We observe that for those queries referencing



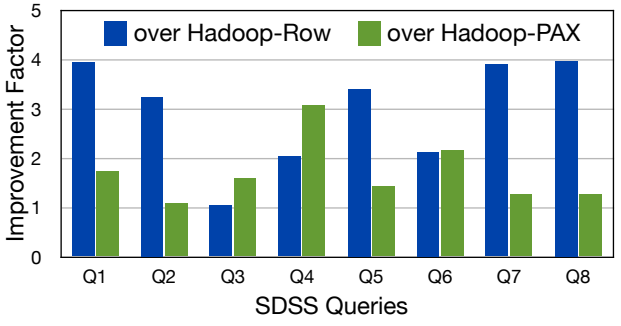
(a) TPC-H `Customer`



(b) TPC-H `LineItem`



(c) SSB `LineOrder`



(d) SDSS `PhotoObj`

FIGURE 4.7: Improvement of data access time when using Trojan Layouts over HADOOP-ROW and HADOOP-PAX.

few attributes, e.g. Q_4 in `LineItem` and all queries in `LineOrder`, Trojan Layouts improve HADOOP-ROW up to factor of 4.8. Indeed, this is because HADOOP-ROW reads a large number of redundant attributes as shown in Table 4.12. In particular, we observe that HADOOP-ROW slightly outperforms Trojan Layouts only for Q_3 in `LineItem`. This

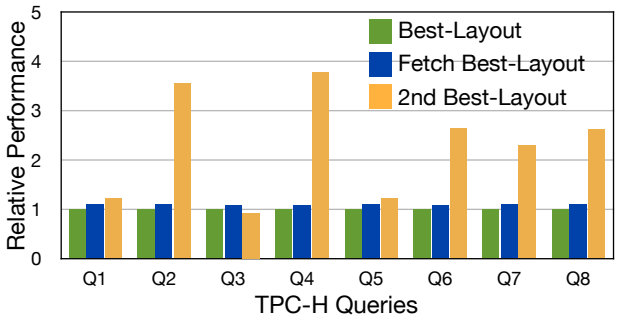


FIGURE 4.8: Worst-case relative data access performance when using different scheduling policies. We observe that the `2nd Best-Layout` policy significantly hurts performance for some queries, while the `Fetch Best-Layout` policy has an overhead of at most 9% over the `Best-Layout` policy. Therefore, in practice, one should try to use the best layout to perform queries even if data blocks has to be copied through the network.

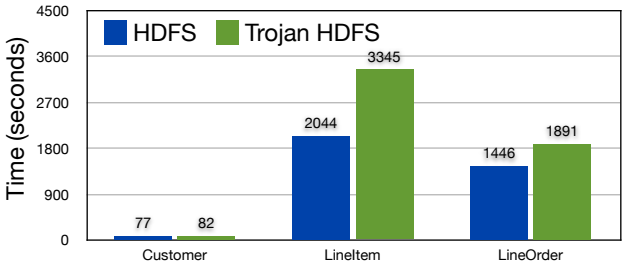
is because all attributes are referenced and Trojan Layouts have an extra tuple reconstruction cost that HADOOP-ROW does not have. On the other side, we observe that for those queries referencing many attributes, e.g. Q_1 in `LineItem` and Q_4 in `PhotoObj`, Trojan Layouts outperform HADOOP-PAX up to a factor of 3.5. The reason is that tuple reconstruction cost in HADOOP-PAX increases as the number of referenced attributes increases as well. Trojan Layouts amortize tuple reconstruction cost by co-locating attributes in the same column groups. Furthermore, the results show that Trojan Layouts never perform worse than HADOOP-PAX, having at least the same performance as HADOOP-PAX in the worst case (e.g. Q_6–Q_8 in `Customer`).

Overall, our experimental results show that Trojan Layouts significantly outperform HADOOP-ROW as well as HADOOP-PAX. Our experimental results also support the simulation results we presented in Figure 4.2.

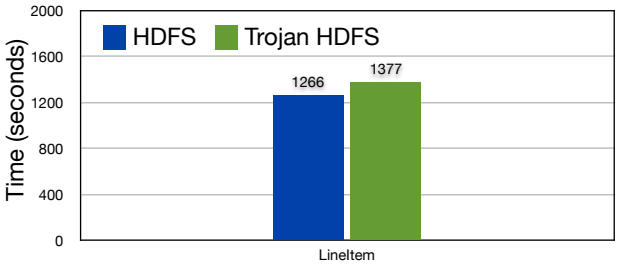
4.5.7 Comparing Scheduling Policies

In the above experiments, we considered the `Best-Layout` scheduling policy (see Section 4.4.4), which always allocates map tasks to those nodes storing the best Trojan Layout for incoming map tasks. However, as discussed in Section 4.4.4, one could apply two other scheduling policies as well: the `Fetch Best-Layout` policy and the `2nd Best-Layout` policy. To understand which policy performs better, we measure their relative performance with respect to the `Best-Layout` policy over `TPC-H Lineitem` table. We compute the relative performance of a given scheduling policy as the ratio of the data access time of the given policy to the `Best-Layout` policy.

Figure 4.8 shows the results of these experiments. As expected, the `Best-Layout` policy performs better than the other two policies. However, we observe that the `Fetch Best-Layout` policy performs almost as well as the `Best-Layout` policy. This is not the case for the `2nd Best-Layout` policy, which is slower by a factor of up to ~3.8. This is because map tasks end up reading all attributes from disk in many cases. Thus, we can conclude that, when having data block replicas in different layouts, one should apply only the `Best-Layout` and `Fetch Best-Layout` policies.



(a) Using 50 virtual nodes.



(b) Using 10 physical nodes.

FIGURE 4.9: Comparison of Data Loading Times in `Trojan` and standard HDFS.

4.5.8 Data Loading

Now we compare and analyze the data load performance of `Trojan` HDFS with standard HDFS. On a cluster of 50 virtual nodes, we consider the data load times of three data sets from our benchmarks: TPC-H `Customer`, TPC-H `Lineitem`, and SSB `LineOrder`. For each of these data sets, we load the data files on all data nodes in parallel, i.e. each of the fifty nodes loads ∼470 MB of TPC-H `Customer` data (23.74 GB in total), ∼15 GB of TPC-H `Lineitem` data (759 GB in total), and 12 GB of SSB `LineOrder` data (600 GB in total). We use the same command-line utility for both `Trojan` as well as standard HDFS.

Figure 4.9(a) illustrates the results of loading these three data sets into `Trojan` and standard HDFS. As expected, standard HDFS is faster than `Trojan HDFS` because it simply copies the data from local hard disks to the distributed file system. On the other hand, `Trojan` HDFS parses the data sets into binary representation and formats them into their Trojan Layout. However, from Figure 4.9(a), we see that the difference between the loading times of `Trojan` and standard HDFS becomes significantly high for larger tables, e.g. TPC-H `Lineitem` table. The reason for this overhead is that the `Trojan` HDFS is CPU-intensive due to data parsing and layouts transformation. However, because of node virtualization more than 60% of the CPU resources are already consumed. Furthermore, since each physical node of our cluster has a Quad-core processor (see Section 4.5.1), each virtual node gets only ∼0.7 core. These two problems slow down the data loading in `Trojan` HDFS considerably. Standard HDFS, on the other hand, is I/O intensive and therefore does not get affected.

To actually verify our claims, we repeated the data loading experiments for `Lineitem` using only the 10 physical nodes, i.e., without any node virtualization. However, we still keep the amount of data per data node same. Figure 4.9(b) shows the loading times of `Trojan` and standard HDFS. We observe that `Trojan` HDFS now compares very well

with standard HDFS. This is because the data nodes get much better CPU resources by not sharing the Quad-core processors anymore.

In summary, we can say that with appropriate cluster settings, the data load time overhead of `Trojan` HDFS is negligible. Furthermore, the one-time data load cost of `Trojan` HDFS pays off as recurring speed-ups over several MapReduce jobs.

4.5.9 Comparison with HYRISE

In this section, we compare our column grouping algorithm with recently proposed HYRISE [60] layout selection algorithm. HYRISE proposes a cost-based divide and conquer technique for layout selection. It divides the set of candidate column groups using a k-way partitioner and then applies brute force search for the best layout per partition. Thereafter, it tries to merge column groups across partitions, before producing the final layout. This approach effectively improves upon the complexity of prior column grouping algorithms, e.g. [64]. However, it has two major problems: (i) there is little column grouping quality control, and (ii) query grouping, and hence per-replica layouts, is not possible.

In contrast, our interestingness-based column grouping algorithm takes the quality of column grouping into account. To illustrate this, we implemented HYRISE layout selection algorithm. Table 4.13 shows the redundant attributes accessed and tuple reconstruction joins in HYRISE and Trojan layouts.

	#Redundant Attributes Read	*#Joins in Tuple Reconstruction*
HYRISE Layout	2	64
Trojan Layout	14	20

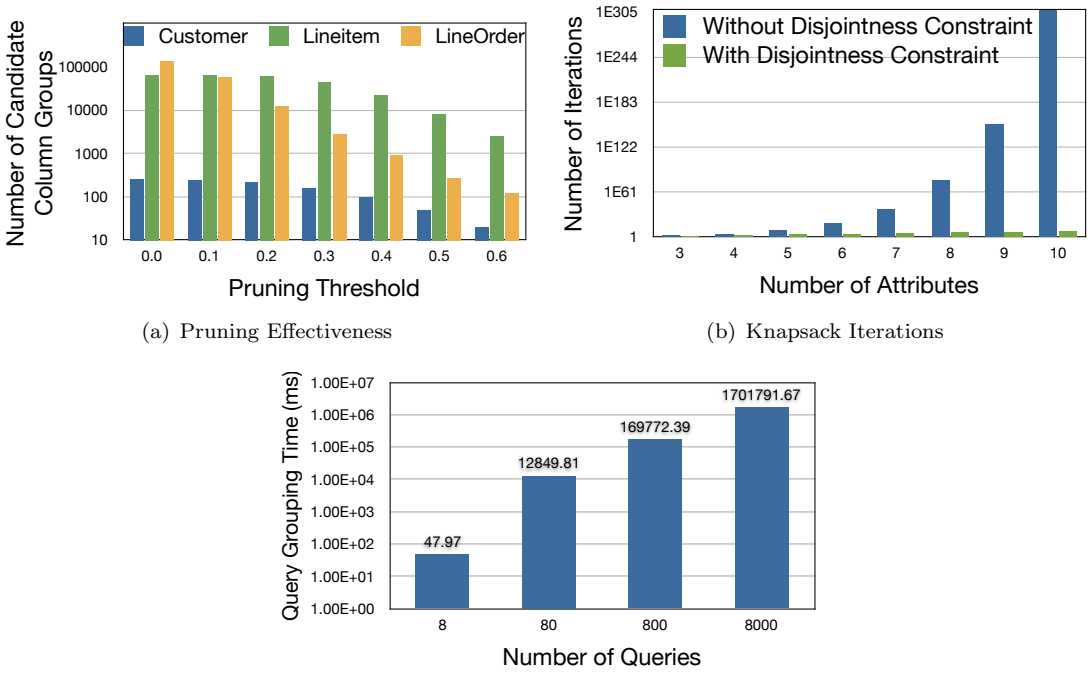
TABLE 4.13: Quality Comparison of HYRISE and Trojan Layouts

We can see that even though HYRISE significantly reduces the redundant attributes accessed, it still incurs a very high tuple reconstruction cost. In contrast, Trojan Layout minimizes tuple reconstruction cost while allowing for additional (cheap) redundant reads. To verify our claim, we ran our benchmark queries over HYRISE layout. Indeed, our results showed over 14% improvement in total runtime of Trojan Layouts over HYRISE layouts on TPC-H `Lineitem`, TPC-H `Customer`, and SSB `LineOrder` tables. Similarly, Trojan Layouts have an improvement of 5.9% over HYRISE layouts on SDSS `PhotoObj` table. Lower improvement on `PhotoObj` table is due to the large number of attributes accessed by queries and the skewed query groups produced in our Trojan Layout (2 groups of 1 query each, and 1 group of 6 queries).

4.5.10 Grouping Algorithm Performance and Scalability

We now focus on the effectiveness of our algorithm to group attributes inside a data block.

Column Group Pruning. First of all, we show the effectiveness of our interestingness-based column group pruning. Figure 4.10(a) shows the pruning performance over TPC-H Customer, TPC-H `LineItem`, and SSB `LineOrder`. We observe that candidate column groups get pruned progressively with the pruning threshold.



(a) Pruning Effectiveness

(b) Knapsack Iterations

(c) Query Grouping Scalability

FIGURE 4.10: Performance of our column grouping algorithm.

Number of Iterations. Next, we show the effect of adding the disjointness constraint (Equation 4.2) to our knapsack formulation. Figure 4.10(b) compares the number of iterations with and without the disjointness constraint. Recall that the disjointness constraint prevented us from using dynamic programming algorithm. However, as we see from the figure, the disjointness constraint significantly reduces the number of iterations in our algorithm.

Query Workload Scalability. Finally, we show the scalability of our algorithm with query workload. Recall that to deal with large number of queries, we apply our grouping algorithm recursively, i.e. we first independently group sets of queries, then we independently group sets of query groups, and so on. Figure 4.10(c) shows the time taken to create query groups when scaling the number of queries. For instance, for $8,000$ queries, the time taken to group the queries is around 28 minutes. This is acceptable, given that grouping is an offline process. Thus, our algorithm scales well with query workload size.

4.6 Related Work

Column Layouts in Traditional Systems. As an alternative to traditional n-ary storage model, Decomposition Storage Model (DSM) [40] was the earliest approach to store data in a column-oriented layout, i.e. all values of an attribute stored together. Later, researchers proposed PAX [11], a page-level column layout, to improve cache performance. Finally, Column-Stores [4] reinvented DSM to significantly improve upon storage requirements as well as query processing. However, all these approaches were designed for single-node data processing systems and thus do not care about replication. In a distributed setting, Fractured Mirrors [106] keeps two replicas of data: one in row

and the other in column layout. However, it has a fixed number of replicas (two) as well as layouts (row and column). Trojan layouts, on the other hand, can work with any number of replicas and can even create hybrid layouts. Still, in special cases, Trojan layouts can mimic Fractured Mirrors.

Column Layouts in MapReduce. Row layouts in MapReduce could incur significant overheads over large datasets. To deal with this, recent works such as Cheetah [33] and ES2 [24] propose to use block-level PAX layouts in MapReduce. Similarly, a more recent work [52] creates, for each horizontal range, a different physical file per attribute. However, all these works create identical layouts for different replicas of a data block and do not consider column grouping. In contrast, Trojan Layouts can create a different layout per data block replica and also consider column grouping.

Column Grouped Layouts. Given a query workload, finding the optimal column grouping (or vertical partitioning) is a NP-hard problem [109]. Thus, most of the works on vertical partitioning, including the initial approach [95], Data Morphing [64], and HYRISE [60] focus on heuristics to improve the runtime complexity but do not consider the quality of column grouping. [9] considers the interestingness (CG-Cost) of candidate column groups and prunes the ones below a certain threshold. However, CG-Cost has several problems (discussed in Section 4.3.1). Our interestingness measure significantly improves upon CG-Cost, thus enabling our column grouping algorithm to produce better quality results. Furthermore, previous column grouping algorithms produce a single best layout for the entire workload. However, in a parallel data processing system having inherent data replication, different replicas could be mapped to different layouts. Our column grouping algorithm makes this possible by producing a set of column-grouped layouts, each of which are best suited for a different subset of the workload. With this, we can later route an incoming query to a more specialized layout.

DBMS Stores with MapReduce. HadoopDB [5] replaces the HDFS storage in MapReduce with a database. Thus, the data is now in the DBMS data layout (row layout for row-oriented DBMS, column layout for column-oriented DBMS). However, this involves severe changes to the Hadoop execution framework. Similarly, another approache analyzes the map functions and translate them into SQL queries to be run on a database [77]. These approaches are orthogonal to our *Trojan* philosophy: affect Hadoop from inside in a non-invasive manner by injecting our technology at the right places through UDFs only [49].

Multi-column Indexes. Multi-column indexes, e.g. in [29], are combined indexes over multiple columns and thus store multiple columns together (column groups). However, multi-column indexes are additional storage structures created on top of an underlying table, whereas Trojan Layouts are the physical representations of the underlying table itself. Furthermore, multi-column indexes consider only the indexable attributes, i.e. attributes appearing in selection and join predicates, while Trojan Layouts considers all accessed attributes. As a result, multi-column indexes reduce the number of rows to access, whereas Trojan Layouts reduces the number of columns to access. Thus, multi-column indexes are orthogonal to Trojan Layouts.

Materialized Views. Materialized views are persisted copies of arbitrary query results [35]. One could think of using materialized view design wizards from relational databases, e.g. SQLServer, to achieve some of the optimizations presented in this chapter. However, in order to apply these tools to Hadoop MapReduce, we would need

to extend them significantly. For instance, we need to take into account the default data replication. Still, materialized views are additional storage structures and need additional storage space as well as creation and query techniques. Trojan Layouts, on the other hand, keeps the same data size and reuses the existing Hadoop MapReduce pipeline.

4.7 Conclusion

MapReduce suffers from very slow execution times in some analytical tasks compared to DBMSs. One of the reasons for this is that MapReduce processes input data blocks in a strictly row-oriented fashion, which leads to full scans of the input data [101].

In this chapter we proposed Trojan Layouts, a new data layout that organizes data inside data blocks according to the incoming workload. We followed the PAX principle in that we did not change the outside view of data. However, we considerably depart from PAX as we: (i) might co-locate attributes together according to query workloads, (ii) may use different Trojan Layouts for different data block replicas, and (iii) may, in a special case, mimic fractured mirrors: having the best from both PAX and Row Layouts. We implemented our algorithms on top of HDFS 0.20.3. A salient feature of using per-replica Trojan Layouts is that we can schedule incoming jobs to data block replicas having the best Trojan Layout. With Trojan Layouts, we have the flexibility to choose a different storage view for each data block replica in Hadoop MapReduce. As a result, we can now have efficient column-oriented as well as arbitrary partial-projection-oriented processing, in addition to standard row-oriented processing in Hadoop MapReduce.

We experimentally evaluated Trojan Layouts using three real-world benchmarks: TPC-H, SSB, and SDSS, and compared its effectiveness against Hadoop using Row (Hadoop-Row) and PAX (Hadoop-PAX) layouts. The results demonstrated that our approach significantly outperforms both Hadoop-Row and Hadoop-PAX in all three benchmarks: up to a factor of 4.8 for Hadoop-Row (3.3 times faster on average and 1.1 times slower in the worst case); and up to a factor of 3.5 for Hadoop-PAX (1.6 times faster on average and no improvement in the worst case). Figure 4.11 illustrates how the experimental runtimes of queries Q1 to Q8 varies with the number of referenced attributes for TPC-H `Customer` table. In particular, the results showed that the per-

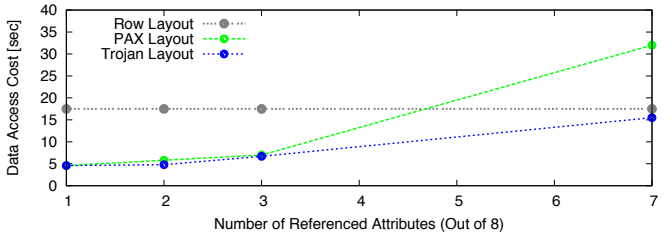


Figure 4.11: Simulation Validation for TPC-H `Customer`

formance of Hadoop-PAX decreases quickly as the number of referenced attributes increases. This is not the case for Trojan Layouts. In other words, our experimental results support the simulation results we presented in the introduction of this chapter (see Figure 4.2).

As future work, we plan to adapt Trojan Layouts to changes in the workload. Several strategies, such as piggy backing into ongoing MapReduce jobs, can be employed and need to be investigated in more detail.

Chapter 5

Column-oriented Storage for Relational Data Management

Column stores are becoming popular with modern enterprises. However, database vendors offer column stores as a different database product all together. As a result there is a all-or-none situation for column store features. To bridge the gap, a recent effort introduced column store functionality in SQL server (row store) by making deep seated changes in the database system. However, this approach is expensive in terms of time and effort. In addition, it is limited to SQL server. In this chapter, we present Trojan Columns, a novel technique for injecting column store functionality into a given row-oriented commercial database system. Trojan Columns does not change the source code of the database system. Instead, it uses UDFs as a pluggable storage layer to write and read data. Furthermore, Trojan Columns is transparent to the user, i.e. the user does not need to change his schema and his queries remain almost unchanged. We demonstrate Trojan Columns on a row-oriented commercial database DBMS X, a closed source top notch database system. We show experimental results from TPC-H benchmarks. Our results reveal that Trojan Columns work very well with non-nested and high selectivity queries. From our results on non-nested and high selectivity TPC-H queries, we see that Trojan Columns can improve the performance of DBMS X by a factor of 9 in the best case, a factor of 3.9 on average, and a factor of 2.5 in the worst case, without changing the source code and with minimal user effort. On the other hand, for non-nested and low selectivity TPC-H queries, our results showed that DBMS X slows down by a factor of 2 on average with Trojan Columns. We also ran experiments on simplified TPC-H queries, as used in popular papers like [18, 116], and our results show that Trojan Columns can improve the performance of DBMS X by a factor of 13 in the best case, a factor of 5.2 on average, and a factor of 3.9 slowdown in the worst case.

5.1 Introduction

5.1.1 Background

Row stores like Oracle and DB2 as well as column stores like Vertica, MonetDB, have, in recent times, emerged as two major technologies in the commercial database market. However, database vendors typically offer different database products for row and column

stores respectively. This is a huge problem for customers since they have to make a strategic decision on which (one or more) database products to use. Alternatively, in recent times, several people have argued for the superiority of column stores over row stores [1, 3, 66, 116]. As a result, several enterprise customers might choose to migrate to a column store. However, this is an expensive process. It involves new licensing costs, additional training for the administrators and developers, and migration effort from the old to the new database product. For a customer, such change of faith from one database product to another is a once-in-while process.

5.1.2 Problem

A recent approach integrated column store indexes into SQL Server [86]. In that approach, column store indexes store segments of columns as blobs in the standard row store table. This, however, requires deep changes in all the layers of the database system, including query processing and data storage enhancements. As a result, this is a considerable effort for the vendors, and meanwhile the users have to wait for the next product release. The above approach certainly helps SQL Server users, since they can still use the rich DBMS features as well as the sophisticated query optimizer. But what about the users of other database products, e.g. IBM DB2, Oracle? It is not clear whether they can or are willing to emulate column stores as well. Users of these database products will certainly ask this question.

In summary, the research problem we explore in this chapter is as follows. For any proprietary closed source database product, is it possible to introduce column store technology in it? All this *without* having access to the source code of the database product.

5.1.3 Research Challenges

The above discussion brings several research challenges to the forefront, which we phrase as hard hitting questions below:

(1.) Should the enterprise customers continue living with *all-or-none* situation for row and column store features? Do they really have to make a binary decision between row and column stores or can there be an alternative approach?
(2.) Can the enterprise customers try out a column store in their current database product before actually migrating to one? What would be the performance gain when emulating a column store?
(3.) Do the users have to rely on database vendors to integrate column store functionality within the database system? Do they really have to wait till the next product release or is it possible for users to start using column store functionality on demand?
(4.) Is it possible to emulate a column store in a given commercial row store database system? Can we have a generic approach to simply *inject* column store functionality into a given database system?

5.1.4 Our Idea

Our idea is to use User Defined Functions (UDFs) as an access layer for data storage and retrieval. To do so, we create and install certain UDFs within the database system and exploit them whenever we need to store or access data. The data is actually stored in a compressed column-oriented fashion on disk. But the UDFs translate it into the row layout for the query processor. This means we trick the database into believing that the data is still stored in row fashion, even though it is not. It is like adding trojans (but good ones) to the existing database in order to significantly boost its performance, yet without trespassing the propriety code base. Finally, note that our approach is very different from the two extremes of data stores: either having a different product for different stores or doing deep seated changes in the database product. We do neither of these, but still gain performance significantly. The major benefits of our approach are as follows:

(1.) Injects column store functionality into existing closed source database products e.g. IBM DB2, Oracle.
(2.) Logical view of data remains unchanged for the outside user; instead, the changes are transparently injected inside the DBMS.
(3.) Does not invade or make heavy changes in the system; rather, uses lightweight UDFs to store and access the data.
(4.) Just fixes the storage layer (row, column, or even column-grouped layouts) by inserting appropriate UDFs.
(5.) Reuses the query optimizer (at least partially) and therefore no need to re-implement state-of-the-art database technologies.
(6.) User queries remain (almost) unchanged.

5.1.5 Contributions

Our main contributions are as follows:

(1.) We explore database UDFs as a novel way to supporting different data layouts in a database system. The UDF approach introduces the novel functionality without touching the source code of the database system, and yet it is completely transparent to the outside user. (Section 5.2)
(2.) We present *Trojan Columns*: a radically different technique to inject column store functionality into a given database system. Trojan Columns masks the database storage layer and translates back and forth from the user row-view to the physical column-view of data. All the while, the user view remains unchanged. (Section 5.3)
(3.) We present techniques to query Trojan Columns. We show how to push down one or more operators in the query tree to the UDFs. We describe how to rewrite the user queries in order to use the UDFs for data access. (Section 5.4)
(4.) We present details for implementing Trojan Columns in DBMS X, a commercial closed source database system. Additionally, we also explore and contrast stored procedures as an alternative to UDFs for Trojan Columns. (Section 5.5)
(5.) We present experimental results from DBMS X over TPC-H datasets and queries. We compare our approach to C-Tables [18], a recent approach of emulating columns stores in row stores, and demonstrate how Trojan Columns outperforms C-Tables. We

investigate the pros and cons of Trojan Columns by varying several query and storage parameters. We also compare Trojan Columns with a native open source column-oriented storage implementation in PostgreSQL. We discuss the benefits and the costs of native column-oriented storage over Trojan Columns. (Section 5.6)

5.2 The UDF Storage Layer

5.2.1 Background

User defined functions (UDFs) provide a mechanism for a user to inject custom behavior for data processing. In databases, a UDF is a piece of code written by a user to add or extend the database system functionality. Typically database systems support three kinds of UDFs based on their return types: (i) scalar value returning UDFs, e.g. a UDF to compute the total number of vowels in a given string, (ii) row returning UDFs, e.g. a UDF to compute the individual counts of each vowel in a given string, and (iii) table (of rows) returning UDFs, e.g. a UDF to compute the individual vowel counts in 20 random strings. The table returning UDFs are of particular interest to us and we exploit them to customize the storage layer in a database. Below we discuss why.

5.2.2 Why UDFs as the Storage Layer?

Following are the reasons why UDFs are well suited to be used as a storage layer for databases.

Plug-and-play. Current database products have hard-coded assumptions about the data storage: a given database system assumes a given fixed data store. This makes changing the storage layer all the more difficult, since we need to modify almost all other components of the database system, including query parser, optimizer, and executor. Fortunately, UDFs provide plug-and-play functionality, i.e. we can inject the new functionality without making any deep seated changes in the database system. The user simply creates the UDFs once and can thereafter use them in his SQL statements over and over again.

Tight integration. Other approaches, such as full vertical partitioning and C-Tables [18], mimic column stores in row stores at the schema level. Schema level approaches are loosely integrated within the database system and they incur redundant joins at query time. Hence, as we will show, schema level approaches are highly performance inefficient. On the other hand, even though UDFs are plug-and-play, they are tightly integrated within the database system. They are executed at the server itself within the server process. This avoids any networks costs and the execution is highly efficient. Furthermore, the databases exposes several core functionalities, which can be reused, to the UDFs. This allows for even more powerful and efficient use of UDFs. The tight integration allows the UDF storage layer to be performance efficient.

Universal applicability. One of the most important advantages of UDFs is their universal applicability. Almost every database product, e.g. IBM DB2, Microsoft SQL Server, Oracle, provides interfaces for UDFs. These databases not only allow the users to write UDFs in standard SQL, but also provide rich programming language support.

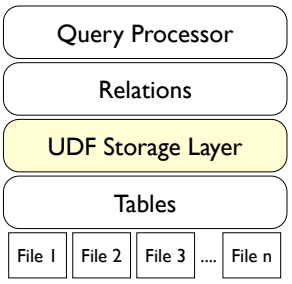


FIGURE 5.1: The UDF Storage Layer

This means that the users can code the UDFs in languages such as C and Java. Thus, an approach using UDFs is generic enough for widespread applicability.

SQL nesting. Finally, the UDFs thus created can be easily nested in SQL queries. This is in contrast to stored procedures which need a CALL statement to be invoked. At execution time, whenever the query executor finds a UDF in the query tree, it simply loads and executes the corresponding code. This effectively means that a part of the functionality in a given SQL query is provided by the UDF. As a consequence, we do not need to change the entire query. Instead, we just add the functionality wherever required. This is in contrast to both full vertical partitioning as well as RowCol, where the user is forced to change his schema, incur complete query rewriting, and introduce additional joins.

5.2.3 Mapping Relations to Tables

Let us now see how we map relations to tables when using UDF as a storage layer in databases. Typically, in a row-oriented database system, the logical relation is mapped to a physical row-oriented table, i.e. there is a one-to-one mapping. However, with the *UDF storage layer*, as shown in Figure 5.1, the UDFs serve as the mapping between the logical and the physical representation of data, i.e. the user can have any arbitrary physical representation of his logical relation, similar to *storage views* in OctopusDB [48, 78]. Even though the UDF storage layer allows for any physical representation of data, internally it still uses (exploits) the row store tables of the database to store data on disk. We call these internal tables as *physical tables*. Note that in contrast to vertical partitioning, the user, and in contrast to columns stores, the database internals (query parser, optimizer, executor) still assume that the data is stored in the standard row representation. Only at the data access layer, the UDFs intervene to instead map the data to a different physical representation.

5.3 Trojan Columns

In this section, we present Trojan Columns: a novel way of *injecting* column store functionality in a given row-oriented database system. Trojan Columns uses the UDF storage layer to translate the logical row view from the user to the physical column view on disk. The core philosophy of Trojan Columns is very similar to Trojan Index and Joins in Hadoop++ [49, 80]: affect the changes from inside without changing the

source code of the system. In the following, we describe how to create and query Trojan Columns, as well as how to handle inserts and updates.

5.3.1 Data Storage

Trojan Columns maps logical relations to physical tables as follows. First, we horizontally partition a given relation T into segments. Then, we store each attribute in a given segment as a separate BLOB (binary large object) in physical table T_trojan(segment_ID, attribute_ID, blob_data). To illustrate, consider the following entries of a Customer relation.

Customer			
ID	name	phone	market_segment
1	smith	2134	automobile
2	john	3425	household
3	kim	6756	furniture
4	joe	9878	building
5	mark	4312	building
6	steve	2435	automobile
7	jim	5766	household
8	ian	8789	household

Assuming a segment size of 4, i.e. 4 entries of each column are mapped to a different row, the Trojan Columns leads to the following Customer_trojan table:

Customer_trojan		
segment_ID	attribute_ID	blob_data
1	name	smith, john, kim, joe
1	phone	2134, 3425, 6756, 9878
1	market_segment	automobile, household, furniture, building
2	name	mark, steve, jim, ian
2	phone	4312, 2435, 5766, 8789
2	market_segment	building, automobile, household, household

We store each entry in the blob_data column of the above table as a BLOB, thus mimicking a column-oriented storage. Experimentally, we found much bigger segment sizes, e.g. 10M rows, to be more suitable. The data storage idea for Trojan Columns is inspired by SQL Server Column Indexes [86]. However, in practice, Trojan Columns is radically different from Column Indexes in several ways.

First, Trojan Columns uses UDFs to store the blob data instead of native SQL support in case of Column Indexes. This not only makes Trojan Columns plug-and-play, but also allows us to customize the blob storage to user applications. For example, we might prefer light weight compression (e.g. RLE) for read-intensive application and higher compression ratio (e.g. Huffman) for archive application. Or, we could simply let the database system apply the default compression method for blobs, e.g. TOAST compression in PostgreSQL. Likewise we might choose to sort the data within or even across the blobs in any way, e.g. sort on attribute IDs. Essentially, we have full control and flexibility to decide how the blobs must reside on disk.

Second, Trojan Columns is quite different from SQL Server Column Indexes not only in data storage but also in data access. Similar to data storage, the access method is plugged into a given database, instead of making deep changes in the data access layer itself. Again, we have full control to decide, based on user application, how to access

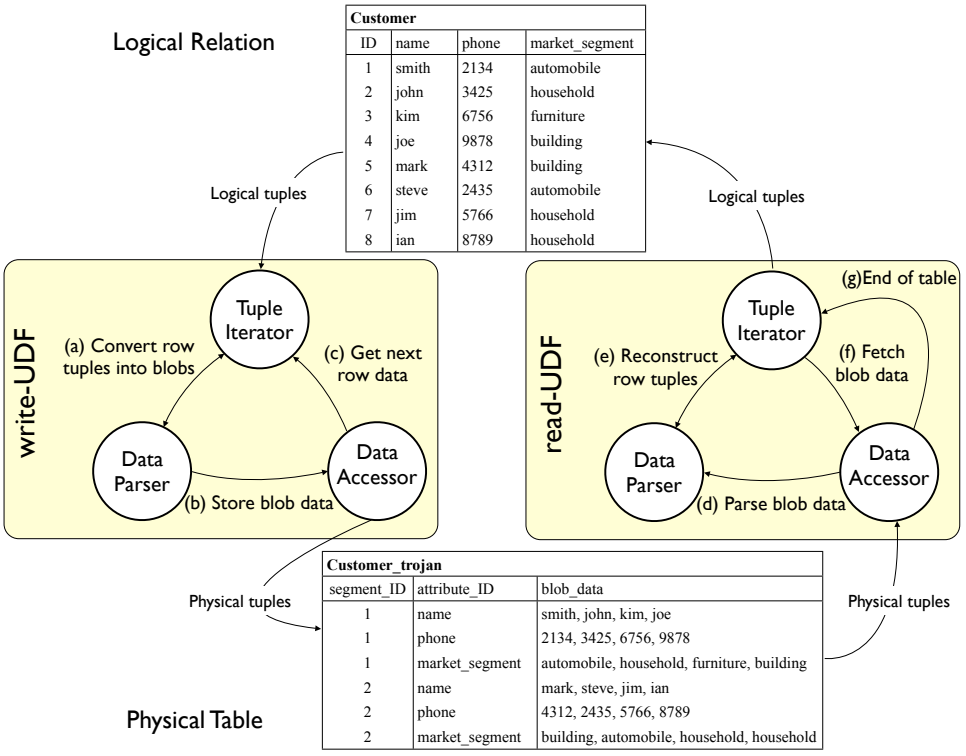


FIGURE 5.2: Read and Write UDFs for Trojan Columns

the data. For example, for an airline company, we may choose to look for all possible connections only for premium customers.

Third, Trojan Columns uses standard database tables to store the blob, instead of a new index type in case of Column Indexes. The database uses its own physical storage mechanism to persist the blob table on disk. In other words, Trojan Columns simply provides a mechanism to decouple logical representation of relations from their physical implementation, a.k.a *physical data independence*, which, unfortunately, still remains a myth in several modern databases [127].

Finally, with Trojan Columns the database system is entirely agnostic of the column store functionality injected within, i.e. no new SQL keywords, or data types, or any entries in the system catalog have to be added. The consequence is that Trojan Columns is *database product independent*, i.e. they can be *plugged* into any existing database product.

The left side of Figure 5.2 shows the *write-UDF* to upload customer relation into Trojan Columns. The write-UDF in the UDF storage layer has three components: tuple iterator, data parser, data accessor. Let's walk through each of them. The tuple iterator iterates over the input logical tuples. The tuple iterator passes the logical tuples to the data parser (step (a) in Figure 5.2) to convert them into Trojan Column representation. The data parser passes the Trojan Column tuples to the data accessor (step (b) in Figure 5.2), which stores them on disk. Finally, the data accessor signals the tuple iterator to read the next set of logical tuples (step (c) in Figure 5.2).

5.3.2 Data Access

Trojan Columns is stored without any native support from the database system. Therefore, Trojan Columns must be accessed in a way such that the database system remains oblivious to it. To do so, we use *table returning UDFs* to access Trojan Columns. The right side of Figure 5.2 shows such a *read-UDF*. Again, the read-UDF has three components: tuple iterator, data parser, data accessor. The data accessor accesses the required blobs (if any) from the physical table and passes it on to the data parser (step (d) in Figure 5.2). The data parser reconstructs the logical tuple from the physical blobs (step (e) in Figure 5.2). Now in each UDF call, the tuple iterator checks if there are more tuples to be passed from the data parser to the user. If yes, it simple passes the next tuple to the user and returns. Otherwise, it asks the data accessor to access the next blobs from the physical table (step (f) in Figure 5.2). When all blobs have been accessed by the data accessor, i.e. no more logical tuples to output, the UDF returns end-of-table (step (g) in Figure 5.2). Essentially, the table returning UDF acts as a data access path.

5.3.3 Handling Inserts and Updates

By design, column store functionality is suited for analytical workloads, which are typically read-only. Still, Trojan Columns handles inserts as follows: (1) maintain a temporal row table to store the newly inserted records, (2) create an insert trigger to keep a count of the number of rows inserted in the temporal row store, and (3) use write-UDF to create and insert blobs into the physical table, once segment size number of rows have been inserted into the standard row table. At query time, the read-UDF must also read the temporal row table for newly inserted rows.

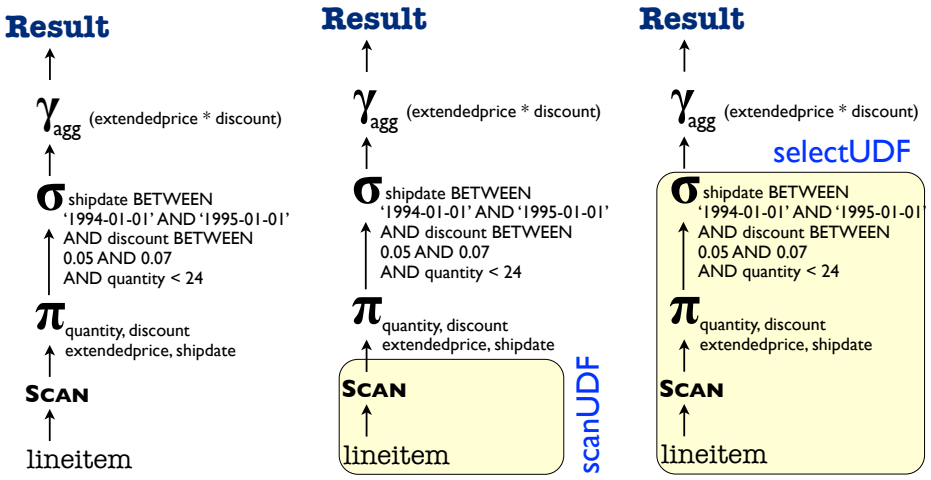
Note that the above strategy of a trigger and function call for every insert might be too expensive for insert intensive applications. An alternate strategy could be to periodically bulk load Trojan Columns from the base table. To handle updates, the update-UDF first needs to determine the segment in which the update must be applied. Thereafter, the update-UDF must read the affected blobs in that segment and write them back.

5.4 Query Processing

In the previous section, we described how to create Trojan Columns. In this section, we describe how we process queries using Trojan Columns. Below, we first describe operator pushdown as a technique to process Trojan Columns, and then we describe how to rewrite the user queries.

5.4.1 Operator Pushdown

The core idea of querying Trojan Columns is to push a part of the query tree down to the UDF. This means that a part of the query is processed by the UDF while the remaining query is still processed by the standard database query executor. Let's consider query 6 from the TPC-H benchmark [126] as a running example below. Figure 5.3(a) shows the logical query plan for this query. Below, let's see how we can push down one or more operators in query 6 to a UDF.



(a) Standard plan (b) Scan pushdown (c) Select pushdown

FIGURE 5.3: Standard and UDF query plans for TPC-H Query 6.

5.4.1.1 Scan Pushdown

First of all, we need to push down the scan operator to the UDF. This is because we need to interpret Trojan Columns correctly (and differently) at the leaf level. Suppose that `lineitem` table in query 6 is stored as Trojan Columns. Figure 5.3(b) shows the query plan with the UDF. As shown in the figure, the UDF now figures out which physical table to read (the blob and not the row representation) for `lineitem` table. Also, the UDF is responsible for interpreting the physical table, reconstructing the logical `lineitem` tuples, and passing them on to the upper part of the query tree.

Algorithm 5.1 shows the pseudocode of scan UDF. The UDF takes the table name to scan as input and outputs one row tuple in each call. The UDF first checks the type of function call (Line 1). All databases have roughly three call types for table retuning UDFs: (1) *init* – when the UDF is called for the first time, (2) *next* – when the UDF is called second subsequent times, and (3) *end* – when the UDF is called after the last row has been received. If the UDF call type is INIT, then the current segment points to the first segment in the blob table, all blobs in the first segment are fetched, and the current blob offset is reset (Lines 2–6). If the UDF call type is NEXT (Line 7), i.e. the database system is asking for the next tuple to process, the scan UDF checks whether it is already at the end of current blob (Line 8). If that is the case, the scan UDF moves the segment pointer and fetches the corresponding blobs (Lines 9–10). If the current blobs point to NULL (Line 12), i.e. no more blobs are available, the scan UDF returns NULL, indicating end of data. If there is still more data, the scan UDF reconstructs the next logical tuple from the blobs (Line 15), moves the blob offset forward (Line 16), and returns the logical tuple to the database system (Line 17). If the UDF call type is END, then the scan UDF simply closes the connections for accessing physical tables and frees the memory (Lines 19–22).

Algorithm 5.1: `scanUDF`

Input : tableName
Output: Logical tuple of table *tableName*

```
1  switch UDFCallType do
2      case INIT
3          currentSegment = 1;
4          currentBlobs = fetchBlobs(currentSegment);
5          currentBlobOffset = 0;
6      end
7      case NEXT
8          if currentBlobOffset == SEGMENT_SIZE then
9              currentSegment++;
10             currentBlobs = fetchBlobs(currentSegment);
11         end
12         if currentBlobs==NULL then
13             return NULL ;
14         end
15         tuple = reconstructNextTuple(currentBlobs);
16         currentBlobOffset++;
17         return tuple;
18     end
19     case END
20         closeConnections();
21         freeMemory();
22     end
23 end
```

5.4.1.2 Projection Pushdown

Along with the scan, we can also push down the projection operator to the UDF, i.e. pass the projected attributes as parameters to the UDF. Therefore, the input in Algorithm 5.1 now changes as follows:

Input : tableName, projectionAttributes

The UDF returns only the projected attributes. Since the UDF return type is still the complete row, all other attribute values are set to NULL. A consequence of pushing projection down to the UDF is that the UDF now needs to fetch the blobs of only the projected attributes. Thus, Algorithm 5.1 now fetches the blobs as follows:

currentBlobs = `fetchBlobs`(*currentSegment, projectionAttributes*);

Fetching blobs of only the projected attributes saves considerable I/O cost and improves query performance. Except for the above two changes, Algorithm 5.1 remains unchanged with projection pushdown.

5.4.1.3 Selection Pushdown

To push the selection down, we simply pass the selection predicate to the UDF, as shown in Figure 5.3(c). The input to Algorithm 5.1 is now as follows:

Input : tableName, projectionAttributes, selectionPredicate

The UDF is now responsible for evaluating the select predicate on each of the incoming tuple. To do so, the UDF now only fetches the selection attributes first:

currentBlobs = `fetchBlobs`(*currentSegment, selectionAttributes*);

The UDF maintains separate pointers to projection attribute segments and blobs. These pointers are initialized when the UDF call type is INIT:

projSegment = 0;
projBlobs = `NULL`;

Then, before returning the tuple, the UDF evaluates the selection predicate. If the predicates satisfy then the UDF fetches the projection attribute blobs, if needed (currentProjSegment < currentSegment), and returns a tuple of the projected attributes. If the selection predicates do not satisfy, then the UDF inspects the next selection attribute values. This continues until either a qualifying tuple if found or end of data is reached. In summary, Line 17 of Algorithm 5.1 is replaced with the following logic:

if (`evalSelection`(*tuple, selectionPredicate*))
  if (projSegment < currentSegment)
    projBlobs = `fetchBlobs` (*projSegment, projectionAttributes*);
  return `reconstructNextTuple` (*projBlobs*);
else
  goto NEXT;
end

Pushing down selection to the UDF has two advantages: (1) the number of UDF output tuples, and consequently the number of UDF calls are reduced, and (2) we can perform late materialization by fetching projection attributes only for segments having at least one tuple qualifying the selection predicates. The first advantage saves the overhead in each UDF call, while the second advantage saves I/O for projection attributes.

5.4.1.4 Aggregation Pushdown

We can even push down the aggregates (and group by) to the UDF. However, this will involve one major change in Algorithm 5.1. The UDF must do the grouping and aggregation *before* outputting any of the tuples. This means that the UDF must precompute the results when the UDF call type is INIT, and simply return them when the UDF call type is NEXT. Algorithm 5.2 shows the pseudocode of aggregation UDF. We can see that when the UDF call type is INIT (Line 2), the UDF fetched the required blobs and computes the aggregate (Lines 3–14). It also creates an iterator to iterate over the resulting output (Line 15). When the UDF call type is NEXT (Line 17), the UDF checks if there are more results in the output iterator (Line 18). If yes, the UDF returns the next precomputed result. Otherwise, the UDF returns NULL, indicating end of data. The cleanup when UDF call type is END remains unchanged. The major benefit of pushing aggregation down the UDF is to dramatically reduce the number of UDF calls.

Algorithm 5.2: `aggregateUDF`

Input : tableName, `aggregateFunc`
Output: Logical tuple of table *tableName*

```
1  switch UDFCallType do
2      case INIT
3          currentSegment = 1;
4          currentBlobs = fetchBlobs(currentSegment);
5          output = NULL ;
6          while currentBlobs ! = NULL do
7              currentBlobOffset = 0;
8              while currentBlobOffset < SEGMENT_SIZE do
9                  tuple = reconstructNextTuple(currentBlobs);
10                 aggregateFunc(tuple, output);
11                 currentBlobOffset++;
12             end
13             currentBlobs = fetchBlobs(currentSegment++);
14         end
15         outputIterator = iterator(output);
16     end
17     case NEXT
18         if outputIterator.hasNext() then
19             return outputIterator.next();
20         else
21             return NULL ;
22         end
23     end
24     case END
25         closeConnections();
26         freeMemory();
27     end
28 end
```

5.4.1.5 Dealing with Join Queries

So far we have considered single table queries, i.e. no join conditions. Now let us see how joins are processed in the presence of Trojan Columns. For queries having join conditions, we simply push down the scan, selection, and projection operators to the UDF and let the database do the join. This works well because the output of UDF can be processed by the database query executor. Figure 5.4 shows the UDF query plan for TPC-H query 14. From the figure we see that the `lineitem` leaf is pushed inside the UDF, while the join is still performed outside. Also note that the query plan in Figure 5.4 accesses `part` table using the standard database access method. This is because `part` is much smaller table and it does not pay off to use a UDF for it. Thus, we see that UDFs can be seamlessly integrated into the query pipeline. This holds true even for nested queries, e.g. TPC-H query 8.

Alternatively, instead of letting the database executor process the join, one could think of even pushing down the join to the UDF. The UDF would then have to access two physical tables and join them based on the join condition. The advantage would be that we could have even lesser output tuples (depending on join selectivity). However, the problem is that we will need to recode the physical join operators as well as the optimizer logic to pick the physical join operator. Thus, we see the pros and cons of pushing too many operators down the UDF. Exploring these in more detail will be part of a future work.

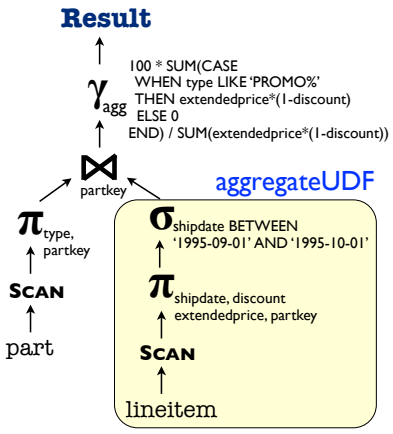


FIGURE 5.4: Example UDF query plan for TPC-H query 14.

5.4.1.6 Where does operator pushdown lead to?

In the extreme case, we can push down the entire SQL query, i.e. all query operators, down to the UDF. However, this means that the UDF is now responsible for deciding how to execute a given query. In other words, the UDF must take care of query optimization as well as execution, making it a micro-kernel for processing SQL queries. The consequence is that the user must now recode all physical operators, cost models, as well as the optimization logic. Obviously, this is very hard to do. Therefore, it is important to strike the right balance when pushing down to the UDF. While too much is nasty, too less kills performance. A general practice could be to push only the leaves to the UDF and let the database handle the joins, unless they could be rewritten to selections.

5.4.2 Query Rewriting

Typically, there are two extremes of query rewriting with column stores: (1) complete query rewriting, due a complete change in schema, e.g. C-Table [18] and standard vertical partitioning, and (2) no query rewriting, if the column stores are natively implemented, i.e. no schema change at all. Our approach finds the middle ground. At bare minimum, we only rewrite the data access paths in the query, while keeping the rest of the query unchanged. This means that we can simply specify a data access UDF, for accessing Trojan Columns, in the `FROM` clause of the SQL statement. Note that we can use Trojan Columns for any subquery, thereby having layouts (row or column) on a per-table basis. In the following, let us see how to rewrite the SQL statements when using Trojan Columns. Consider TPC-H query 6:

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
  lineitem
WHERE
  l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
  AND l_discount BETWEEN 0.05 AND 0.07
  AND l_quantity < 24;
```

Assume we have a scan UDF `scanUDF`(*table_name*) to read the blob data and convert them into logical tuples of table *table_name*. The query with scan pushdown is as follows:

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
   scanUDF('lineitem')
WHERE
  l_shipdate >= '1994-01-01' AND l_shipdate < '1995-01-01'
  AND l_discount BETWEEN 0.05 AND 0.07
  AND l_quantity < 24;
```

If we further push down the selection and projections, using UDF `selectUDF`, query 6 is now as follows:

```
SELECT
  SUM(l_extendedprice*l_discount) AS revenue
FROM
   selectUDF(
      'lineitem',
      'quantity,discount,extended price,price',
      'l_discount in (0.05,0.07)
         AND  l_shipdate in (1994-01-01,1995-01-01)
         AND l_quantity < 24'
      )
```

In the extreme case, if we push everything inside the UDF, the query will simply be:

```
SELECT * FROM everythingInUDF(....);
```

Note that we expect the view mechanism in standard databases to take care of automatically rewriting the incoming user queries to the ones using the UDF access path. Further study on this will be a part of future work.

5.5 Implementation Details

In the previous section we presented the general idea of Trojan Columns. In this section, we discuss the implementation details for Trojan Columns in DBMS X.

5.5.1 DBMS X Table UDF Interface

Interface. The table UDF interface in DBMS X is a C interface to write UDFs. In general, the signature of UDF to query the Trojan Columns (with `n` attributes) looks as follows.

```
void SQL_API_FN udf_name (
    SQLUDF_CHAR* tableName,
    SQLUDF_CHAR* projectionStr,
    SQLUDF_CHAR* selectionStr,
    SQLUDF_CHAR* groupbyStr,
    SQLUDF_CHAR* logFile,
    SQLUDF_RESULTTYPE1* RESULT1,
    SQLUDF_RESULTTYPE2* RESULT2,
    .
    .
    SQLUDF_RESULTTYPEn* RESULTn,
    SQLUDF_SMALLINT* RESULT1_IND,
    SQLUDF_SMALLINT* RESULT2_IND,
    .
    .
    SQLUDF_SMALLINT* RESULTn_IND,
    SQLUDF_TRAIL_ARGS_ALL           // current UDF state
)
```

The first 5 arguments are actually passed to the function by the user, when invoking the UDF. The remaining arguments are not passed to the function by the user. They are the result variables to which we write the result row. For each call, DBMS X maps the memory of the result table to these pointers.

Scratch Pad. There is a specific memory area provided by DBMS X, which is called the scratchpad. This memory area is kept alive during the individual calls to the UDF and can be used to maintain a state between the calls. In our approach, all main data structures are held inside of the scratchpad, since they are used in several phases.

Multi-threading. Apart from the main (output) thread, we maintain separate threads for I/O and processing. This means that as soon as one of the buffers is free, the next segment is already loaded into it. For each loaded attribute that is compressed, we decompress it in a separate processing thread. The processing thread also performs the selection and produces a selection vector, indicating which tuples qualify. Until the processing thread performs the decompression and selection, the main (output) thread waits. When the processing thread is finished, the main thread inspects the selection vector. If a selected row is found in the vector, then the row data is immediately returned. We keep the current position of the selection vector in the scratchpad to output the next row in the subsequent UDF call. There are several wait/notify constructs necessary to coordinate the segment loading with double buffering, selection and outputting. In between the function calls, we keep alive the processing thread (together with the decompression threads) in the scratchpad.

Communication with DBMS X. To communicate with DBMS X from inside the UDFs, we use embedded SQL in C (SQC). As a result, the UDF program is not written in pure C, but in a mixture of C and SQL. For example, it is possible to place statements like "EXEC SQL FETCH ... INTO ..." within the functions. These statements are used to query the Trojan Columns blobs from the database. Anytime we query the database with embedded SQL statements, we have to store the result of this query inside host variables, which act as a connection bridge between DBMS X and our program. Host variables have to be declared in a separate area and they support special datatypes, corresponding to SQL types. Note that it is not possible to modify the database using UDFs in DBMS X; only querying is allowed. Before installing, the SQC file must be precompiled to create a standard pure C file. This file contains calls to internal DBMS X functions that represents the SQL equivalents. We can then compile this file using a standard C compiler and link it to the database.

Installation. Finally, we install the UDF to query Trojan Columns (with `n` attributes) in DBMS X as follows.

```
CREATE FUNCTION udf_name(
    tableName VARCHAR(128),
    attributeStr VARCHAR(1000),
    selectionStr VARCHAR(1000),
    groupbyStr VARCHAR(1000),
    timeFile VARCHAR(1000)
  )
  RETURNS TABLE (
    attribute_1 TYPE1,
    attribute_2 TYPE2,
    .
    .
    attribute_n TYPEn,
  )
  SPECIFIC udf_name              // internal UDF name
```

```
  EXTERNAL NAME 'ext_udf_name'    // UDF program name
  LANGUAGE C
  PARAMETER STYLE DBMS_X_SQL
  NOT DETERMINISTIC
  FENCED NOT THREADSAFE     // different address space
  READS SQL DATA
  NO EXTERNAL ACTION
  SCRATCHPAD 10000         // size in bytes
  FINAL CALL               // final call phase executed
  DISALLOW PARALLEL;       // single database partition
```

5.5.2 DBMS X Call Level Interface (CLI)

The table UDF interface in DBMS X has the limitation that the exact schema of rows to return must be fixed at compile time. To overcome this, we developed a second approach based on the DBMS X Call Level Interface (CLI) and Stored Procedures (SP). CLI is a C/C++ interface that translates queries and data between an application and a database. It allows us to create the queries for accessing the data dynamically at runtime. Furthermore, as a CLI exists for many DBMSs, the routine is easily portable. In DBMS X, the entry function for stored procedures looks as follows.

```
SQL_API_RC SQL_API_FN udf_name(
    CHAR *tableName,
    CHAR *projectionStr,
    CHAR *selectionStr,
    CHAR *logFile,
    SQLINT16 *tableName_IND,
    SQLINT16 *projectionStr_IND,
    SQLINT16 *selectionStr_IND,
    SQLINT16 *logFile_IND,
    SQLUDF_TRAIL_ARGS          // stored procedure state
  )
```

Note that in contrast to table UDF interface, we do not need to specify the return type for stored procedures. However, they are not able to return their results directly. Instead, in contrast to UDFs, they allow write accesses to the database. We exploit this to store the results of a query into a temporary table in the database. There it can be queried by normal SQL or used as an intermediate result for further computation. This means an existing user query cannot be translated one to one, but it must be split into two calls: (1) a stored procedure call, and (2) a query for post-processing and returning the results to the user. For example, for TPC-H query 14 the rewritten query first calls the stored procedure to project and select the relevant data from `lineitem`. The final query then joins `part` table with the result table, which is several orders of magnitude smaller than the original `lineitem` table. So the final query is extremely cheap and the overall costs are dominated by the stored procedure. The main advantage of CLI -SP is that because of its highly dynamic interfaces, we don't need to recompile the routine for every query or table. We only have a single stored procedure that can be used in arbitrary queries without more effort than rewriting the query. We install the stored procedure for querying Trojan Columns as follows.

```
CREATE PROCEDURE sp_name(
    IN name VARCHAR(128),
    IN attr VARCHAR(512),
    IN selPred VARCHAR(512),
    IN logfile VARCHAR(128)
  )
  SPECIFIC sp_name
```

```
DYNAMIC RESULT SETS 0         // do not return results
NOT DETERMINISTIC
LANGUAGE C
PARAMETER STYLE SQL
FENCED NOT THREADSAFE
MODIFIES SQL DATA
PROGRAM TYPE SUB              // used as library
EXTERNAL NAME 'ext_sp_name'   // procedure program name
```

5.6 Experiments

We implemented Trojan Columns in DBMS X, a closed source commercial database system. We ran experiments to see the performance improvements due to Trojan Columns in DBMS X.

5.6.1 Setup

We ran all experiments on a single node with 3.3 GHz Dual Core i3 running 64-bit platform Linux openSuse 12.1 OS, 4x4 GB main memory, 2 TB 5,400 rpm SATA hard disk. We use cold file system caches for all our experiments and restart the database, in order to clear database buffers, for every measurement. We repeat each measurement 3 times and report the average.

5.6.2 Baselines

Standard row store. Since Trojan Columns injects column store functionality into an existing row store, we compare the improvements over the existing standard row store.

C-Tables implementation details. Additionally, we also compare our approach with C-Tables [18], a recent approach to mimmic columns stores in row-oriented databases. We tried to implement C-Tables as closely to the description in the paper as possible. The main idea of C-Tables is "to extended the vertical partition approach to explicitly enable the RLE encoding of tuple values". For a given relation, we need to sort its attributes before applying RLE encoding. However, the paper does not describe how to order the attributes when sorting. Therefore we ran micro-benchmarks to determine the best ordering. The results indicated that the most promising strategy is to order the attributes by their cardinality, i.e. we sort on the lowest cardinality attribute first. While creating C-Tables we also discovered that it does not make sense to create C-Tables for every attribute. This is because the RLE encoding becomes lesser effective as we move towards the rightmost attribute in the sort order. Thus, in our implementation we create C-Tables only for attributes which result in smaller tables. For the remaining attributes we simply create an uncompressed vertical partition. Finally, in [18], the authors suggested to use pre-joined materialized views for C-Tables. However, we do not create any pre-joined materialized views for any of the experiments since we want to see the overall query costs, including possibly costly joins.

5.6.3 Methodology

The aim of our experiments is to answer three questions:

(1.) *What difference can Trojan Columns make on TPC-H queries?* To answer this, we study the impact of Trojan Columns on TPC-H queries. It is noteworthy here that most of the major studies on column stores [1, 3, 66, 116] have used either modified or simplified TPC-H benchmark. Still, we decided to use unmodified TPC-H benchmark since we do not want to reduce the number of projected attributes nor use pre-joined materialized views. We compare the query and upload performance of Trojan Columns with CTables [18] and standard row. (Section 5.6.4.2).
(2.) *How well does Trojan Columns work on micro-benchmarks?* After seeing the impact of Trojan Columns on TPC-H benchmark, we study the pros and cons of Trojan Columns on a single table micro-benchmarks. We show how tuple and attribute selectivity affects Trojan Columns. We also ran Trojan Columns over the simplified TPC-H queries proposed in C-Store paper [116] and discuss the results. (Section 5.6.5).
(3.) *Does implementing column stores natively make sense?* Finally, we discuss how much we could improve, in terms of I/O, if Trojan Columns was to be implemented natively within the database system, instead of being plugged-in as UDFs. We present a modified PostgreSQL implementation which implements Trojan Columns as a new access method within the database system. We compare the I/O costs of Trojan Columns with those of a native implementation in PostgreSQL, and discuss the costs and benefits of doing so. (Section 5.6.6).

5.6.4 Trojan Columns on TPC-H queries

First of all, let us see what difference can Trojan Columns make on analytical workloads. For this, we use TPC-H benchmark queries *unmodified* [126]. Since Trojan Columns affects only the data access layer, we sub-divide TPC-H queries into two sets: nested and non-nested queries. Furthermore, in order to see the impact of selectivity, we sub-divide the sets of nested and non-nested queries into high and low selectivity ones. Table 5.1 shows the four subsets of TPC-H queries thus produced.

Set ID	Nested	Selectivity	Queries
1	no	high	Q_1, Q_6, Q_{12}, Q_{14}
2	no	low	Q_3, Q_5, Q_{10}, Q_{19}
3	yes	high	Q_2, Q_4, Q_8, Q_{15}
4	yes	low	remaining 9 queries

TABLE 5.1: TPC-H queries divided into four subsets

5.6.4.1 Experiment 1: TPC-H dataset load times

First let us see the creation times of Trojan Columns and C-Tables in comparison to standard row in DBMS X. We started with TPC-H scale factor of 10, i.e. total size of 10GB. However, even after several hours C-Table did not finish uploading. This lead us to scale down the dataset to factor 1, i.e. total size 1GB. Table 5.2 shows the upload time for 3 TPC-H tables which use either of C-Table or Trojan Column in any of the queries. For Standard Row we measure the time it takes to load data from files into the

row store. For C-Table and Trojan Columns we measure the additional time to load data from the row store into their representations. However note that Trojan Columns could in principal also be loaded directly from outside files. For Trojan Columns, we set segment size to 1M for `lineitem`, 500K for `orders`, and 100k for `part` tables. From Table 5.2 we can see that the creation time for C-tables is up to 12 times slower than row. On the other hand, Trojan Column creation time is better than the upload time for standard row.

Table	Standard Row	C-Table	Trojan Columns
`lineitem`	313	3,777	179
`orders`	64	706	47
`part`	14	124	14

TABLE 5.2: TPC-H load times (in seconds) for scale factor 1.

Table 5.3 shows the upload times for standard row and Trojan Columns with scale factor 10. We do not show C-Table in the figure since since they were not competitive for scale factor 10. For Trojan Columns we also scale the segment sizes by factor 10, e.g. for `lineitem` we use segment size of 10M. From Table 5.3 we can see that, for `lineitem`, Trojan Columns creation time is twice the standard row upload time. This is because the data load UDF first reads the standard row table (I/O costs), converts it into the blob representation (CPU costs), and finally inserts the blobs into the physical table (again I/O costs). However, the Trojan Columns creation time is much better for smaller tables like `orders` and `part`. Hence, in summary we conclude that the Trojan Columns creation time is much better in comparison to C-Tables and has acceptable overhead over the standard row store loading times.

Table	Standard Row	Trojan Columns
`lineitem`	771	1,451
`orders`	484	369
`part`	100	59

TABLE 5.3: TPC-H load times (in seconds) for scale factor 10.

5.6.4.2 Experiment 2: TPC-H query times

Table 5.4 shows the query times for standard row, C-Table, and Trojan Columns over Set 1 of TPC-H queries, i.e. non-nested and highly selective queries.

Query	Standard Row	C-Table	Trojan Columns
Q_1	8.20	211.39	3.31
Q_6	8.23	96.17	2.18
Q_{12}	9.80	5457.37	5.07
Q_{14}	8.61	335.55	4.41

TABLE 5.4: TPC-H query times (in seconds) for scale factor 1.

We see that for all four queries C-Tables perform worse than standard row. Furthermore, C-Tables could be up to 500 times slower (for Q_{12}) than standard row. In contrast, Trojan Columns improves the runtime of all queries in Set 1, up to factor 3.8 (for Q_6). Thus, henceforth, we will not consider C-Tables and only use scale factor 10 in further experiments.

Now let us see the same query Set 1 on scale factor 10. Figure 5.5 shows the results. We can see that Trojan Columns outperforms standard row over all queries in query Set 1.

The maximum improvement is by factor 9 for Q_6, followed by factor 4 for Q_1, factor 2.6 for Q_{14}, and factor 2.5 for Q_{12}. All this in the same database system (DBMS X) and without touching the source code.

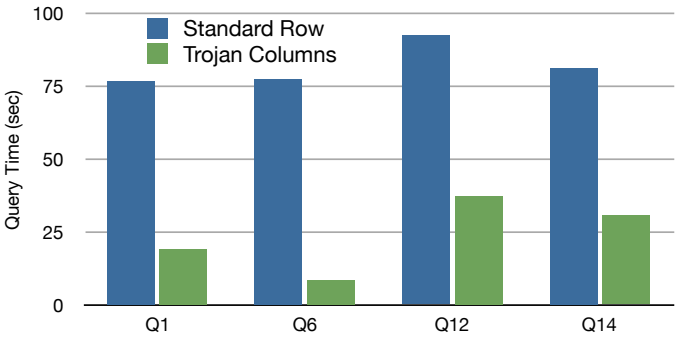


FIGURE 5.5: Unmodified TPC-H query Set 1 runtimes.

Next, let us see the query times for TPC-H query Sets 2 and 3. Tables 5.5 and 5.6 show the results. We can see that, apart from Q_{19}, Trojan Columns does not perform very well with low selectivity queries of Set 3. This is because each call to the UDF interface has some overhead: the lower the selectivity, the more function calls, the higher the overhead. In principal, this overhead could be removed if the database storage interface were available in LLVM bitcode. Then the UDF query could at runtime be dynamically recompiled *together with the DBMS storage layer* to remove that boundary and bake the UDF into the kernel. This remains an interesting avenue for future work.

Query	Standard Row	Trojan Columns
Q_3	111.88	809.38
Q_5	99.73	169.34
Q_{10}	110.94	119.46
Q_{19}	79.14	43.12

TABLE 5.5: TPC-H query Set 2 runtimes (in seconds).

Query	Standard Row	Trojan Columns
Q_2	-	-
Q_4	110.76	-
Q_8	97.38	97.66
Q_{15}	80.51	66.91

TABLE 5.6: TPC-H query Set 3 runtimes (in seconds).

Trojan Columns performs similar or better than standard row for query Set 3 (nested and high selectivity), as shown in Table 5.6. However, query nesting reduces the benefits of using Trojan Columns. This is because Trojan Columns only improves the I/O costs, which is just a fraction of the overall query costs. Apart from I/O, the remaining query processing costs are still the same as those for standard row. Note that standard row does not terminate for query Q_2, since we do not consider indexes in our experiments. Likewise, Trojan Columns does not terminate for both queries Q_2 and Q_4. This is because the optimizer cannot correctly estimate the costs of UDFs. DBMS X allows for providing UDF cost estimate hints to the optimizer. However, in the current version, the optimizer still chooses nested loop joins instead of hash joins in the query plan — we consider this a bug in DBMS X's optimizer.

5.6.4.3 Experiment 3: read-UDF costs

The focus of Trojan Columns in this chapter is to improve query I/O cost. However, as mentioned in the previous section, I/O is just a fraction of the total query costs. Since the database system is unaware of the column store inside, the query processing costs remain the same *outside* the read-UDF. To better understand the impact of Trojan Columns, let us now see the query times *inside* the subquery. To do so, we measure the time to compute the subquery computed by the read-UDF using (1) Standard Row, (2) Trojan Columns, and (3) reading a Materialized View perfectly matching the query expression.

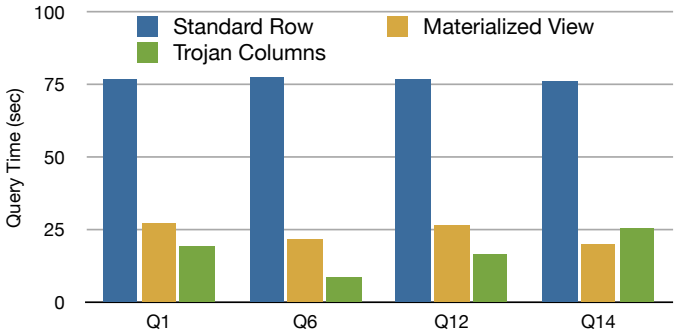


FIGURE 5.6: Query processing costs of TPC-H query Set 1 for read-UDFs. Trojan Columns versus Materialized Views.

Figure 5.6 shows the results. We can see that Trojan Columns is significantly better (factor 5 on average) than standard row. Furthermore, we also see that except for Q_{14} Trojan Columns actually outperforms Materialized Views by a factor of up to 2.5. This is because Trojan Columns benefits from efficient column-oriented compression. Query Q_{14} has the lowest selectivity (1.25%) in query Set 1, and therefore Trojan Columns does not perform as well as Materialized Views. This is a very good result considering that Materialized Views require 12GB of storage in this experiment, whereas Trojan Columns only requires 5GB. Still, the performance of Trojan Columns is very close to Materialized Views for Q_{14}. Thus, we conclude that Trojan Columns provides considerable improvements in terms of I/O costs.

5.6.5 Trojan Columns on micro-benchmarks

In this section, we evaluate Trojan Columns on two micro-benchmarks. The idea is to see the impact of Trojan Columns on simpler queries. These type of queries have been used in previous studies [1, 3, 66, 116].

5.6.5.1 Experiment 4: Varying selections and projections over a single table.

Our first micro-benchmark consists of queries of the following form over the `lineitem` table.

```
SELECT attr_1,attr_2,..,attr_r FROM lineitem
  WHERE l_partkey >= lowKey AND l_partkey < highKey;
```

We vary the selectivity of the above query (by adjusting lowKey and highKey) as well as the number of projected attributes. Figure 5.7 shows the improvement factor of Trojan Columns over standard row when varying the number of referenced attributes from 1 to 16, and selectivity from 10^{-6} to 1.

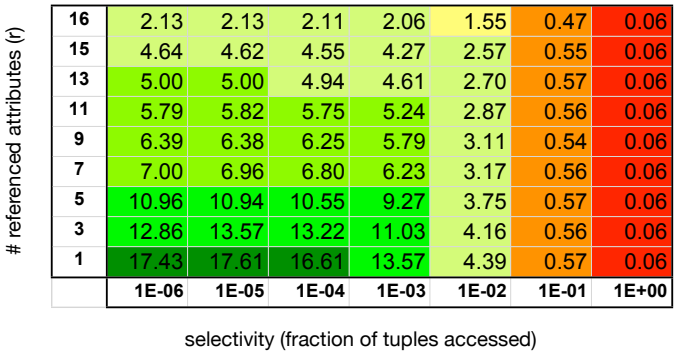


FIGURE 5.7: Trojan Columns improvement factor in DBMS X.

Figure 5.7 shows that Trojan Columns has the maximum improvement factor of over 17 (lower left region). Also, we see that for low selectivities (≥ 0.1) Trojan Columns performs worse than standard row. This is because for these selectivities function call overheads overshadow our performance improvements for low selectivities — as already discussed in Section 5.6.4.2. However, overall even for medium sized selectivites the performance gains of Trojan Columns are tremendous.

5.6.5.2 Experiment 5: Simplified TPC-H queries.

In our second micro-benchmark, we use the *simplified* TPC-H queries as proposed in the C-Store paper [116], and also used by other researchers [18]. The simplified benchmark made considerable changes over TPC-H: (1) it exploits prematerialized joins (D-tables) for column store results only. Hence the column store does not have to compute a table-join at query time. (2) It defined a Set of 7 much simpler queries than the original TPC-H queries. The simplified queries have to touch only very few attributes, i.e. up to 3 rather than up to 19. (3) it presorts the tables to allow for efficient sort-based grouping and compression, (4) it simplifies the schema of the tables. We believe that these kind of changes make benchmark results somewhat hard to interpret.

Therefore we *neither* pre-materialized joins *nor* pre-sort the data in our experiments; we simply take the 7 simplified queries from [116] and *leave everything else unchanged*. Table 5.7 shows the results. From the table we can see that Trojan Columns has an

Query	Standard Row	Trojan Columns	Improvement Factor
Q_1	76.61	5.76	13.30
Q_2	76.77	6.49	11.84
Q_3	77.95	9.20	8.47
Q_4	96.50	135.28	0.71
Q_5	91.37	78.72	1.16
Q_6	100.35	137.73	0.73
Q_7	121.74	472.83	0.26

TABLE 5.7: Simplified TPC-H query runtimes (in seconds).

improvement factor of 13.3, 11.84, and 8.47 over standard row for Q_1, Q_2, and Q_3

respectively. However, for queries Q_4, Q_6, and Q_7, Trojan Columns performs worse than standard row. This is because these queries have very low selectivity and, as seen in the previous micro-benchmark, Trojan Columns does not pay off.

5.6.6 Trojan Columns vs Column Stores

5.6.6.1 Experiment 6: Trojan Columns vs PostgreSQL Column

One final question remains to be answered: what if we implement column store functionality natively inside an existing row-oriented DBMS? That should be much more efficient than using Trojan Columns as we can change every piece of source code, right?

As the source code for a commercial system was not available for us, we used PostgreSQL and extended its storage layer to provide blob storage similar to Trojan Columns, however all implemented into the source code. It turned out that realizing this in PostgreSQL was a major endeavor: it is quite easy to add new operators to PostgreSQL, however changing the storage layer is not as easy due to considerable system-wide effetcs including TOAST. We implemented PostgreSQL Column in PostgreSQL 9.0.1. We compare PostgreSQL Column with an implementation of Trojan Columns in PostgreSQL (not DBMS X as before). We use a synthetic dataset of 18 integer attributes (1 primary key, 1 selection key, 16 different skew-cardinality combinations). We vary the selectivity from 10^{-1} to 10^{-8} and the number of projected attributes from 2 to 18. Figure 5.8 shows the results, i.e. the improvement of Trojan Columns over PostgreSQL Column.

Unsurprisingly, we observe that PostgreSQL Column is faster than Trojan Columns for low selectivities. This is because PostgreSQL does not have the function call overhead.

However, contrary to our expectation, Trojan Columns is considerably faster than PostgreSQL Column for medium to high selectivities! This result was quite surprising to us. The reasons is PostgreSQL's rowID organization: it collects 24 Bytes of metadata for each row. We keep that metadata in a separate (virtual) blob. However, at query time we need to read that metadata for every row due to interfacing issues with the PostgreSQL code. Hence Trojan Columns benefits much more from fewer columns than PostgreSQL Column. Hence the dark green area in the top left corner. These results might look different for other commercial database systems. But again: those systems are closed source and not publicly available.

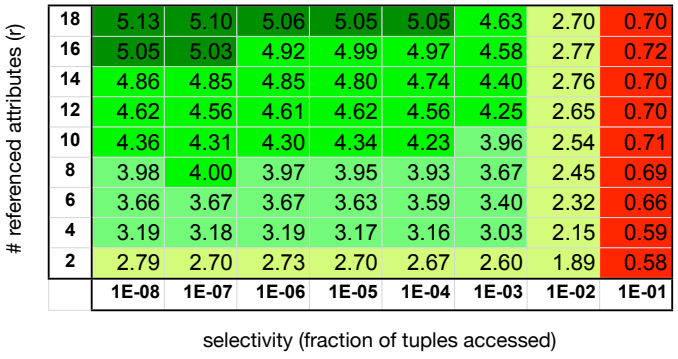


FIGURE 5.8: Improvement of Trojan Column over PostgreSQL Column

5.6.6.2 PostgreSQL Column Implementation Effort

PostgreSQL, while written in a modular fashion, is tightly coupled and contains a large amount of assertions. This makes it very difficult to make deep seated changes in PostgreSQL, such as in the access layer, since the chain of modifications result in much more work than the implementation itself. A custom access layer must work seamlessly with the query executor as well as with the database buffers. Thus integration required a considerable amount of effort. For instance, it is necessary to first write tuples in standard row format (to support inserts), and then to reprocess them into column format once enough data has been written. Also, additional columns need to be inserted for metadata that needs to be kept for each tuple (24 bytes per tuple). Overall, the lines added/changed amount to less than 200 (compared to about 1,300 lines overall). However, the integration took more than twice the implementation effort. In total, PostgreSQL Columns took about three man-months — and still it only supports insertions and selections.

5.6.7 Additional Results

5.6.7.1 Effect of Compression

Now let us see the impact of compression on Trojan Columns. We play around with four light weight compression techniques for column stores: delta, 7-bit, dictionary, and run length encoding. We look at the cardinalities of each of the attributes, estimate the expected compression ratio, and pick the compression method which gives the maximum compression ratio. For dictionary compression, we also consider the expected dictionary size. Since Trojan Columns works very well on query set 1, we focus on that in this section. Also note that Trojan Columns is used only for `lineitem` table in query set 1, since other tables have no selection predicates and hence have very high function calls overhead. We successively turn on four levels of compression on `lineitem` table as follows:

Compression Level 0. no compression applied.
Compression Level 1. delta encoding enabled.
Compression Level 2. Level 1 + 7-bit encoding enabled.
Compression Level 3. Level 2 + dictionary compression enabled.
Compression Level 4. Level 3 + run length encoding enabled.

Table 5.8 shows the data load times for different compression levels.

Measurement	Level 0	Level 1	Level 2	Level 3	Level 4
Upload Time (sec)	1636.44	1642.73	1610.59	1550.63	1451.39
Table Size (GB)	11.83	11.57	10.89	5.19	5.07

TABLE 5.8: Upload times and tables sizes with compression.

We can see that the upload time decreases with higher levels of compression. From compression level 0, i.e. no compression, to compression level 4, the improvement in upload time is 185 seconds. The reason for this is that while we spend more CPU cycle to compress the data, we save on I/O when writing the data to disk. Figure 5.9 shows the query times of query set 1 for different compression levels. We can see that the query time improves up to compression level 4. For instance, for query 6, the improvement in

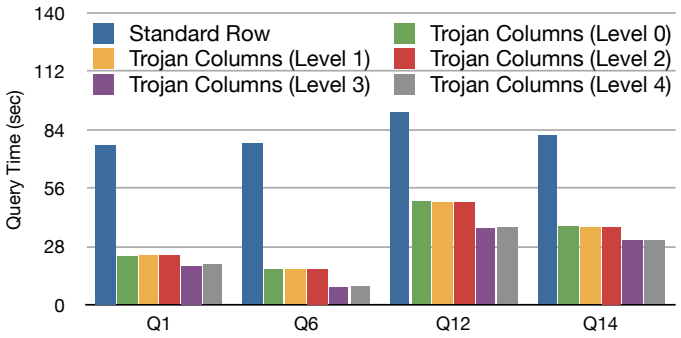


FIGURE 5.9: Effect of compression on query set 1.

runtime from compression level 0 to compression level 4 is 49%. However, the improvement is not dramatic. The reason for this is that the overall query costs as well as the UDF overheads dominate the improvements in I/O due to compression.

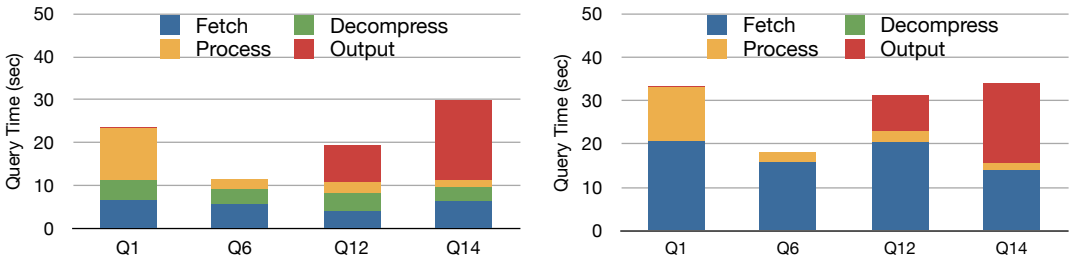
Table 5.9 shows the compression methods used for each of the `lineitem` attribute. Except `ExtendedPrice` (too small compression ratio) and `Comment` (VARCHAR), we apply compression on all `lineitem` attributes.

Attribute	Compression Method	Expected Compression Ratio
OrderKey	Delta	4
PartKey	7-Bit	1.4
SuppKey	7-Bit	2
Linenumber	7-Bit	4
Quantity	Dictionary	8
ExtendedPrice	None	-
Discount	Dictionary	8
Tax	Dictionary	8
ReturnFlag	Run Length Encoding	1.42
LineStatus	Run Length Encoding	3.63
ShipDate	Dictionary	4.98
CommitDate	Dictionary	4.98
ReceiptDate	Dictionary	4.98
ShipInstruct	Dictionary	25
ShipMode	Dictionary	10
Comment	None	-

TABLE 5.9: Compression methods and the expected compression ratio for each `Lineitem` attribute.

5.6.7.2 Query cost break-down

In order to understand where we can further improve the UDFs, we need to see its cost breakdown. Figure 5.10(a) shows the breakdown of UDF processing time into four costs: fetching data, decompressing data, processing (selections, grouping/aggregation), and outputting the results. From the figure we can see that processing costs dominate in query Q_1 while outputting costs dominate in query Q_{14}. However, fetching and decompression are at the major costs for Q_6 and Q_{12}. To contrast the effect of compression, Figure 5.10(b) shows the cost breakdown for uncompressed data. We can see that there are no decompression costs now, however the fetching costs go up significantly and dominate most queries.



(a) With compression (b) Without compression

FIGURE 5.10: Query cost breakdown (in seconds) for Trojan Columns over TPC-H query set 1.

5.6.7.3 Stored Procedures

Figure 5.11 shows the runtimes of Trojan Columns using stored procedures (SP) for TPC-H query set 1. We can see that except for query Q_{14}, Trojan Columns using stored procedures are very close to those Trojan Columns using UDFs. Stored procedures are slow for Query Q_{14} because it produces a large number of output tuples. Since stored procedures cannot return the results, they must write these output tuples into another table.

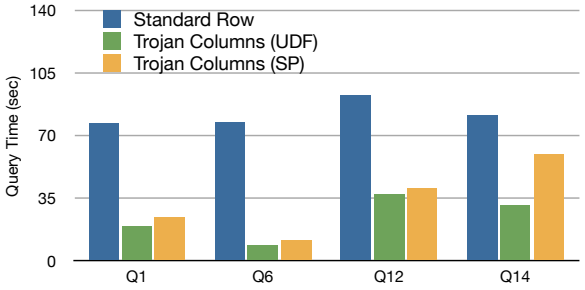


FIGURE 5.11: Query times (in seconds) with stored procedures.

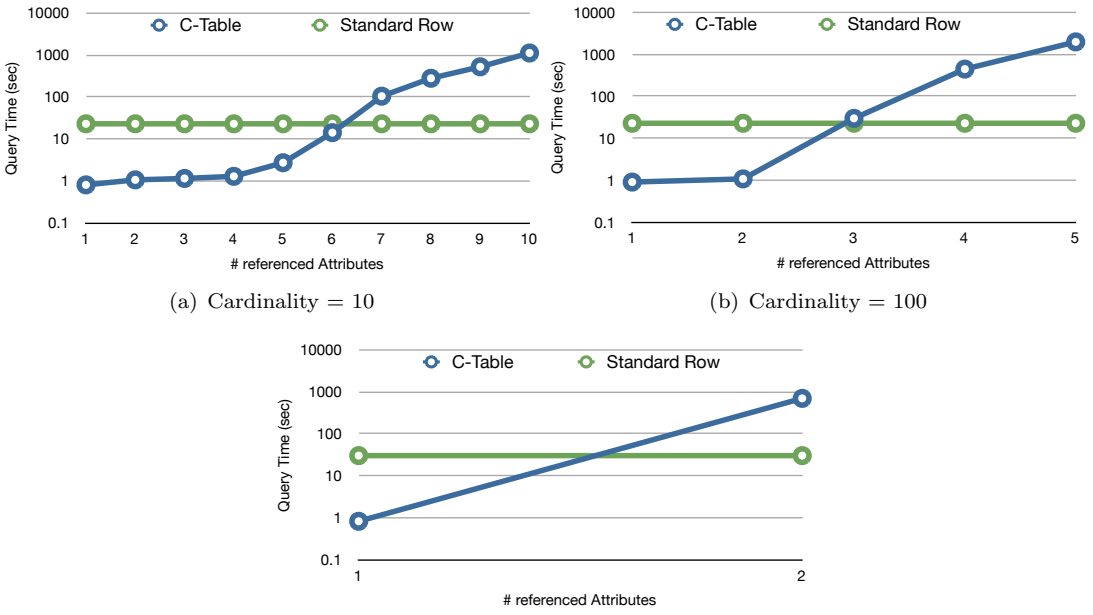
5.6.7.4 C-Table Evaluation

In order to investigate C-Table in more detail, we ran some micro-benchmarks on them. We take three synthetic datasets. Each dataset contains integer attributes with the same cardinality (10, 100, and 1000 respectively). For each dataset, we create C-Tables over its attributes and vary the number of referenced attributes.

Figure 5.12 shows the results. We can see that for lower cardinalities C-Tables work very well compared to standard row. For instance, for cardinality 10, C-Tables are better than row for up to 6 referenced attributes. However, for higher cardinalities e.g. 1000, C-Tables become bad pretty soon.

5.7 Discussion

Trojan Column Benefits. From the above experiments, we see that Trojan Columns significantly improves the performance of DBMS X. This is because Trojan Columns



(a) Cardinality = 10

(b) Cardinality = 100

(c) Cardinality = 1000

FIGURE 5.12: Comparing query times of CTable and standard row for different attribute cardinalities.

can successfully emulate a column store without any overhead that typically exists in full vertical partitioning or other schema level approaches. The main advantage of Trojan Columns comes from improved I/O performance: we access only the referenced attributes. In addition, we apply light-weight column-oriented compression schemes. Furthermore, we push one or more SQL operators down to the UDF and evaluate them directly directly on BLOB data.

In contrast, C-Tables work only for up to 3 attributes, unless the input datasets are pre-joined. For a higher number of referenced attributes, the tuple reconstruction joins kill the C-Table performance. This is not the case for Trojan Columns. In fact, as we saw in the experiments, Trojan Columns is not at all affected by the number of referenced attributes.

Trojan Column Limitations. We found that the major performance problem in using table UDFs, for accessing Trojan Columns, is the additional overhead of function calls. These function calls lead to a significant decrease in performance if many rows are returned. For each row that is returned to the outside, DBMS X invokes one call of the UDF and passes a large number of arguments to it. For a table like `lineitem` with 16 attributes, this means passing already 32 return variables (16 return variables + 16 indicators) to the function in each call, additional to the arguments passed by the user. Thus low selectivity queries are a problem for Trojan Columns.

The UDFs which perform only projection and selection are very flexible and need only the table schema to be generated. For example, we have one UDF for `lineitem`, and it can perform all kinds of projections and selections on the table. This is possible since the result schema of the query is always a subset of the `lineitem` schema. However, for queries which also perform grouping and aggregation, we need to generate the UDFs for each individual query. This is because the schema of the query result might not be a

subset of the original table schema. Though manual, these adaptions can still be done easily and quickly.

As future work, a main goal is to increase the flexibility of the approach. We are planning to build generators and compilers (also in the form of UDFs), which create and install all necessary functions for a given query or table and database product. This is possible since only small adaptations are needed to tweak a function towards a query. Another main problem we face at the moment is the additional overhead at the leaf level caused by too many result rows. We could eliminate this problem in many cases, if we could push the join operator into the UDF. This would also allow for performing grouping/aggregation after the join and would lead to a significant decrease of function call overhead.

Query Optimization Considerations

(a) Selectivity. At the moment, we decide whether or not to use Trojan Columns manually. Ideally, however, we would like to hide this decision using a view, which is then used in the query invoked by the user. The view should be able to switch between Trojan Columns and the standard row store, depending on the query selectivity. Note that the view needs to pass the projection and selection operators down to the UDF, in case it chooses Trojan Columns.

(b) UDF cost estimates. DBMS X supports a mechanism to adjust the estimated cost of a UDF in terms of the expected cardinality, i.e. it is possible to specify the number of rows that the UDF might return. Unfortunately, this cardinality is static and has to be set for each individual query. This cardinality information is then used in the access plan calculation to find the best plan.

(c) Intermediate results. If a query materializes intermediate results on disk, then the optimizer could consider using Trojan Columns for them, thus improving performance higher up in the query tree.

5.8 Conclusion

In this chapter, we presented Trojan Columns, a radically different approach for injecting column store technology into a closed source commercial row-oriented database system. Trojan Columns does not make any changes to the source code of the database system, but rather use UDFs as a pluggable storage layer for data read and write. Trojan Columns can be easily integrated into an existing database system environment (without even restarting the DBMS). As a result, Trojan Columns is transparent to the user, i.e. the user continues using his existing database product with minimal changes to his queries. We implemented Trojan Columns in DBMS X and show query runtimes from unmodified TPC-H benchmark, simplified TPC-H queries (as proposed by other researchers), as well as from single table micro-benchmarks. Our results show that Trojan Columns improves the performance of DBMS X by up to a factor 9 (3.9 on average, 2.5 in the worst case) for non-nested high selectivity queries of unmodified TPC-H benchmark, by up to a factor 13 (5.2 on average, 3.9 slowdown in the worst case) for simplified TPC-H queries, and by up to a factor 17 for high selectivity queries on single table micro-benchmarks. All this without touching the source code of the database system.

With Trojan Columns, we have the flexibility to create a compressed column storage view, in addition to the standard row storage view, in commercial row-oriented database systems. As a result of this greater storage layer flexibility with Trojan Columns, we can now have efficient OLAP performance in the OLTP efficient database system.

Chapter 6

Conclusion

Data managing needs in our modern world are changing faster than ever before. Gone are the days of fixed query workloads. Today, enterprises typically face the challenge of managing big datasets with dynamically changing query workloads. It is clear that the enterprises cannot afford to have different data managing systems for different query workloads. Otherwise, there is an additional penalty of managing a zoo of data managing systems in the first place, which is tedious, expensive, as well as counter-productive for modern enterprises. Instead, modern data managing systems must inherently provide flexible data management techniques in order to cope with the constantly changing business needs.

6.1 Summary

In this thesis, we introduced Trojan Techniques for data management. We observed that each specialized database product has a different data store. This indicates that different data layouts are suited for different query workloads. Therefore, a key requirement for efficiently supporting several query workloads is to support several data layouts (physical data representation). Trojan Techniques add additional data layouts to an existing data management system without making heavy untenable changes to the system. At the same time, Trojan Techniques brought significant improvements in query performance.

(1.) OctopusDB Vision. First we presented the long term vision and goal of our research. Our vision is a unified data managing architecture for supporting several types of workloads. The primary storage structure of such a system, coined OctopusDB, is a logical log. All other storage structures are just secondary *storage views* on this log. Storage views include row layout, column layout, indexes, but also windows on unbounded streams. With OctopusDB we inverted the traditional development of a DBMS: previously there always was a specific store which was an irrevocable design-decision, built-in into the DBMS. On top of that, an ARIES-style [90] log-based recovery was implemented to guarantee atomicity and durability. In our vision, we took exactly the opposite approach: we started with the log (which is totally disconnected from any store) and if necessary, we define optional SVs on that log suited for a particular workload.

The Trojan Techniques, introduced in this thesis, aim to develop the flexible storage layer as envisioned in OctopusDB. Thus, Trojan Techniques are the first steps of the bigger OctopusDB vision. Although we originally presented the OctopusDB vision in the context of relational databases, it is more broad in scope and can be applied to any data managing system. Therefore, in this thesis, we focus more on implementing this vision in Hadoop MapReduce, given its recent popularity. Still, we presented one Trojan Technique in relational databases as well.

(2.) Trojan Index and Joins. We started with Hadoop MapReduce and proposed new index and join techniques: Trojan Index and Trojan Join, to improve runtimes of MapReduce jobs. Our techniques are non-invasive, i.e. they do not require us to change the underlying Hadoop framework. We simply need to provide appropriate user-defined functions (and not only the two functions `map` and `reduce`). The beauty of this approach is that we can incorporate such techniques to any Hadoop version with ease. We exploited this during our experiments when moving from Hadoop 0.20.1 to Hadoop 0.19.0 (used by HadoopDB) for fairness reasons. We implemented our Trojan techniques on top of Hadoop and named the resulting system Hadoop++. The experimental results demonstrated that Hadoop++ outperforms Hadoop. Furthermore, for tasks related to indexing and join processing Hadoop++ outperformed HadoopDB – without requiring a DBMS or deep changes in Hadoop's execution framework or interface.

With Trojan Index and Trojan Join, Hadoop++ introduces indexed and co-partitioned storage views, apart from the default row storage view, in Hadoop MapReduce. As a result of this greater flexibility in the storage layer, Hadoop++ allows for efficient index and join processing, in addition to standard scan-oriented processing.

(3.) Trojan Layouts. After indexing and co-partitioning, we turned our attention to data layouts in Hadoop MapReduce. We proposed Trojan Layouts, a new data layout that organizes data inside HDFS data blocks according to the incoming workload. We followed the PAX principle in that we did not change the outside view of data. However, we considerably departed from PAX as we: (i) might co-locate attributes according to query workloads, (ii) may use different Trojan Layouts for different data block replicas, and (iii) may, in a special case, mimic fractured mirrors: having the best from both PAX and Row Layouts. We implemented our algorithms on top of HDFS 0.20.3. A salient feature of using per-replica Trojan Layouts is that we can schedule incoming jobs to data block replicas having the best Trojan Layout. On TPC-H, SSB, and SDSS benchmarks, Trojan Layouts outperformed the default row layout by up to a factor of 4.8 (3.3 times faster on average and 1.1 times slower in the worst case); and PAX layout by up to a factor of 3.5 (1.6 times faster on average and no improvement in the worst case).

With Trojan Layouts, we have the flexibility to choose a different storage view for each data block replica in Hadoop MapReduce. This has important consequences since data replication is intrinsic to Hadoop MapReduce, in order to provide fault-tolerance and failover properties. We can now have a flexible storage layer with as many storage views as the data replication factor. As a result of Trojan Layouts, we can now have efficient column-oriented as well as arbitrary partial-projection-oriented processing, in addition to standard row-oriented processing in Hadoop MapReduce.

(4.) Trojan Columns. Finally, we looked at relational data management systems. We presented Trojan Columns, a radically different approach for injecting column store technology into a closed source commercial row-oriented database system. Trojan Columns does not make any changes to the source code of the database system, but rather use UDFs as a pluggable storage layer for data read and write. Trojan Columns can be easily

integrated into an existing database system environment (without even restarting the DBMS). As a result, Trojan Columns is transparent to the user, i.e. the user continues using his existing database product with minimal changes to his queries. Our experimental results showed that Trojan Columns improves the performance of DBMS X by up to a factor 9 (3.9 on average, 2.5 in the worst case) for non-nested high selectivity queries of unmodified TPC-H benchmark, by up to a factor 13 (5.2 on average, 3.9 slowdown in the worst case) for simplified TPC-H queries (as proposed by other researchers), and by up to a factor 17 for high selectivity queries on single table micro-benchmarks. All this without touching the source code of the database system.

With Trojan Columns, we have the flexibility to create a compressed column storage view, in addition to the standard row storage view, in commercial row-oriented database systems. As a result of this greater storage layer flexibility with Trojan Columns, we can now have efficient OLAP performance in the OLTP efficient database system.

6.2 Future Work

Fully Flexible Storage Layer. Trojan Techniques presented in this thesis added several new storage views in Hadoop MapReduce. Still, we are quite far from having a fully flexible storage layer in Hadoop MapReduce. Ideally, we should be able to decide the physical data representation on a per-HDFS-block basis, i.e. each HDFS data block can have a different layout. Trojan Layouts piggy backs on the default replication factor of 3 to create a different layout for each replica. However, we can create even more data layouts, and hence achieve even better query performance, with a higher replication factor. Thus, now the replication factor is given by not only the degree of fault tolerance but also by the desired query performance. Furthermore, currently, Trojan Layouts needs to replicate the data entirely in order to have a different storage view. However, we could also replicate only parts of the data, within a data block, and store them in a different physical representation. Similarly, we can also store any intermediate or final result in any physical representation. Essentially, we want the storage layer to be completely decoupled and fully flexible. Only then, we will fix a major flaw in current data managing systems — absence of true physical data independence.

Trojan Techniques for Query Optimization. Trojan Techniques have three major advantages: (i) they are implemented in the existing (and familiar) data managing system, i.e. the user is not overwhelmed with yet another new data managing system, (ii) they are plug and play, i.e. the administrators do not have to worry about configuring or tuning them, and (iii) they are generic and can be used across several systems. However, currently, Trojan Techniques focus on data access paths and physical data design in existing data managing systems. As a result even though Trojan Techniques provide for a highly flexible storage layer, existing query optimizers still cannot make full use of that layer. Thus, as future work, we need to extend Trojan Techniques to work with existing query optimizers. This means that we have to inject optimization decisions, which exploit the underlying flexible storage layer and at the same time make use of an existing query optimizer.

Considering OLTP and Other Applications. In this thesis, we focussed on OLAP/-analytics applications. We started with the default row-oriented storage of Hadoop MapReduce and introduced several Trojan Techniques, including indexes, joins, and

layouts, to support analytical applications efficiently. We did the ground work for a flexible storage layer, which would make it possible to support several kinds of workloads. Still, we are far from a one-size-fits-all data managing system. As a future work, we need to extend Trojan Techniques to efficiently support OLTP, streaming, and other demanding applications.

Putting Trojan Techniques Together. The Trojan Techniques presented in this thesis were proposed in a variety of systems. Each Trojan Techniques was proposed independently in order to understand and evaluate it in isolation. However, as future work, we need to bring together all these techniques into a single data managing system. The single system should examine the feasibility of Trojan indexes, joins, data layouts, and columns and automatically adapt according to the incoming query workload.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

[3] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, 2008.

[4] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented Database Systems (Tutorial). *PVLDB*, 2(2), 2009.

[5] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[6] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.

[7] Foto N. Afrati, Chen Li, and Jeffrey D. Ullman. Generating Efficient Plans for Queries Using Views. In *SIGMOD*, 2001.

[8] Sanjay Agarwal, Lubor Kollár Surajit Chaudhuri, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.

[9] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*, 2004.

[10] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD*, 2006.

[11] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[12] Ioannis Alagiannis, Debabrata Dash, Karl Schnaitter, Anastasia Ailamaki, and Neoklis Polyzotis. An Automated, Yet Interactive and Portable DB Designer. In *SIGMOD*, 2010.

[13] Don S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE TSE*, 14(11), 1988.

[14] BerkeleyDB. `http://www.oracle.com/technetwork/products/berkeleydb`.

[15] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[16] Dina Bitton and David J. DeWitt. Duplicate Record Elimination in Large Data Files. *ACM TODS*, 8(2), 1983.

[17] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[18] Nicolas Bruno. Teaching an Old Elephant New Tricks. In *CIDR*, 2009.

[19] Nicolas Bruno and Surajit Chaudhuri. Physical Design Refinement: The 'Merge-Reduce' Approach. In *EDBT*, 2006.

[20] Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.

[21] Nicolas Bruno and Surajit Chaudhuri. Constrained Physical Design Tuning. *PVLDB*, 1(1), 2008.

[22] Michael J. Cafarella and Christopher Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *WebDB*, 2010.

[23] George Candea, Neoklis Polyzotis, and Radek Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *PVLDB*, 2009.

[24] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. ES2: A Cloud Data Storage System for Supporting Both OLTP and OLAP. In *ICDE*, 2011.

[25] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *VLDB*, 1986.

[26] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In *SIGMOD*, 1994.

[27] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2), 2008.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[29] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

[30] Surajit Chaudhuri and Vivek R. Narasayya. Self-Tuning Database Systems: A Decade of Progress (Ten Year Best paper Award). In *VLDB*, 2007.

[31] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, 2000.

[32] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance. In *SIGMOD*, 2002.

[33] Songting Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2), 2010.

[34] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker Jr. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD*, 2007.

[35] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *VLDB Journal*, 11(3), 2002.

[36] Rada Chirkova, Chen Li, and Jia Li. Answering Queries Using Materialized Views with Minimum Size. *VLDB Journal*, 15(3), 2006.

[37] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph Hellerstein, and Caleb Welton. Mad Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[38] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *NSDI*, 2010.

[39] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[40] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[41] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L. Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, Bobbi Heath, David Maier, Samuel Madden, Jignesh M. Patel, Michael Stonebraker, and Stanley B. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS (Demo). In *PVLDB*, 2009.

[42] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[43] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *PVLDB*, 2010.

[44] DBColumn on MapReduce. `http://databasecolumn.vertica.com/2008/01/mapreduce-a-major-step-back.html`, 2008.

[45] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[46] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1), 2010.

[47] Nihal Dindar, Baris Guc, Patrick Lau, Asli Ozal, Merve Soner, and Nesime Tatbul. DejaVu: declarative pattern matching over live and archived streams of events (Demo). In *SIGMOD*, 2009.

[48] Jens Dittrich and Alekh Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.

[49] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[50] Jens Dittrich, Jorge-Arnulfo Quiane-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(9), 2012.

[51] Jens-Peter Dittrich, Peter M. Fischer, and Donald Kossmann. AGILE: adaptive indexing for context-aware information filters. In *SIGMOD*, 2005.

[52] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-Oriented Storage Techniques for MapReduce. *PVLDB*, 4(7), 2011.

[53] Clark D. French. "One size fits all" database architectures do not work for DSS. In *SIGMOD*, 1995.

[54] Clark D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.

[55] Fusion-io. `http://www.fusionio.com`.

[56] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a HighLevel Dataflow System on Top of MapReduce: The Pig Experience. *PVLDB*, 2(2), 2009.

[57] Jonathan Goldstein and Per-Åke Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, 2001.

[58] Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[59] Goetz Graefe and Harumi A. Kuno. Adaptive indexing for relational keys. In *ICDE Workshops*, 2010.

[60] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2), 2010.

[61] Hadapt. `http://www.hadapt.com`.

[62] Hadoop. `http://hadoop.apache.org/mapreduce/`.

[63] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6), 2012.

[64] Richard A. Hankins and Jignesh M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, 2003.

[65] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[66] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

[67] HDFS Bug. `http://issues.apache.org/jira/browse/HDFS-96`, 2009.

[68] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *ICDE*, 2011.

[69] Sandor Heman, Marcin Zukowski, Arjen P. de Vries, and Peter A. Boncz. Efficient and Flexible Information Retrieval using MonetDB/X100. In *CIDR*, 2007.

[70] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *SIGMOD*, 2007.

[71] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, 2007.

[72] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[73] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[74] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9), 2011.

[75] Robert Ikeda and Jennifer Widom;. Provenance for Generalized Map and Reduce Workflows. In *CIDR*, 2011.

[76] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[77] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *EuroSys*, 2010.

[78] Alekh Jindal. The Mimicking Octopus: Towards a one-size-fits-all Database Architecture. In *VLDB PhD Workshop*, 2010.

[79] Alekh Jindal and Jens Dittrich. Relax and Let the Database do the Partitioning Online. In *BIRTE*, 2011.

[80] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.

[81] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System (Demo). In *PVLDB*, 2008.

[82] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[83] Setrag Khoshafian, George P. Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *ICDE*, 1987.

[84] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. In *VLDB*, 2010.

[85] Willis Lang and Jignesh M. Patel. Energy Management for MapReduce Clusters. *PVLDB*, 3(1), 2010.

[86] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL Server Column Store Indexes. In *SIGMOD*, 2011.

[87] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1), 2008.

[88] Stefan Manegold, Martin L. Kersten, and Peter A. Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.

[89] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[90] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1), 1992.

[91] MongoDB. `http://www.mongodb.org`.

[92] Kristi Morton and Abe Friesen. KAMD: A Progress Estimator for MapReduce Pipelines. In *ICDE*, 2010.

[93] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[94] MySQL. `http://www.mysql.com`.

[95] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical Partitioning Algorithms for Database Design. *ACM TODS*, 9(4), 1984.

[96] Neo4j. `http://neo4j.org`.

[97] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3 (1), 2010.

[98] ObjectDB. `http://www.objectdb.com`.

[99] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[100] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-Aware Storage Layout for Database Systems. In *SIGMOD*, 2010.

[101] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[102] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, 1997.

[103] Rachel Pottinger and Alon Halevy. MiniCon: A Scalable Algorithm for Answering Queries Using Views. *VLDB Journal*, 10(2-3), 2001.

[104] Jorge-Arnulfo Quiané-Ruiz et al. RAFTing MapReduce: Fast Recovery on the Raft. In *ICDE*, 2011.

[105] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

[106] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[107] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *ICDE*, 2008.

[108] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.

[109] Domenico Sacca and Gio Wiederhold. Database Partitioning in a Cluster of Processors. *ACM TODS*, 10(1), 1985.

[110] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiane-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3 (1), 2010.

[111] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: Continuous On-Line Database Tuning. In *SIGMOD*, 2006.

[112] SciDB. `http://www.scidb.org`.

[113] Russell Sears, Mark Callaghan, and Eric A. Brewer. *Rose*: Compressed, log-structured replication. *PVLDB*, 1(1), 2008.

[114] Michael Stonebraker. The Case For Partial Indexes. *SIGMOD Record*, 18(4), 1989.

[115] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, 2005.

[116] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[117] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, 2007.

[118] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, 2007.

[119] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *CACM*, 53(1), 2010.

[120] StreamBase. `http://www.streambase.com`.

[121] Sybase IQ. `http://www.sybase.com/products/datawarehousing/sybaseiq`.

[122] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, 1992.

[123] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *VLDB Journal*, 17(1), 2008.

[124] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2), 2009.

[125] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*, 2010.

[126] TPC-H. `http://www.tpc.org/tpch/`.

[127] Odysseas G. Tsalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *VLDB*, 1994.

[128] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable Performance for Unpredictable Workloads. In *PVLDB*, 2009.

[129] Vectorwise. `http://www.actian.com/products/vectorwise`.

[130] VoltDB. `http://voltdb.com`.

[131] Paul Yan and Paul Larson. Data Reduction Through Early Grouping. In *CASCON*, 1994.

[132] Christopher Yang, Christine Yen, Cerye Tan, and Samuel Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, 2010.

[133] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[134] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. Dynamic Materialized Views. In *ICDE*, 2007.

[135] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

[136] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.