A Design Framework for Efficient Distributed Analytics on Structured Big Data

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Noah Orensa

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

Distributed analytics architectures are often comprised of two elements: a compute engine and a storage system. Conventional distributed storage systems usually store data in the form of files or key-value pairs. This abstraction simplifies how the data is accessed and reasoned about by an application developer. However, the separation of compute and storage systems makes it difficult to optimize costly disk and network operations. By design the storage system is isolated from the workload and its performance requirements such as block co-location and replication. Furthermore, optimizing fine-grained data access requests becomes difficult as the storage layer is hidden away behind such abstractions.

Using a clean slate approach, this thesis proposes a modular distributed analytics system design which is centered around a unified interface for distributed data objects named the *DDO*. The interface couples key mechanisms that utilize **storage**, **memory**, and **compute** resources. This coupling makes it ideal to optimize data access requests across all memory hierarchy levels, with respect to the workload and its performance requirements. In addition to the DDO, a complementary DDO controller implementation controls the logical view of DDOs, their replication, and distribution across the cluster. A proof-of-concept implementation shows improvement in mean query time by 3-6x on the TPC-H and TPC-DS benchmarks, and more than an order of magnitude improvement in many cases.

Acknowledgements

I would like to thank my supervisors Dr. Dwight Makaroff and Dr. Derek Eager. Completing this dissertation would not have been possible without their expertise, guidance, and continuous feedback. I would also like to thank my committee advisors, Dr. Natalia Stakhanova and Dr. Nadeem Jamali, for their valuable insight and suggestions which have further improved the quality of my work.

The Natural Sciences and Engineering Research Council of Canada (NSERC) and the Department of Computer Science have both provided me with financial support without which I would not have been able to pursue this programme. I express my sincerest gratitude to NSERC and the department for the support I have received. And finally, thank you to all my friends and family who kept supporting and encouraging me every step of this journey.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application programming interface
DAG	Directed acyclic graph
DDO	Distributed data object
DSL	Domain-specific language
GFS	Google file system
HDFS	Hadoop distributed file system
JVM	Java virtual machine
MPI	Message passing interface
RDD	Resilient distributed dataset
SSD	Solid state drive
SQL	Structured query language
STL	Standard template library

1 Introduction

1.1 Big Data Analytics

The ubiquity of big data and distributed computing solutions has opened new frontiers for data analytics. Terabytes of data are being ingested, transformed, and aggregated on a daily basis. As data volumes continue to grow, becoming more and more difficult to handle using a single powerful machine, the use of distributed systems is steadily increasing in numerous disciplines. The most notable and widely used distributed platform is Hadoop,[1] occupying roughly 33% of on-premise solutions [5].

MapReduce was created as a simple programming model that offers an easy and scalable way to use the compute resources of a cluster [15]. One of the main advantages of a MapReduce-supporting framework is that it removes the burden of worrying about the intricate details of a distributed computing system. Details such as synchronization, networking, and fault tolerance are no longer the worry of a distributed application developer. Instead, the developer can focus on solving the problem at hand and express their solution through two simple compute primitives: map() and reduce().

A separate, fault-tolerant, distributed file system is often used in combination with MapReduce to make use of distributed storage and feed data to the compute engine. It soon became very common to have applications consisting of multiple successive stages of map() and reduce(). Under that architecture, it meant that the output of every stage needed to be written to the file system and read again at the beginning of the next stage. It was obvious that these two systems lacked primitives to utilize distributed memory. Resilient distributed datasets (RDDs) were introduced to solve this problem [43]. The RDD interface offers an intuitive and easy way to implement even more complex distributed applications. Intermediary results are often cached automatically by the framework but can also be explicitly cached, if needed. This greatly speeds up the execution of multistage applications.

The RDD implementation of Apache Spark, coupled with a distributed file system such as the Hadoop distributed file system (HDFS) [31], is the one of the most commonly deployed big data analytics solutions today. HDFS provides data to the compute engine. The compute engine tries to optimize tasks and cache intermediary results, alleviating repetitive and costly file system access. However, there are some inherent characteristics of a distributed storage pool that make this architecture lack a few key optimizations. The remainder of this chapter discusses these drawbacks and potential ways to overcome them.

[1] Apache Hadoop. `https://hadoop.apache.org/`. Accessed November 30, 2020

1.2 Motivation

1.2.1 Cost of Generalization

Most distributed file systems and execution engines are built for general-purpose use and enabling rapid development of distributed applications. This level of generality can dictate many implementation details and the overall system behaviour. For example, in HDFS, block allocation and placement is completely workload-agnostic because no assumptions can be made about the workload at this level of generality. The performance penalty of this approach can be substantial when compared to more specialized systems which implement many optimizations for their workloads.

Furthermore, these general-purpose systems, which were built mainly for one-time jobs, are being increasingly used in specialized applications such as data warehousing [14]. This thesis advocates an approach that attempts to balance both performance and the amount of development work required to introduce new distributed applications. The following subsections discuss concrete examples of performance loss due to generalization and gives some insight on the magnitude of performance loss.

The performance metrics used here are concerned with makespan and resource utilization for a given workload. Makespan can be defined as the total duration of execution of a workload (e.g. a set of analytical queries) on the system, including any potential queuing and scheduling delays. Resource utilization for big data clusters is usually concerned with spatial allocation of resources; e.g. total number of CPU cores. These two metrics highlight differences in system efficiencies. For example, if a system a is able to finish a given workload in less time (or using less resources) than a system b, then it can be deduced that system a is more efficient than system b.

1.2.2 Distributed Storage for Analytics

Distributed file systems usually offer a hierarchical file and directory namespace, similar to most centralized storage systems. This abstraction helps build an over-simplified image about the nature of the file system. In reality, a distributed file system is very complex, and its performance can be unpredictable if not used carefully. Access times can vary greatly, due to locality differences (block location relative to access location) as well as node heterogeneity. The file and directory abstraction is intuitive and easy to use. However, in the case of distributed storage, it obscures important details which are sometimes the main focus of performance-conscious applications. One of these important details is block locality.

The files are usually split into blocks to be scattered and replicated across nodes for performance and fault tolerance purposes [19]. This introduces heterogeneity and puts stringent constraints on what can be realistically done with these files. Realizing a warehousing system on top of such distributed file systems must take into account causes of heterogenous access, which can be summarized as follows:

1. The time to read a block depends on the network distance; i.e. access location compared to physical

location (same node, same rack, same cluster, etc.), and

2. The nodes themselves could be heterogenous in terms of compute capacity and I/O bandwidths.

To mitigate the heterogeneity issues, a logical data object (such as a table) needs to be stored across many files, each roughly equal to the configured block size of the storage system. In the job planning phase, individual tasks are assigned exactly one input file and each task will be sent to (or generated at) the node housing the single-block file. This mitigates the first heterogeneity problem. The number of tasks sent to a node will also depend on its compute and I/O capacity, solving the second heterogeneity problem. Furthermore, additional metadata might be needed during query planning for optimization purposes as well as keeping track of the files that comprise the logical data objects and their structure. A lightweight DBMS (database management system) for metadata and file information could be installed at a central location to solve this issue [34]. In essence, this is similar to augmenting the namespace metadata using a sub-system.

This seems to work well if all analytical tasks to be executed will need to access one logical data object. What happens if there is more than one data object involved in an analytical task? This requires some way to indicate to the file system to co-locate the blocks of different (and seemingly unrelated) files so that we can create tasks that access more than one file without compromising the solution for locality. A more feasible option would be to denormalize the data. If the storage system were flexible enough to allow control over block placement, this would mean more opportunities for workload-specific optimizations.

The separation of file system and execution engines makes it difficult to optimize costly disk and network operations due to obscured details of both systems. HDFS, for instance, does not allow an application to specify block placement constraints. As far as the storage system is concerned, it needs to only maintain a certain number of replicas for each data block. One missing key element is the ability to co-locate data blocks that make up two or more logical data objects; i.e. co-locating blocks from two or more objects (such as tables), each having many blocks scattered throughout the cluster.

In addition to denormalizing data, a common solution is to use a caching layer, and hope that its policies (prefetch and eviction) are intelligent enough to learn the access patterns of the workload. These access patterns could have been communicated to the file system if the file system were able to give such fine-grained control over block placement. And even if the caching layer is successful in discovering the access patterns, relying on a middle layer to optimize away architectural problems may not be the best option.

A warehousing system that uses two independent compute and storage systems is bound to be inefficient due to the complexity of storage systems and varying needs of many different workloads. Moreover, distributed warehousing systems that try to offer a logical view of the data, such as Apache Hive, end up using the hierarchical structure of the file system namespace to store partitioned and bucketed data records [34]. The file and directory namespace is not always capable of efficiently encoding the logical structure of data objects. Depending on the type of workload and data object, different namespace implementations may be needed.

1.2.3 Optimal Performance of a Cluster

Most distributed platforms were designed to be general execute engines, providing powerful primitives to transform and analyze large volumes of data. These engines are well-suited for one-time jobs. However, it is becoming more common to use these platforms to perform repetitive and somewhat predictable tasks in specialized applications often involving structured data. For such workloads, there are wasted opportunities for optimization, such as indexing. These systems operate inefficiently and waste expensive compute resources.

The way these distributed frameworks were originally re-incarnated and improved upon was through identifying a certain problem and designing a new system that solves that problem. The new system's performance is then compared to the previous system to quantify relative improvement without knowledge of an upper bound. The process is incremental and can require multiple iterations to reach an optimal solution.

To demonstrate inefficiencies in current systems, in an experiment described in Section 5.7.1 a highly specialized and optimized program implementing a single analytical query was found to execute as much as 26 times faster than current state-of-the-art systems. The performance gap shows empirically that current systems are far from optimal even for simple workloads. This observation is consistent with previous works comparing the performance of platforms such as Hadoop and Spark against minimalistic implementations using MPI (message passing interface) [11, 25, 28].

What if we free ourselves from all architectural constraints and implement an optimized version of a given workload and consider that to be the upper bound of performance? We can then rely on both first principles thinking and previous knowledge to design a new system that supports an intuitive API (application programming interface). The goal of such an API is to provide primitives that the larger system can invoke. A successful system design in this case is one that can use the provided primitives to achieve comparable performance to the optimized implementation.

1.2.4 Proposed Solution

To address the issues discussed, this thesis proposes a modular system design that differs from current architectures in two main respects. First, the generalization problem is mitigated by fine-tuning and specializing system behaviour on a per-workload basis using pluggable modules for workload implementations. Second, the proposed design enables the coupling of execution mechanisms to establish efficient compute pipelines spanning across all resources of concern for a given workload. This allows any possible optimizations and/or coordination between these mechanisms to be implemented behind the scenes.

These two novel changes address all of the problems described throughout this chapter. An analytics application is now able to specify how its computation and storage may be realized by providing implementations of system modules that are only utilized for that particular workload. These modules control data representation, compute logic, serialization, physical query planning, logical view of the data, among others. The base system manages common functions, for which a single implementation is guaranteed to be suitable

for all workloads. Examples include synchronization, networking, resource management, fault tolerance and recovery, etc.

The tradeoff of this approach is the substantial development effort required to implement many system modules. This development effort can only be justified in use cases involving repeated analytical tasks and long-term data warehousing. For short-term storage and/or one-time jobs, existing systems such as Hadoop and Spark are more suitable.

1.3 Thesis Statement

This thesis proposes that a modular distributed system design which tightly couples **storage**, **memory**, and **compute** resources can largely improve the performance of analytical queries on structured big-data when compared to existing general-purpose frameworks. The resource coupling is achieved through a distributed data object (DDO) interface which gives the application developer control over key node-local execution mechanisms. The modular design allows many system components to be specialized and fine-tuned on a per-workload basis. The core component of the system is a unified storage and execution engine which manages common functionality such as networking, scheduling, fault tolerance, etc. Using the TPC-H [35] and TPC-DS [36] benchmarks, this thesis shows that the proposed design can outperform systems such as SparkSQL and Apache Hive.

1.4 Scope

The goal of this thesis is to lay the foundations of a system design for efficient distributed analytics. This is achieved through careful examination of both the inefficiencies in current systems and the empirically established upper bounds of a few select workloads. The process yields a set of requirements for the design. Most of these requirements could be satisfied by using a modular system design as explained. Furthermore, a data object design that couples different execution mechanisms allows efficient use of node-local resources.

To demonstrate the benefit of coupled execution mechanisms under a single data object, consider the example of a list of records indexed by an attribute α and written to a file f. If an analytical task needs to perform an operation on records having $\alpha \in r$, such that r is a known interval, then the index of α can be used to physically locate r in f. The storage-controlling mechanism would only read the region of interest from f. On the other hand, the compute-controlling mechanism can take this information and skip the "filter" operation which would otherwise have been performed. Hidden internal optimizations and cooperation between all the different mechanisms allows workload implementers to create efficient compute pipelines. In comparison, this is not possible using HDFS due to system separation and the fact that the storage system only deals with entire blocks.

This kind of optimization is not new and has been used before in specialized applications such as relational databases. In this thesis, the concept has been generalized and brought to distributed systems. The evaluation

done in this thesis only examines structured relational data and SQL (structured query language) workloads. It is clear, however, that this applies to cases involving any structured data that can be indexed and workloads that have a tendency to access limited regions of large collections.

The term "structured data" usually refers to data that adheres to a predefined data model, which defines how the data entries relate to each other and what type of information can be found within each entry; e.g. a data model may require that a *sales* record will always have a pointer to an *item* record, quantity of the sold item, and the total price. Data models will usually enforce constraints on data records such as having not-null key fields. Constraints such as these can be used to build record indices to improve the data warehouse's performance in extracting ranges of records. This thesis does not investigate the case of semi-structured and unstructured data. However, similar optimizations may still be applicable and other improvements, such as block co-location, can result in speed-ups over current distributed storage systems.

1.5 Organization

The rest of the document is organized as follows. Chapter 2 covers the background, presenting important design considerations and exploring related works in the literature. Chapter 3 presents the DDO interface and details key design elements of a system that supports DDOs. Chapter 4 discusses a proof-of-concept implementation: the Relation DDO, a DDO created specifically for relational data warehousing and analytics. Chapter 5 gives the experimental results and compares with multiple baselines. Finally, Chapter 6 summarizes the findings of this work and discusses future work.

2 Background and Related Work

This chapter presents the background and some of the most notable related works. Sections 2.1 through 2.4 present the background material on big data processing. Section 2.5 briefly discusses resource management. Shifting focus towards distributed data warehousing in particular, Section 2.6 discuses notable efforts in this area. The concept of coupling different resources to allow some optimization or performance improvement was explored in Section 2.7. Finally, Section 2.8 concludes the chapter and identifies the main goals to be realized in this thesis.

2.1 Mainstream Big Data Processing

The origins of almost all modern big data applications can be traced back to two notable works in the early 2000's, namely, the Google File System (GFS) [19] and the MapReduce programming paradigm [15]. However, these two works remained closed-source for a few years until the Hadoop MapReduce and the Hadoop Distributed File System (HDFS) [31] open-source implementations became available. In order to understand how to design an entirely new system, one must understand why these monoliths of big data remained an industry-standard almost two decades after their inception. It also means that challenging their design must be approached with care.

It is a few key characteristics of these systems that allow them to remain the cornerstone of big data processing. Reliability, scalability, and intuitive parallel processing abstractions are some of the most notable. These systems are fault-tolerant to both persistent and transient data losses. For losses of persistent data, replicas are often employed to decrease the likelihood of data loss. As for transient data (temporary data between compute steps), recomputation and checkpointing are often used. Another key characteristic is scalability. Both Hadoop MapReduce and HDFS were shown to be scalable up to a few thousand nodes in multi-tenant clusters [37]. Finally, having intuitive parallel processing abstractions is a characteristic that has led to widespread adoption of MapReduce and later frameworks. Application developers no longer need to worry about the intricate details of synchronization and networking. By limiting the developer's concern to application logic, a wide range of developers from different backgrounds are more likely to collaborate on processing and analyzing multi-terabyte datasets. Almost all modern distributed computing frameworks offer some form of high-level parallel processing abstraction, such as Spark,[1] Storm,[2] and Flink,[3] to name a

[1] Apache Spark. `https://spark.apache.org/`. Accessed November 30, 2020.
[2] Apache Storm. `https://storm.apache.org/`. Accessed November 30, 2020.
[3] Apache Flink. `https://flink.apache.org/`. Accessed November 30, 2020.

few.

While all of the previous examples were concerned with the key characteristics mentioned, very few have examined the upper limits of performance and system efficiency. When it comes to measuring performance, the process is always comparative to previous systems in an attempt to quantify improvement. In most cases, this is done without full investigation of the reasons why these systems were able to be improved in the first place. Many works in the literature often identify one of many bottlenecks in current designs and address that particular bottleneck only, leaving many other architectural inefficiencies. The upper bound of performance is unknown and incremental improvements seem to be always possible. This thesis advocates an approach in which an upper bound is first identified and later used to drive the design and development process of the new architecture. Comparing to the upper bound also gives an estimate of how well the system design achieves optimal operation and whether it has acceptable overheads. In the following sections, relevant works from the literature will be examined in more detail to understand important design issues, identify possible areas of improvement, and note the proven successes of current systems.

2.2 Distributed and Massively Concurrent Programming Models

2.2.1 Beyond MapReduce

Dryad

Dryad is a general purpose execution engine for distributed data-parallel programs [23]. It offers a programming model that describes computation and data dependency using directed acyclic graphs (DAGs). Programs developed for this model are called "vertex programs". The runtime system can optimize the graph and distribute computation efficiently as a result of the exposed data dependency. The design is mainly influenced by GPU shader languages, MapReduce, and parallel databases. The authors argue that the success of these influential systems comes from the fact that they force the developer to think about data parallelism. Dryad runtime deals with vertex programs that expose parallelism opportunities through a description of data dependency. The runtime system can parallelize execution using available resources and cover faults. Experimental results show near linear scaling with increasing number of nodes.

An interesting approach to measuring performance was taken in this study. The running time of a SQL query running on SQLServer was compared with an equivalent Dryad vertex program that accesses data and indices the same way as the baseline. This comparison was only viable for the single node case, as SQLServer does not support distributed execution. The running time of Dryad was largely better than that of SQLServer. The authors attribute this to the fact that SQLServer supports logging, transactions, and mutable relations, all of which may hinder its performance when compared to an execution engine free of these requirements.

Apache Spark

Spark is a general-purpose data processing framework built on top of the RDD abstraction [44]. Implementing non-trivial logic and/or iterative algorithms in MapReduce would often require multiple successive stages, each having their *map*, *shuffle* and *reduce* steps. The inputs to *map* tasks can only be read from the distributed file system, and outputs of *reduce* tasks are always written to the file system. Zaharia *et al.* proposed a method for caching intermediary results using distributed memory. This approach pipelines compute stages and avoids writing data to persistent storage.

The RDD abstraction couples both distributed memory and distributed compute resources and offers a wide variety of compute primitives (including the typical *map* and *reduce* primitives) as well as memory control primitives such as *cache*. More complex primitives implementing commonly used functionality such as *filter* and *groupBy* are also provided by the RDD. Compared to MapReduce, Spark can be as much as 100 times faster due to its use of distributed memory and overall improved efficiency.

Alongside the low-level RDD API, Spark also offers higher-level APIs (DataFrames and DataSets) and libraries for a variety of use cases including SQL, stream processing, graph processing, and machine learning. In the recent years, Spark has undergone many modifications and improvements. The development has been driven by the community at-large as well as the original creators and individuals from several major corporations.[4]

Apache Tez

Tez is a unifying framework for developing distributed data processing engines [29]. Tez is not an execution engines itself. Instead, it provides a library layer that can be used to build data processing engines of various concerns. This alleviates the burden of implementing resource negotiation, fault tolerance, task monitoring and coordination, etc. with every new engine.

The framework offers a DAG API that allows developers to define the computation using a directed acyclic graph in a manner similar to (and largely inspired by) Dryad. In a Tez DAG, the edges which represent data dependencies (and also data movement) can be configured to perform broadcast, key-value based shuffle, one-to-one mapping, or any custom user-defined routing rules. The vertices are composed of three sub-components: the inputs, the processor, and the outputs. The processor defines the actual code to be executed. The input and output classes are defined by the incoming and outgoing edges. Tez offers an abstraction called the vertex manager which allows currently running vertices to reconfigure future vertices and their payloads (including the processor code). This allows engines running on Tez to adapt their physical execution plan mid-job and make decisions based on information that is only available during execution.

Evaluating Tez depends on the workload and the engine implementation being run on top of the framework. Performance tests of Hive on Tez using workloads derived from TPC-H [35] and TPC-DS [36] bench-

[4] `https://spark.apache.org/committers.html`. Accessed May 11, 2021.

marks show significant speed-ups compared to Hadoop MapReduce. Pig[5] on Tez was tested on a cluster of 4200 servers at Yahoo!. The performance results showed 1.5 to 2x improvement compared to Pig on Hadoop MapReduce.

Comparing a typical Spark on YARN distribution and an implementation of Spark on Tez reveals the importance of fine-grained ephemeral resource allocation in a multi-tenant environment. Tez allocates smaller and short-lived YARN containers for Tasks while Spark allocates resources to run its VMs (virtual machines) for the entire duration of the job. While there may be an increased overhead (delays due to resource negotiation and queueing in YARN) in the approach taken by Tez, the price of having allocated unused resources may be too great in some multi-tenant use cases.

2.2.2 Actors, Message Passing, and Function Passing

Message passing interface (MPI) has been widely used in high-performance computing applications [32]. Programming complex data processing applications at the transport layer is arguably still a huge effort. With MapReduce and later systems, there are two main advantages over MPI: high-level data processing abstractions, and fault tolerance. Many of the current compute engines, which provide those two advantages, are built on top of MPI themselves. For instance, Spark heavily relies on Akka,[6] a message passing library for Java and Scala.

Haller and Odersky developed an actor implementation in Scala that aims to unify both thread-based and event-based actors [20]. The implementation relies heavily on Scala's extensibility and rich feature set; in particular, partial functions and advanced pattern matching capabilities. Compared to SALSA,[7] a Java-based actor language, the throughput of this implementation was found to be 20 times higher. This allows for performant implementations of massively concurrent systems on the mainstream JVM (Java virtual machine) platform.

Miller *et al.* proposed a model for distributed functional programming called "function passing" [27]. The model allows sending function closures to nodes where corresponding data resides. The model can be seen as a generalization of many modern execution engines such as Spark. Data is stored in containers called *silos* which have handles–called *SiloRefs*–to allow the programmer to access the data within them. *SiloRefs* can point to either local or remote silos, and even silos which have not yet been materialized. The serializable function closures operating on *SiloRefs* are called *spores*. A spore is made up of two serializable parts: a header and a body. The header carries state data while the body contains code that performs some computation. Fault recovery is realized in this model by associating silos with lineage DAGs. Any lost silos can be recovered by reapplying the transformations in the lineage DAG. The repeated computation must begin from input silos originating from reliable storage. An implementation of this system in Scala was

[5]Apache Pig. `https://pig.apache.org/`. Accessed April 15, 2021.
[6]Akka. `https://akka.io/`. Accessed May 4th, 2021.
[7]SALSA. `https://wcl.cs.rpi.edu/salsa/`. Accessed April 7, 2021.

developed and miniature examples of Spark RDDs and the MBrace framework[8] were tested.

2.3 Reliability

An ideal distributed system appears to its users as a single logical entity; i.e. the distribution is hidden from the users of the system [33]. However, partial failures are quite common in distributed systems. A fault-tolerant design can continue to operate efficiently while automatically trying to recover failed nodes. The single entity view of the system isolates the user from the internal partial failures the system may have. The goal is often achieved by using redundancy.

Node failures can be transient or permanent. A transient failure is a failure which can occur due to bugs in the system implementation, bugs in libraries, spurious hardware behaviour, etc. and can be mostly recovered from automatically by restarting the process and/or node. Permanent failures are usually triggered by a failing or misbehaving hardware components. This failure can be catastrophic leading to node unavailability, or it could greatly impact the performance of a node rendering it largely unavailable; e.g a failed drive in a RAID (redundant array of inexpensive disks) unit, or firmware updates causing hardware to misbehave. Permanent failures are assumed to be handled by the system administrator. In both cases, there is a measurable mean-time-between-failures (MTBF) for different classes of nodes which can be used to model the system and calculate the overall rate of failure.

The system design proposed in this work is mainly concerned with performance. There is no particular effort done to improve the reliability of a distributed big data system. In this section, a selection of fault-tolerant designs will be briefly examined.

Ghemawat *et al.* first proposed a method for sharding and replicating files into fixed-size data blocks [19]. Data nodes are considered to be unreliable commodity machines having fairly limited compute and storages resources. A more reliable and highly available *name node* keeps track of all data blocks in the cluster and makes decisions using global information. The *name node* is usually a more reliable, more powerful machine. However, the *name node* remains a single point of failure and a potential bottleneck. Shvachko *et al.* reused the same architecture in the open-source HDFS [31]. Shvachko *et al.* have shown that the system can have a secondary *name node* that is mostly dormant and is only activated in the event the primary *name node* is unavailable. During its normal operation, the secondary *name node* mirrors the operation log of the primary *name node* to keep track of all blocks, but does not make any decisions or instruct *data nodes* to take any action.

The design of GFS and HDFS proved to be reliable and fault-tolerant, albeit some concerns (such as the total number of files/directories) that may exist. The reliability of the entire cluster depends on one or two special nodes. The architecture is also not applicable to clusters not having a performant, reliable node to be trusted as *name node*/master.

[8]MBrace. `http://mbrace.io/`. Accessed April 15, 2021.

Behera *et al.* studied the concept of failure prediction and proactive migration to avoid recomputation [7]. In addition to periodic checkpointing, the system uses a failure prediction model to assess the possibility of an imminent failure and predict *lead time to failure*. If the predicted lead time to failure is sufficient to perform a live migration, the computation is migrated to one of a set of reserved idle nodes. The failed node should then be restored and added to the set of reserved nodes. Moreover, local *burst buffers* are used to improve checkpoint performance. Burst buffers allow fast checkpointing which are later written asynchronously to reliable distributed storage. Results show that checkpoint time was reduced by $\approx$ 30%-82% due to the use of burst buffers. The use of failure prediction and migration reduced the recomputation time by $\approx$ 51%-56% for applications with large checkpoint sizes and $\approx$ 13%-85% for applications with small checkpoint sizes. As most modern systems use recomputation to recover from task failures [23, 27, 29, 43], this approach may prove to be a valuable addition to big data processing systems.

2.4 Scalability

The ability to handle growing volumes of data and increasing numbers of users is one of the main design concerns of big data systems. Bondi's definition of load scalability puts focus on two main aspects: (1) managing shared resources, and (2) efficient resource consumption [9]. Assuming optimal resource management and allocation, efficient resource consumption is only dependent on the efficiency of the compute and storage frameworks. As hardware continues to evolve, the design requirements of these frameworks and certain bottlenecks of concern are changing as a result. This section investigates the scalability of common system designs, identifies problems in current systems, and speculates possible solutions.

2.4.1 Compute Efficiency

Bakratsas *et al.* performed an empirical study to evaluate the performance of HDFS and MapReduce on solid state drives (SSDs) [6]. The purpose of the study was to find whether upgrading servers to use SSDs rather than hard disk drives (HDDs) would have a significant impact on the performance of a graph processing application using Hadoop MapReduce and HDFS. For *TestDFSIO*, an HDFS throughput benchmark, the results perfectly reflect the throughput difference between SSDs and HDDs. However, the graph processing workload saw minimal improvement even for map tasks which are only dependent on the I/O throughput of HDFS and the compute throughput. Bakratsas *et al.* also noticed an increased variance in task times in the SSD case.

It can be argued that the main bottleneck found by Bakratsas *et al.* was due to overheads imposed by the Java implementation of Hadoop. Shaffer *et al.* confirmed this in an earlier study [30]. In particular, they noted three problems that hinder the performance of the Hadoop architecture:

1. **Software design bottlenecks** due to inefficient compute pipelines. Compute and I/O invocations are serialized, resulting in scheduling delays. Instead, asynchronous I/O operations could have been

implemented to improve I/O and CPU utilization,

2. **JVM architecture limitations** due to missing native filesystem features which can be used to improve performance, and

3. **Native filesystem behaviour assumptions** which exist in the implementation of HDFS despite the fact that the JVM architecture was chosen for portability.

Many popular big data systems currently operate using JVM bytecode, including Spark, Hadoop, Tez, and HBase. With increasing I/O speeds, the need for more efficient compute pipelines is starting to be an issue of concern in big data systems. This may shift the attention towards developing new engines written using more performant languages like C. Drocco *et al.* make an argument for modern distributed programming in C++ [17]. They demonstrate strongly scalable implementations of distributed STL (standard template library) containers. While this approach to distributed systems has only been largely used in production scale closed-source software, the need to leverage the performance offered by new I/O technologies may prompt bringing this implementation philosophy to the world of open-source big data.

2.4.2 Replica Setup

The centralized *name node* of HDFS keeps namespace and block placement information at a central location (with an optional secondary backup) which greatly simplifies the design, but can lead to scalability issues, or at least certain scalability concerns. Furthermore, HDFS replicates blocks using a replication pipeline, starting at the primary replica. Once the pipeline is initiated, each node receives the data block, verifies the checksum, then passes the data block down the link to the next node. This can lead to long replica setup delays [40].

To reduce the replica setup delays in HDFS, Zhang *et al.* propose an asynchronous multi-pipeline data transfer method in an improved version of HDFS which they call SMARTH [45]. A SMARTH client only waits for the first *data node* to receive the block. After that, the client initializes a new pipeline to other nodes to send the next block. Each of the pipelines is kept open until the data is transferred successfully to all nodes. Moreover, SMARTH's *name node* monitors each *data node*'s performance (network and disk I/O bandwidths), and ranks the data nodes in real time. This information is used to ensure that the first *data node* in a pipeline is the fastest. HDFS is known for its lack of heterogeneity awareness and usually any effort to address this design problem can improve performance in heterogenous environments. Sources of heterogeneity can include differences in hardware, resource contention in virtualized environments, or bandwidth limitations between different server racks. Experiments show that SMARTH can speed up data upload by 27-245%, depending on cluster configuration and bandwidth heterogeneity.

2.4.3 Centralized vs. Decentralized Catalogs

In both the GFS and HDFS, the *name node* needs to keep all block information in memory. For every file, directory, and block, there needs to be an in-memory record [19, 31]. The GFS implementation uses 64-byte records while the HDFS implementation uses 100-byte records. It can be argued that the system scalability is limited by the amount of memory available at the *name node*. In practice, however, this limitation is usually not a concern if the system houses a reasonable number of large files.

In use cases which involve a large number of small files, the files are often grouped together in larger archive files, such as the Hadoop Archive format.[9] This adds an extra layer of complexity and latency required to look up a file. Other works proposed solutions for handling small files in HDFS with varying degrees of success [22, 26, 39].

Al-Kiswany *et al.* proposed a key-value storage system called *NICE* that uses software-defined networking (SDN) to utilize network resources efficiently [2]. In *NICE*, the records are scattered across a virtual hashing ring of nodes that is mapped to a physical ring of nodes using software-defined networking. The mapping is maintained by a special node called the metadata node. The metadata node uses the OpenFlow standard[10] to configure SDN rules at the switches. Replication is performed using network-level multicast groups. Multicasting also has the benefit of eliminating replica setup delays entirely. The separation of virtual and physical rings offers flexibility to scale the system easily for different size physical rings. Moreover, the performance requirements of the metadata node are much less than that of the *name node* of HDFS, since it is only concerned with updating routing rules when needed.

2.5 Resource Management

The system design proposed in this work should support multiple concurrent users. The hardware resources that this system uses may also be a part of a shared resource pool used by other users who may not use the system. This section reviews some of the mainstream and highly scalable resource management solutions in order to integrate with existing mainstream resource managers as well as support multitenancy within the system.

YARN is a shared-cluster resource manager [37]. It consists of a per-cluster *resource manager*, a set of running *application masters* (one for each application), and many *node managers*. The *application masters* send one or more requests describing the number of containers, resources per container, locality preferences, and request priority. The resource manger tries to schedule and find an optimal set of nodes to fulfil the application master's request. The *resource manager* is not concerned with per-application local optimizations. This burden is left to the *application master* itself. Once resources are allocated, the *resource manager* replies to the *application master* with leases for nodes. Resources can also be revoked in a graceful manner via

[9] `https://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html`. Accessed January 29, 2021.
[10] `https://opennetworking.org/software-defined-standards/specifications/`. Accessed January 21, 2021.

preemption. YARN was implemented as a successor to the Hadoop 1.x JobTracker which was only limited to MapReduce jobs. In a large-scale deployment on a 2500 machine cluster, YARN's peak number of daily jobs was almost double that of Hadoop 1.x. Average CPU utilization also increased from 20% (3.2/16 cores) to almost 37.5% with peaks up to 62.5%. Furthermore, YARN has many pluggable scheduler and resource calculator implementations which allow flexible reconfiguration for a variety of use cases.

Mesos is a fine-grained decentralized scheduler which uses a novel abstraction called *resource offers* to achieve decentralized near-optimal resource management [21]. Hindman *et al.* argue that optimal resource scheduling in a shared cluster is a difficult problem to solve due to the amount of jobs and tasks in a cluster of a few thousand nodes. The scheduling policies of individual computing frameworks are also continuously evolving, making the design requirements for a global resource manager unclear. Mesos takes a different approach by offering resources to frameworks based on organizational policies (such as fair scheduling) and the frameworks would accept only the resources they need for a given job. It was found that this approach, albeit seemingly naive, is in fact near-optimal in practice. Mesos consists of a per-cluster *master process* and a per-node *slave daemon*, similar to YARN's *resource manager* and *node manager*. Frameworks using Mesos would need to implement a *scheduler* and an *executor*. The *scheduler* registers with the master to accept resource offers while the *executors* are the actual processes launched on the slave nodes. The evaluation done compares Mesos with static partitioning only. However, it was found that the *resource offers* abstraction resulted in near-optimal data locality for the individual frameworks.

Hindman *et al.* note that at the time of writing, YARN was still in development and not many details were available. Both studies confirm that a decentralized approach to fine-grained resource management is sufficient even in a cluster of thousands of nodes. Nearly a decade later, these two approaches remain industry standards when it comes to cluster resource management.

In a more demanding setting, a robust resource manager called Borg was developed at Google [38]. Borg is mainly a cluster resource manager but can be considered as a cluster operating system. Borg manages collections of nodes called *cells* within a data center. A large-scale data center will typically occupy a few buildings, each housing many cells. The median cell size managed by Borg is roughly 10,000 nodes. Borg supports two types of workloads: long-running services that are always up, and batch jobs. Aside from resource management and allocation duties, Borg also installs applications and dependencies, monitors application health, and restarts applications if necessary. The ideas implemented in Borg are not fundamentally different from those of YARN or Mesos, but they are more specialized and running at a massive scale.

2.6 Distributed Warehousing and Analytics of Structured Data

Armbrust *et al.* proposed a *DataFrame* API that combines both procedural and relational styles [4]. The API is now the central component of SparkSQL, and in the recent years it has been a well-established component of the Apache Spark software package. The DataFrame API evaluates operations lazily making it possible

to implement automatic relational optimizations. The query optimizer, *Catalyst*, is primarily a rule-based optimizer with some cost-based optimizations applied at later stages. It is written in Scala and makes use of Scala's advanced pattern matching features.

Catalyst uses trees to represent query plans. Using pattern matching rules, it is able to transform the query tree iteratively until no more patterns can be matched. One interesting feature of the Catalyst query optimizer is that it can be extended with data sources optimizations that push down operations such as filtering and aggregation to the data sources themselves. This is particularly useful when integrating with a warehousing system such as HBase or Hive. However, this only applies to the leaf input nodes. Once execution begins and the data is in Spark's memory, no further optimizations can be done. Catalyst has to guess the size of intermediary results and make all execution-related decisions in advance. This thesis will demonstrate that it is possible to optimize I/O during execution and make use of helping structures such as record indices to opportunistically speed up relational operations.

Hive is a data warehousing system built on top of Hadoop [34]. The system organizes data into tables and columns. The data may be partitioned and/or bucketed by any column. The core components of Hive are a *query compiler* which parses and interprets a query statement written in HiveQL, a *metastore* which stores object table metadata, and an *executor* which submits tasks to the execution engine. Originally, Hive was limited to using Hadoop MapReduce as its execution engine. It is now possible to use other execution engines, such as Spark or Tez. Hive was the first distributed big data SQL solution that could operate on multi-terabyte datasets using commodity clusters.

Costa *et al.* investigated the use of partitioning and bucketing strategies in Apache Hive [13]. They found that the use of partitioning proved to be advantageous and reduced the query time by about 40%. The use of bucketing did not show any significant improvement in query time. This was partly attributed to lack of a join algorithm utilizing record buckets. One would normally expect that a structure somewhat comparable to a sparse record index would speed up queries by a large factor. However, the authors show the contrary, with only 40% improvement in query time. While this may be attributed to lack of features, it also highlights the fact that the architecture is incapable of matching the efficiency of decades-old traditional data warehouses that utilize indices more efficiently. There is no reason to believe that this type of performant and efficient system cannot also be distributed.

Bigtable is a distributed structured storage system built on top of the GFS [10]. The data is organized in three dimensions: row, column, and timestamp. Unlike a typical RDBMS (relational database management system), both rows and columns are named in Bigtable and can be any arbitrary strings. Timestamps are 64-bit integers that can be used to store multiple values of the same cell. Groups of rows are called tablets and form the unit of distribution. The system is composed of one master server and many tablet servers, much like the *name node* and *data node* architecture of the GFS and HDFS. One interesting feature of this system is column *locality groups*. By controlling locality groups, certain columns can be co-located to minimize network transfer, or not co-located to improve concurrency. Bigtable was mainly designed to support random reads

and writes on structured tabular data. It also supports transactions on single rows. Apache HBase[11] is an open-source implementation of Bigtable and is built on top of Hadoop and HDFS.

2.7 Coupling Compute, Memory, and Storage Elements

Compute and Memory: Workload-aware and Workload-controlled Memory Caches

The RDD concept introduced the idea of coupling compute elements with distributed memory [43]. This allowed RDD-based systems to cache intermediary results and avoid having to flush intermediary results to secondary storage. This resulted in nearly a 100 times[12] improvement of Apache Spark over traditional MapReduce. However, Apache Spark's automatic cache management policies leave room for improvement.

Yu *et al.* proposed a cache management policy called LRC (least reference count) [42]. They propose that lineage DAGs of data objects can be used to accurately predict data access patterns. The reference count for any arbitrary object is the number of child objects that have not yet been computed. Their better-informed cache eviction policy reduced application runtime by up to 60%, when compared to Spark's default LRU (least recently used) eviction policy. This makes a strong argument for coupling memory and compute elements under one system. Only the compute system knows the actual data dependency and can make better-informed decisions than any other external caching layer. However, this approach only applies to a single job and does not recognize frequently used data objects common in multiple (and possibly concurrent) jobs.

Memory and Storage: Caching Data Blocks

Coupling distributed memory and storage can mitigate the problem of frequently used data objects. Liu *et al.* investigated the idea of integrating HDFS with MemCached [24]. MemCached is a distributed memory object caching system. In the proposed architecture, frequently-used data blocks were replicated to a MemCached cluster. This resulted in up to 36% reduction in execution time. This can be attributed to two main reasons: 1) the dynamic nature of the replication strategy which can adapt to changing workloads, and 2) application-managed buffers are usually more efficient than operating system file buffers. However, the experimental testing in this work relied only on the *WordCount* and *Grep* examples to evaluate performance. Access patterns for more complex workloads were not evaluated.

Dong *et al.* proposed a correlation-based file and metadata prefetching method for HDFS [16]. The data prefetching method was designed for an internet service based on HDFS. Both data blocks and metadata are prefetched to (1) reduce I/O delay, (2) reduce network latency, and (3) reduce latencies due to client and *name node* interaction. Prefetched metadata can be stored at either the HDFS client or in a memory cache at the *name node* itself. The same can also applied to prefetched data blocks, which can be placed in a

[11] Apache HBase. `https://hbase.apache.org/`. Accessed December 21, 2020
[12] Apache Spark. `https://spark.apache.org/`. Accessed November 30, 2020.

memory cache at either the client or the *data node*. Combinations of these techniques can be used resulting in four different configurations. In a test cluster of 9 nodes, storing prefetched data blocks at the client and keeping metadata at the *name node* resulted in an average improvement of 1700% in job completion time for 4 or less concurrent clients. For 8 and 16 concurrent clients, performance improvements quickly dropped to 60%.

Compute and Storage: Workload/Hardware-aware Storage Management

Ciritoglu *et al.* proposed a replica management system on top of HDFS that is aware of node heterogeneity [12]. The system distributes data unevenly based on the available compute resources at each node. The average execution time in a heterogenous environment was reduced by nearly 40% when compared to the default HDFS distribution policy. The improvement over HDFS increases to 60% when multiple concurrent users are using the system.

Yu *et al.* proposed a method for grouping the blocks of single files in HDFS to improve co-locality and reduce off-switch data access [41]. Off-switch (inter-rack) networking in clusters is a limited resource and often impedes the shuffle performance of a MapReduce job. In the workload investigated by the authors, roughly half of the inputs to *reduce* tasks are fetched from off-switch data nodes. To mitigate this issue, the authors propose a method for storing all data blocks of a single file at a limited number of server racks. This is applied to one of the file replicas only. The other replicas are left untouched. This results in a trade-off between reduced off-switch access and parallelism. In particular, the authors describe a *sticky effect* and a *conflict effect*. The *sticky effect* reduces achievable parallelism by limiting the block locality to a few server racks. The *conflict effect* arises when multiple jobs compete for the same set of server racks. For *Sort* and *TextGen* workloads, the execution time was reduced by nearly 50%. The *sticky effect* can be largely mitigated by setting a suitable number of racks for grouped files such that the reduced parallelism does not overshadow the benefit of block grouping. Careful job scheduling can also mitigate the *conflict effect* to reduce competition for the same grouped files.

Data replication is used primarily as a fault tolerance policy. However, it can also be used for performance reasons. Highly available data can be used to mitigate "straggler" nodes by launching replica tasks. Straggler nodes are slower-than-average nodes that can (unexpectedly, due to partial failures or other events) prolong the execution of a distributed application. Replication can also be used to improve concurrent access in multi-tenant environments.

Using replication to mitigate straggler node performance in master-worker architectures has been examined using analytical methods [8]. Behrouzi-Far and Soljanin define a diversity-parallelism spectrum for data replication. **Full data diversity** means that every node in the system has a copy of the entire dataset. **Full data parallelism** means that every node only has a part of the dataset that does not exist elsewhere. Within this spectrum lies an optimum operating point. The analysis was done using two service time distribution models: exponential and shifted-exponential. Examining an exponential service time distribution shows that

full data diversity minimizes both the variance and the expected job completion time. For shifted-exponential service time, minimizing the expected job completion time requires solving a discrete unconstrained optimization problem. The minimum variance in this case is also achieved using full data diversity. The main point to be taken away from this is that redundancy was shown to reduce variance in job completion time. Although the theoretical minimum variance can be achieved with full data diversity, it has been found, both analytically and empirically, that a replication factor of 2 or 3 can significantly shorten the long-tailed distribution.

Scarlett is a system for managing replication to mitigate skewed content popularity in HDFS clusters [3]. The use case described by Ananthanarayanan *et al.* involves skewed and continuously varying file popularity. Scarlett predicts file popularity and proactively mitigates it by controlling the replication factor for individual files. This can result either in increased or decreased replication based on the predicted popularity. Due to the increased availability of popular content, job completion times were improved by roughly 20% at the median and 44% at the 75^{th} percentile. Scarlett also uses data compression to reduce the network overhead of data replication at the expense of a computational overhead. Using data compression, the network overhead caused by data replication was reduced from 24% to 0.9%.

2.8 Discussion

In this chapter, the design, strengths, and weaknesses of classical and notable works in the literature were reviewed. Shifting focus to systems that support warehousing and higher-level query languages, the ability to improve performance by some form of resource coupling or data access pattern prediction was observed in multiple works. In particular, the following themes are recurrent in literature:

- Tightly coupling memory and compute primitives to communicate transient intermediary results through memory rather than storage,
- Improving the compute engine's in-memory object cache management using better informed eviction policies and/or more accurate predictions of data access patterns,
- Augmenting the storage element with a cache layer that can store frequently used data blocks and capture longer dependency chains,
- Organizing the data and/or file structure to avoid full data scans when possible, and
- Utilizing replication to improve performance through increased availability and straggler mitigation.

The performance improvements gained by coupling memory and storage should not be surprising. The same applies to compute and memory cache components as demonstrated by RDDs. After all, storage is just another layer in the memory hierarchy. In cases where the data size is larger than memory, which is often the case with big data, the main performance bottleneck are the limited bandwidths of storage and

networking components. Efficient scheduling and utilization of I/O resources is mandatory for larger-than-memory datasets.

A system that plans and executes the compute workload can communicate fine-grained data access requests to different levels of the memory hierarchy; i.e. caching needed blocks (from lineage) in memory, and have access to fine-grained disk (and network) I/O operations. Many previous works have examined these ideas [4, 24, 34, 42, 43]. Prior work has either used special features that allow communicating certain hints to minimize data access, or a clever caching component that can discover access patterns. Pushing filters to smart data stores [4], query pruning by eliminating unneeded data partitions during query planning [34], and evicting cache blocks based on lineage [42] are some of the most notable efforts. However, all of these attempts were added features to already established systems. There has never been an interface designed primarily for this purpose.

This thesis outlines a design framework for distributed analytics systems that offers two main advantages over the current state-of-the-art:

1. A modular system design that allows workload implementations to introduce specialized system behaviour for their jobs, and

2. A data object interface that can minimize data access requests across all levels of memory hierarchy, based on a given query plan.

3 Detailed System Design

This chapter presents the distributed data object (DDO) interface along with a detailed description of all components of the base system. The objective is to design an efficient system that runs analytical workloads on structured big data. The architecture is largely inspired by well-established distributed computing frameworks, such as MapReduce and Apache Spark. However, the work is also influenced by many ideas from the literature and evidence pointing to weaknesses and/or areas of improvement in those frameworks as discussed in Chapter 2. It is worth mentioning that the system presented here is not a replacement for an on-line transactional data warehouse. The system is only intended to be used for analytics and storage purposes.

Section 3.1 discuss the assumptions of the system design and overall scope. Section 3.2 presents the DDO module. Sections 3.3 through 3.6 detail important components of the base system. Section 3.7 presents the DDO controller module which allows fine-grained block placement. Job planning and execution is detailed in Section 3.8. An approach to fault tolerance, largely inspired by current systems, is explained in Section 3.9. Section 3.10 describes the data ingest procedure. Section 3.11 proposes possible client designs. Finally, Section 3.12 summarizes the architecture presented in this chapter.

3.1 Design Scope and Assumptions

The system is designed to run on a cluster of server nodes interconnected using a high-speed and low-latency local area network. The nodes can be organized into server racks and have limited rack-to-rack bandwidth. The cluster hardware is assumed to provide both compute and storage facilities; i.e. each node in the system must contribute computation and storage resources to the overall system.

The capabilities of the cluster nodes can vary greatly from simple commodity machines to high-end servers. The system design is intended to cover both cases reasonably well and even support heterogenous clusters. The system design also assumes that network I/O bandwidth is typically more constrained and is a slightly more valuable resource than that of disk I/O.

3.2 Distributed Data Objects

The system stores and processes a distributed collection of data objects called *DDOs*. Each DDO is split into parts that are ordered and given sequence numbers. These *DDO parts* are distributed over the cluster nodes

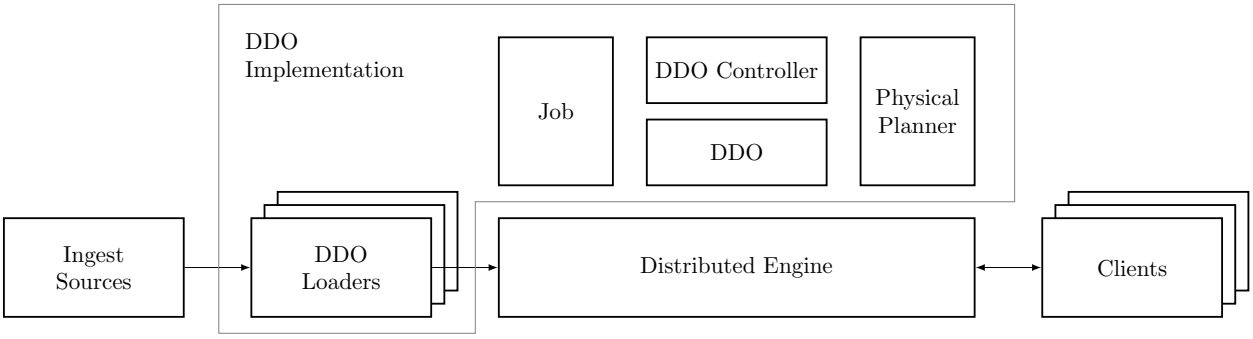


Figure 3.1: Layout of the system architecture. The DDO library contains implementations of multiple interfaces that are plugged into different parts of the system.

and replicated for fault tolerance and performance purposes. The DDO is designed to be a pluggable component. One instance of the system can house multiple different implementations of DDOs, each concerned with a given analytical workload. Figure 3.1 gives a high-level view of the system layout showing the DDO and other modules that will be discussed thoroughly in the following sections. Implementing DDO libraries is discussed in further detail in Section 4.2.

The DDO is largely inspired by the resilient distributed dataset [43]. The RDD concept introduced primitives to leverage the use of distributed memory. This greatly improved the performance of an RDD-based system such as Apache Spark over previous MapReduce systems, which used the file system for intermediaries. Similarly, the DDO provides primitives to utilize distributed memory, but the primitives presented here are slightly different. These memory-related primitives were modified to fit in with the rest of the system components. The RDD, however, lacks primitives for leveraging distributed storage. Other distributed architectures, including Apache Spark, rely on external data sources. The architecture proposed in this thesis tightly couples both storage and memory directly with the data object. This gives the DDO many advantages such as managing its own data representation, indexing, serialization, etc. as shall be discussed in the following sections.

3.2.1 Memory Primitives

A DDO part offers three different groups of primitives: memory primitives, storage primitives, and compute primitives. Memory primitives provide an interface for manipulating the object's memory. The object can be persisted permanently or temporarily on secondary storage, but it needs to have its data in memory before invoking compute primitives. To allow such control, the object must provide the following interface:

- load([LoadingHints]),
- unload(),
- pin(),
- unpin(),

- increaseReferenceCount(), and
- decreaseReferenceCount().

The method *load()* ensures that the object's data is in memory, after which a compute primitive can be safely invoked. Conversely, *unload()* frees up the used memory and writes the object data to secondary storage, if needed. Some objects are dropped from the system without being ever unloaded. This is true for small intermediary results that fit in the worker's memory.

Moreover, the method *load()* may be given an optional parameter called *LoadingHints*. A LoadingHints object indicates loading preferences; their actual meaning depends on the type of DDO. In fact, each DDO implementation can extend the LoadingHints class. One particular use case for this feature is selectively loading a slice of a DDO part. This is one of the main advantages of this system. By allowing the analytical tasks themselves to specify the loading preferences directly to the method responsible for I/O operations, it is possible to shorten long I/O waits by eliminating unneeded data access.

The methods *pin()* and *unpin()* usually surround compute primitives to prevent the object from being unloaded while a compute operation is running. Atomic variants that combine the primitives *load()* and *pin()* may be implemented, such as *loadAndPin()* and *unloadIfUnpinned()*.

The methods *increaseReferenceCount()* and *decreaseReferenceCount()* are used to indicate that the object is being referenced by another object. This can be useful in cases where a compute primitive produces a DDO that is a shallow copy of the base DDO. The new DDO does not own a full copy of the data. Instead, the new DDO references its ancestor and functionally defines a way to view the altered data. A reference count greater than zero signifies that object may be removed from memory, but not dropped entirely from the system because some other object still depends on it. It should be noted that shallow DDO copies, when loaded, automatically trigger *load()* on their parent DDOs. The same is true for *pin()* and *unpin()*.

3.2.2 Storage Primitives

The second group of primitives are the storage primitives. These primitives allow the larger system to instruct the DDO part where to place its files when it is unloaded. These primitives are as follows:

- path(),
- path(String),
- deleteFiles(), and
- reconstruct().

The methods *path()* and *path(String)* are used to get and set the storage path of the DDO part, respectively. These paths point to locations in the local file system of the cluster node where the DDO part can persist its data. The method *deleteFiles()* instructs the DDO part to delete its files. This is usually called

before the object is dropped from the system. Finally, *reconstruct()* is used to complete the creation of a previously-persisted DDO. On startup, persisted DDO parts are constructed using parameterless constructors and *reconstruct()* completes the construction of the object by reading its metadata file. After that, *load()* can be invoked when needed.

To clarify, consider the following scenario. To persist a DDO permanently, the DDO parts are first created in memory (possibly from a data ingest), then a storage location is assigned via the method *path(String)*. After that, *unload()* can be invoked. Once unloaded, the object can be destroyed. To retrieve the DDO parts again, one can simply create a DDO of the same type (using a parameterless constructor), set the storage path via *path(String)*, invoke *reconstruct()* to validate the integrity of the data files and read metadata, and finally invoke *load()* to load the data to memory again.

3.2.3 Compute Primitives

The last group of primitives are the compute primitives. The exact nature of these primitives largely depend on the type of workload and its needs. The DDO part is not required to conform to a specific interface for this set of primitives. However, there are a few guidelines that must be followed for correct system operation. The DDO parts are assumed to be immutable objects. During their lifetime (except during ingest and shuffle block merge), the DDO parts are expected to have immutable data. Compute primitives must always produce their output as a new DDO part. This removes the need for implementing concurrency control.

3.2.4 DDO Identification

Each DDO part is identified by a triple consisting of a unique identifier (*id*), a part number (*part_id*), and a lineage DAG. The unique identifier is a global identifier to group together all DDO parts of the same DDO as one logical entity. The part number is simply a sequence number. Much like the RDD, the DDO has a lineage DAG that describes all the transformations that were applied to obtain the current DDO. Together, all three components uniquely identify any DDO part in the system. This triple is referred to as the *DDO part descriptor*.

3.3 Tasks and Jobs

The *task* is the smallest unit of computation recognizable by the system. A task is defined by the following:

- A block of code to be executed,
- A set of input DDO part descriptors,
- A set of expected output DDO part descriptors (optional), and
- Configuration parameters.

The input and output sets are defined using DDO part descriptors and are used for task scheduling purposes. The block of code to be executed can safely assume that its inputs are loaded and pinned. The produced outputs are expected to be loaded and unpinned (have a pin count of exactly zero). The configuration parameters indicate to the run-time system of the execution preferences of this task. This is mainly for low-level optimizations and scheduling optimizations.

Groups of tasks working towards a user-specific goal are called *jobs*. Users connect to the cluster and submit a job to any node. The submitter of the job is called the *owner* of the job. The node that received the job will broadcast the job information and each node will produce tasks to achieve the job's goal. A job message usually contains a lineage DAG of the desired result. Each node will analyze the DAG and produce the tasks that will operate on the locally available DDO parts. It is worth mentioning that some of the produced tasks might depend on DDO parts sent from other nodes. These tasks remain blocked until the missing DDO parts arrive. This is explained in detail in Section 3.4.

A particularly important feature of this system is the ability to reorder and optimize tasks in a job. For instance, the system could reorder tasks to minimize the number of load/unload operations. Jobs can also share data. This is applicable in cases involving consecutive (or even concurrent) jobs using the same DDO parts. In a multi-tenant environment, the tasks of individual jobs may be interleaved and ordered in such a way to minimize thrashing and other expensive I/O operations. For instance, in the implementation presented in Chapter 4, the system may run tasks out-of-order to favor temporal locality. The reason behind this preference is that newly produced DDO parts are more likely to have their data in memory or even CPU cache than older DDO parts.

3.4 Workers

The system hardware is composed of a collection of networked, independent, and collaborating *nodes*. Each node may house one or more *workers*. A *worker* manages assigned quotas of storage, memory, and compute resources. The worker keeps track of all locally-stored DDO parts and caches frequently needed DDO parts in memory. In the implementation presented in Chapter 4, the worker uses a *least frequently used* policy for cache eviction. The worker determines the frequency of usage by counting the number of times a DDO part is pinned.

It should be noted that any other cache eviction policy may be used and it is not the scope of this work to determine the best policy. In fact, it may be better to implement a worker that supports multiple eviction policies and allows each workload to specify which one it prefers. These policies are only used in evicting seemingly-unneeded DDO parts; i.e. DDO parts that are no longer needed by any task (blocked or queued). In most of the cases, cache evictions are selected using more informed policies that determine future accesses based on the information from the input and output sets of queued and blocked tasks. This approach is similar to the method proposed by Yu *et al.* [42].

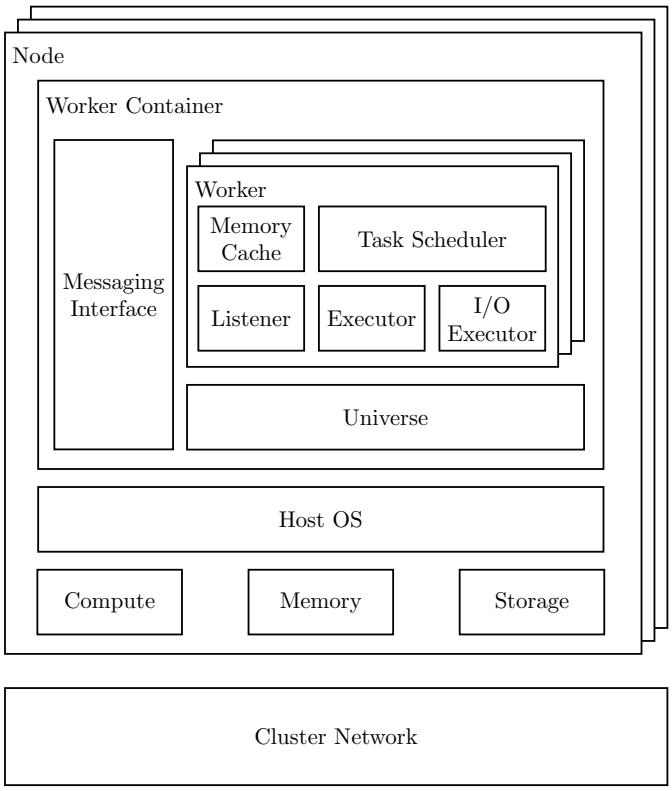


Figure 3.2: Layout of the DDO execution and storage engine. The engine serves as a compute and storage layer. The main goal of the engine is to support the DDO implementations. The components and worker organization shown is discussed in detail in the following sections.

3.4.1 Listener Thread

Each worker has a few long-running threads. One of these threads is the *listener* thread. The listener receives incoming messages and acts accordingly. Each message object has a visitor method called *process()* that implements the intended protocol or action that needs to be taken once the message is received. Depending on the system implementation, there may be numerous types of messages but the most important ones are *job messages* and *DDO part messages*.

Job messages usually contain a lineage DAG as described earlier, as well as the DDO type it uses. The DDO type will determine which physical planner to use to produce tasks for the given lineage DAG. The *process()* method of this message usually assigns a job ID to the message and broadcasts it to all other workers. Each worker will then (independently) pass the lineage DAG to the corresponding physical planner (based on the indicated DDO type), which will return a set of local tasks that can be submitted to the local worker to complete the job.

DDO part messages are used to transfer DDO parts from one worker to the other. This is how shuffling and/or broadcasting data takes place. Messages carry DDO parts and once they arrive the *process()* method registers the DDO part with the worker, potentially unblocking any waiting tasks.

3.4.2 Executor Thread

Each worker maintains a queue of tasks that are ready to execute (called the *ready queue*) and a set of blocked tasks (called the *blocked set*). Blocked tasks are tasks unable to execute due to one or more missing inputs. Once the inputs are available, the task is marked as runnable and is moved to the ready queue. These inputs may be locally- or remotely-produced DDO parts. Remotely-produced DDO parts arrive via a DDO part message.

The executor dequeues tasks from the ready queue and starts the execution of the task. The executor allows running a single task at any moment. During execution, tasks are given access to a pool of threads by the executor. The number of threads in the thread pool is determined by the thread quota given to the worker. The executor runs as long as the ready queue is not empty, or the listener thread is running (which may result in scheduling new tasks).

Some tasks are marked as *breakable*. Breakable tasks can run multiple times on disjoint subsets of the indicated input DDO parts. For instance, tasks that merge shuffle blocks can be marked as breakable to alleviate barrier synchronization delays. This is similar to the incremental reduction method described by Elteir *et al.* [18].

3.4.3 I/O Executor Thread

An optional component of the worker is the I/O executor. This is not required for system operation, but it is believed to be a valuable optimization that most implementers will prefer to have. Since the main compute

executor can only run one task at a time and that task will need to have its input loaded into memory, the main executor will often be blocked waiting for disk reads to finish. Some other tasks, such as ingest tasks, bring data to the system from external sources like the disk or network. This reduces the achievable processor utilization substantially. To overcome this, a secondary executor is needed to look ahead and load data while compute operations are running.

The I/O executor is a lightweight component that has its own ready queue for tasks explicitly marked as I/O tasks (like the ingest tasks). If the ready queue of the I/O executor is empty, the I/O executor can inspect the compute executor's ready queue and pre-load any missing inputs for future tasks. This requires careful management of the worker memory quota. One way to implement this is to have a fraction of the worker's memory used for pre-loading inputs.

3.4.4 Storage Levels

The worker supports three storage levels: permanent, temporary, and hidden. The DDO parts at every storage level comprise a logical set of DDO parts. For simplicity, they are treated as independent sets of DDO parts, although the actual implementation does not necessarily need to make that distinction.

The permanent DDO set contains permanently persisted DDO parts. These DDO parts are reconstructed every time the worker is started. The DDO parts in this set make up the data stored in the cluster warehouse. This is the source data that is prepared to be input to any analytical job. Adding new DDO parts with this storage level unblocks any waiting tasks.

The temporary DDO set is used for intermediate results. Temporary DDO parts are intended to be kept in memory during their entire lifetime. However, certain DDO parts will need to be unloaded when the worker reaches its memory quota limits and attempts to free up some memory. Temporary DDO parts that are no longer needed are dropped by the worker when it attempts to free memory. Adding new DDO parts with this storage level unblocks any waiting tasks.

The hidden DDO set is used to store partially computed DDO parts. These DDO parts are not visible to tasks and they are usually promoted to the temporary or permanent storage levels at some point during their lifetime. For instance, *breakable* tasks will register their partially computed DDO parts in the hidden set so that they are not discoverable by other tasks until they are ready. Adding new DDO parts with this storage level does not unblock any waiting tasks.

3.5 Worker Organization

3.5.1 Worker Containers

A *worker container* is a daemon process containing one or more workers. Each node has exactly one worker container. Worker containers typically house one worker. However, the design allows for multiple workers per

worker container for cases where the system is deployed on high-end server clusters having tens or hundreds of CPU threads per node. Individual tasks may be unable to utilize huge thread pools efficiently, and in this case it makes more sense to have more than one worker per node under a single worker container. This grouping of workers under a single process can also be used to optimize data shuffle between the workers of the same group. In this case, socket use can be eliminated and the DDO parts can be directly shared in RAM.

3.5.2 The Universe

The *universe* is a singleton object in every worker container. Its main role is to maintain necessary data structures that describe the global system state. Global system state is required for local decision making, physical planning, load balancing, and fault tolerance, as shall be described in the following sections. The universe also maintains a set of all active workers that are assumed to be available and can accept tasks.

3.5.3 Virtual and Physical Workers

The universe provides a logical view of the entire system as an infinitely long chain of virtual workers. At the task level, the system appears to have enough workers to schedule tasks concurrently to operate on all DDO parts of any given DDO. This chain of workers is mapped on top of a fixed-size ring of physical workers. This was largely inspired by Al-Kiswany *et al.*'s work on a storage system called NICE [2] and it largely resembles actor naming. Moreover, the mapping from physical to virtual workers can be implemented either in software or in hardware by using software-defined networking as proposed in NICE. However, for the remainder of this section, it is assumed that the mapping is implemented in software.

Each worker is assigned a permanent physical identifier during deployment which determines its position in the physical ring. Tasks can interact with other workers via their virtual identifier only. The mapping from the worker's virtual identifier to the physical identifier is managed solely by the universe. Replication and load balancing are managed by manipulating the mapping from virtual to physical identifiers. Consider a function m mapping from virtual identifiers to physical identifiers, such that

$$id_p = m(id, part\_id, r) = h_2(h_1(id, part\_id) + r) \tag{3.1}$$

$$h_2(x) = x \bmod |workers| \tag{3.2}$$

Given the DDO identifier id, the part index $part\_id$, and the replica index r, the function m could be applied to determine the physical worker identifier id_p at which the specified DDO part resides. The first replica of a DDO part p is at node $m(id, p, 0)$, the second replica is at $m(id, p, 1)$, and so on. Every DDO implementation in the system has an implementation of the hash function h_1. The function should ensure that DDOs are distributed over the worker ring starting at an arbitrary position for every unique DDO

identifier. It also controls the block placement for the DDO parts. If the function returns the same value for different DDO parts, the storage location is guaranteed to be the same physical worker. The function h_2 maps the infinitely long virtual worker chain on top of the fixed size physical worker ring.

These part replicas need not be identical in structure and may be different representations of the same block of data, each optimized for certain compute operations. In fact, the DDO controller implements a function that is able to compute a replica r_2 given any other replica r_1. Nodes in the lineage DAG may indicate that they prefer some replica r for optimization purposes. However, if the requested replica is temporarily unavailable for any reason, another suitable replica will be chosen automatically. The tasks interact with virtual workers through the universe, completely isolated from node failures and the actual cluster size.

3.6 Events

In master-oriented designs, like Hadoop MapReduce, executors send heartbeats to the master at fixed time intervals. The master looks for missed heartbeats and after a certain timeout, the master will assume the node to be dead. Events like finishing a task or having a failed task are reported to the master, piggybacked on top of the heartbeat messages [19]. In the architecture proposed here, instead of a central master node, there is the universe instance in each worker container listening to events broadcasted from all other workers. The universe, in its normal operation, discards all events. However, any system component can subscribe to certain events (such as specific types of events for some job x) and have their listener function called every time a new event is received.

Inspired by ideas from information theory, the periodic heartbeat messages were removed in this architecture. There is very little benefit in reporting that a likely event has occurred, such as a running worker that is (unsurprisingly) still running as expected. It is more informative to report unlikely events and important job milestones. Event messages are broadcasted only to notify the interested listeners of important milestones and potentially unexpected events. The universe itself is subscribed to a few special types of events related to cluster state. Such events include worker start/stop, DDO part add/drop, and lost-worker events. It should be noted that heartbeat messages are used in special cases only. This is described in more detail in Section 3.9.2.

3.7 DDO Controller

Hive allows the user to run queries on partitioned and/or bucketed data objects which are organized in a hierarchical file and directory structure [34]. One problem with this approach is that often the file system will not be optimized for file and directory discovery. To combat this problem, Apache Hive uses a central database to store the system catalog and metadata. This catalog is called a *metastore*. Without the metastore, the query planning phase can be extremely slow for huge tables consisting of thousands of directories and tens of

thousands of files. Another drawback to this approach is that the hierarchical namespace lacks the expressive capability to capture all relations between data objects. Furthermore, storage constraints such as object co-location, heterogenous distribution, and replication are either very difficult or impossible to indicate to the storage layer.

A per-workload implementation of the namespace can address these problems. Each workload implementation also implements a namespace for its objects as part of the DDO controller implementation. The DDO controller module provides three main functionalities:

1. Create a logical namespace for data objects,
2. Control block placement, and
3. Implement block replication

The universes on all nodes collaborate to store shared and private pieces of the namespace. The exact details of the namespace design are left to the implementer of the DDO controller. However, there are a few design guidelines that a DDO controller should follow:

- The DDO controller can map any arbitrary data object name to a DDO identifier. The name is simply a string and the semantics of the object name matter only to the implementers of the DDO. This functionality is used by the client to generate a query DAG that references certain objects.
- The DDO controller keeps metadata which contains summary information such as the total number of parts, total size, etc. for any given DDO. This information is often required for query planning and optimization purposes.
- The full namespace metadata must be replicated on all universes.
- Every worker stores a non-overlapping subset of the global set mapping from DDO identifier to DDO part numbers; i.e. every worker can report all locally available parts for any DDO, given its identifier. The subset of DDO parts reported by a particular worker for any given DDO identifier is referred to as the 'active set' for that worker. The size of the active set in any worker can be changed according to its local compute and storage resources.
- The DDO controller must implement the hash function h_1 described in Section 3.5.3.
- The DDO controller must implement a replica generating function that can produce any desired replica given any other replica.

Using a structure designed specifically for the workload needs allows the system to be aware of the high-level meaning of the stored logical entities. In the implementation presented in Chapter 4, these entities are divided into schemas, tables, and columns. Having this kind of specialized structure allows for more

complex goals, such as storing performance-optimized replicas; i.e. replicas optimized to do certain compute operations.

Another benefit of designing a namespace specifically to suit the needs of the data object is the ability to co-locate or fully replicate (broadcast to all workers) data objects. The implementation of the hash function h_1 in Equation 3.1 is left to the DDO controller implementer. The DDO controller can group together DDOs and have h_1 return the hash of the group rather than the DDO. For example, in the implementation presented in Chapter 4, DDOs are used to hold the data of individual columns, so, naturally one would want to co-locate neighbouring parts of all columns of the same table for obvious reasons.

Furthermore, tables that are frequently joined together could be distributed based on content and have DDO parts co-located with their frequently-joined counterparts. This significantly speeds up join operations by reducing the amount of data that needs to be shuffled and fetched from remote workers. In other systems such as HDFS, this kind of control over block placement is impossible to realize. Even if control over the storage layer is given and blocks are organized in an optimal way according to the workload, the compute engine will not know about this and it will still attempt a shuffle operation to inspect all records. No actual data will be sent over the network but millions or even billions of records will need to be scanned.

3.8 Physical Job Planning

Query planning or job planning takes place at both the client and at the worker nodes. The client is responsible for the majority of query planning work (called *logical planning*), including specifying a DDO implementation type, and creating and optimizing a query plan. However, the client does not generate the individual tasks that workers will execute. Instead, each worker node has a physical planner for every workload type which can generate local (and possibly remote) tasks.

The client presents a user interface that allows the user to specify certain jobs using high level commands and/or query language. In the case of an analytical query job, this will result in the client contacting one of the workers to obtain a copy of the namespace (from the corresponding DDO controller implementation), followed by constructing a query plan to compute the required result. The namespace is retrieved to help in converting user-friendly DDO names to identifiers. The namespace may also contain information about which performance-optimized replicas are available, and the sizes of all DDO parts in the system. Rule-base and cost-based optimizations can then be applied to the query plan. Once the query plan is ready, a job message is sent to any random cluster worker. The random worker choice ensures balanced loads in a multi-tenant environment.

The worker listener will then receive the job message and immediately take three actions: (1) assign a job identifier to the job message, (2) subscribe to all events relating to that job and forwarding these events to the client, and (3) broadcast the job message to all workers. Each worker will examine the DDO implementation type on the job message and invoke the corresponding physical planner to generate tasks for the job.

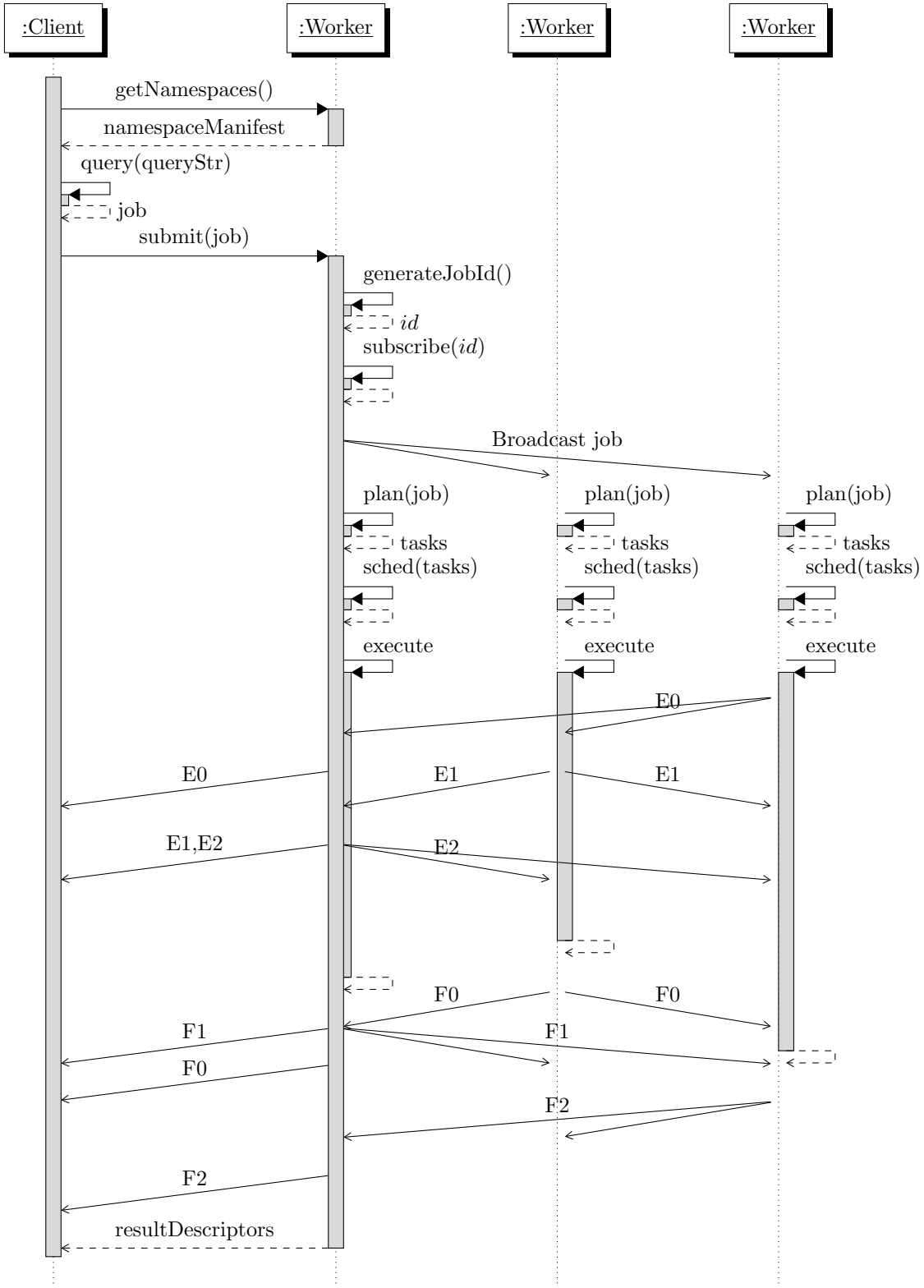


Figure 3.3: Physical query planning and execution. Self calls usually represent calls to subcomponents residing in the same node, but were omitted for simplicity. Broadcasted events are represented by 'E' and 'F' messages. 'E' messages represent regular events while 'F' messages represent finish events for individual workers. Notice how forwarding delays can result in event reordering and/or grouping.

The physical planner will usually consult the worker and retrieve the active set of DDO parts for all DDO identifiers present in the DAG. The physical planner will then traverse the DAG and produce tasks that invoke compute primitives on the DDOs. The actual algorithm used to traverse the DAG and generate tasks is left to the implementer of the physical planner module. This entire process is illustrated in Figure 3.3. Generating tasks/code for locally available data blocks (worker's active set) as a response to incoming job messages can be considered a special case of the more general function passing model proposed by Miller *et al.* [27].

An optional component of the system is a DAG analyzer. A DAG analyzer will examine the job DAG and the task list to determine the optimal order of execution of tasks. The definition of "optimal order" in this case depends on the workload itself to some degree, but generally the system avoids overwhelming the caches (processor, system, and OS buffers), causing otherwise unnecessary evictions and thrashing.

3.9 Fault Tolerance

Like the Hadoop distributed file system and many other distributed systems, fault tolerance is a key concern. The system described in this thesis serves as both a storage system and an execution engine, hence, fault tolerance measures should address those two areas. For storage, the DDO parts are replicated on different nodes. For execution, lineage-based recomputation is used to retrieve any lost DDO parts. The location of replicated DDO parts dictate which nodes will be able to spawn backup tasks to recompute any lost intermediary DDO parts.

3.9.1 Replication

The system uses two strategies for data replication. First, there are the performance-optimized replicas. The only constraint that these replicated DDOs should obey is that any given DDO part must have the same record data as all other replicas of the same DDO part in order to fulfill reliability goals. The data representation (encoding, record order, indexing, etc.) can be entirely different to optimize for certain compute operations. This constraint ensures that a task requiring a given performance-optimized DDO part can be given any other replica and have it produce the same result. In case of a node failure, the system will automatically resort to the DDO controller to identify a suitable replacement DDO.

The second data replication strategy is much more traditional. It simply places multiple copies of DDO parts in the down-chain workers (workers having higher virtual identifiers), as explained in Section 3.5.3. The objects are copied a number of times to satisfy the requirement that each DDO must be replicated a configurable number of times. This is referred to as the replication factor and denoted by r. These two strategies are implemented by the DDO controller in its replicas generating function. The replica generating function could either return the same exact DDO part or compute a different one.

3.9.2 Failure Hiding

Workers expect cooperation from other workers and look for certain relevant events during execution. Let us assume that a running job has a *shuffle* operation in its lineage DAG, followed by a *merge* operation. The *shuffle* and *merge* operations in this example are similar to their counterparts in a MapReduce system. The task dispatched for the shuffle operation takes some local DDO part and splits it into shuffle blocks to be sent to other workers. The merge task remains blocked until its inputs are ready. At some point in time, after all shuffle blocks from other workers arrive, it is expected that the task will be unblocked and be put on the ready queue. In this case, the worker subscribes to events relevant to its own blocked tasks only. If the task has been blocked for a prolonged period of time due to a missing shuffle block, the worker will ask the universe to ping the lagging worker (or workers). The universe will then send a message to the worker, asking it to send a heartbeat to confirm that it is alive. If the heartbeat does not arrive within a time window, the worker is marked as dead. Whenever a universe instance detects a dead worker, it broadcasts a *lost worker* event. All universes will adjust their worker mapping function to redirect tasks to the worker serving the next suitable replica. The worker hosting the replica will be notified of this (by its accompanying universe) and it will re-examine the job lineage DAG to schedule additional tasks to produce the missing shuffle blocks.

As discussed earlier, a DDO part is replicated in the down-chain workers a certain number of times according to the replication factor r. When a node is lost, the next replica in the down-chain order will need to be activated (i.e. temporarily added to the active set of DDO parts). According to Equation 3.1, h_1 ensures that the DDO parts are stored starting from an arbitrary position. Assuming there's a considerably large number of DDOs, one can expect that a single lost worker will place an equal load on all remaining workers.

After receiving a lost worker event, the Universe on each worker container will instruct its workers to take two actions: (1) identify the DDO parts that are locally stored and need to be added to the active set for future jobs, and (2) examine the DAG of all incomplete jobs and immediately schedule tasks that take as an input one or more DDO parts which were added to the active set. The same steps can be applied to the second replica, third replica, and so on. In fact, the workers will add a DDO part to the active set only if they perceive themselves as the next worker in the chain for a given DDO part.

3.10 Data Ingest

The system detailed here is intended to serve as a scalable, fault-tolerant, platform for analytics and long-term storage of huge volumes of data. The nature and type of data objects can vary as described in Section 3.2. Data ingest is performed in bulk and the loaded data is expected to remain unchanged for long periods of time. This limitation is appropriate for use cases concerned with analytics and long-term storage.

3.10.1 Data Ingest Jobs

Loading data into the system is done using a special type of job called an ingest job. An ingest job distributively fetches data from any arbitrary source. In fact, the only requirement a data source must have is supporting concurrent access, because all workers will run ingest tasks concurrently; even this requirement could be relaxed by implementing a special type of single-node job. Unlike the regular job message which would usually contain the lineage DAG of the required result, an ingest job message will contain information specific to the data source and its connector. The tasks produced for this job are often characterized by having an empty input set and a non-empty output set. These tasks could be followed by some other tasks that take the raw ingested data and apply a few transformations, possibly producing performance-optimized replicas. For example, in the implementation presented in Chapter 4, DDO parts could have local and global indexes that help speed up certain compute operations.

During ingest, the data is loaded into the system without its accompanying namespace information. This makes the data objects undiscoverable by system users. Once the ingest job completes, the namespace information is atomically updated on all nodes. This will allow users to discover and request transformations on the data.

To allow correct system operation, the data objects are required to be immutable after ingest. This constraint ensures that any transformation on the data is deterministic once the job is submitted to the cluster, regardless of the point in time the individual tasks are executed. Specifically, result determinism can be stated as "the ability to infer certain characteristics about the data by examining the namespace metadata at the time of job submission". For instance, if the namespace metadata reports that there are x number of records at the time a job is submitted, then a *count* operation is expected to produce the same result. The immutability constraint also allows tasks from multiple concurrent jobs to be interleaved and produce correct results. Unlike a data warehouse, this system does not allow fine-grained transactions on data objects. However, bulk updates could be applied via a *locking update delta*.

3.10.2 Update Deltas

An update delta allows data records to be added, changed, and/or deleted entirely. This requires a complete halt of all access to the data object. Acquiring the lock can be done by delaying submissions of jobs that operate on the data objects to be updated and waiting for all unfinished jobs to complete. Once the lock is acquired, a special type of job will be executed to update the necessary DDO parts. After that, the namespace metadata is updated and the lock is released, allowing the waiting jobs to be submitted.

These limitations are usually acceptable in a purely analytical use-case. It should be noted that DDO parts cannot be updated independently of each other. The entire DDO needs to be locked during the bulk update, even if the update will only affect a single DDO part. If access is permitted to all other parts, this would violate the result determinism constraint described earlier.

3.11 Cluster Clients

The cluster as a whole can be considered as an analytics and warehousing server. As explained in Section 3.8, clients can connect to any node, request namespaces, submit jobs, observe their job's progress, and collect the result DDOs. In this model, the cluster only responds to a handful of message types, most of which carry either DDO parts or lineage DAGs, which comprise most of the cluster's network-level API. Cluster clients could be implemented in three different forms:

- **A plain client** offering a user interface for a user to input queries using some query language,
- **A proxy server** implementing a standard API such as JDBC (Java database connectivity)/ODBC (open database connectivity) connector APIs, or
- **A connector library package** providing a programmatic high-level API that can be integrated into other software utilizing the analytics capability of the system.

3.11.1 Plain Clients

A plain client will need to implement query parsing and optimization of some human-readable query language. Usually, the client will be able to interact with a single type of DDOs in the system. This limitation depends on the query language itself and its expressiveness. It is difficult to imagine a query language supporting analytics on more than one type of data objects, however, it is not impossible.

Consider the case of a DML (data manipulation language) query which requests some transformation on one or more logical entities. The query string is first parsed, and the logical entities are identified using the namespace(s) obtained from the cluster. Once the query is verified and all logical entities are identified, the query planner attempts to traverse the parse tree and generate an initial query plan. The initial query plan can be in the form of a DAG, but not necessarily a lineage DAG that is compatible with the cluster. This graph can then be traversed to apply rule-based and cost-based optimizations. The namespace metadata can be used for cost-based optimizations. Finally, the DAG can be converted to a cluster compatible lineage DAG carrying known DDO identifiers and transformations. The scenario described here is the equivalent of the *query(queryStr)* step in Figure 3.3.

3.11.2 Proxy Servers

A proxy server does not need to provide a user interface but may need to implement parsing of some query language, depending on the API design. Some APIs allow query strings to be passed to the server. A proxy server will also need to implement some socket-based API, such as the JDBC/ODBC APIs, and listen for incoming connections. Query planning and optimization is done in the same manner as in the case of the plain client. Proxy servers can be useful in adapting existing software to interface with the cluster.

3.11.3 Connector Libraries

Unlike plain clients and proxy servers, which may need to implement parsing of some query language, a connector library package will implement a high-level programmatic API. The API would offer a DSL (domain-specific language) to manipulate the data. An example of this is the popular Pandas DataFrame[1] abstraction. The library can be implemented for any language that offers socket programming facilities.

The API would offer dummy primitives (on dummy data objects) which can be mapped to available DDOs and cluster-compatible lineage DAG nodes. The primitives would use deferred evaluation, manipulating some DAG, until a *forcing* primitive is invoked. A forcing primitive will result in the query lineage DAG to be optimized and sent over the wire. Query DAG optimization can be carried out in the same manner as in the case of the plain client.

3.12 Summary

In this chapter, the design of a modular distributed system for structured big data analytics was presented. The modular nature of the architecture allows workloads to specialize areas of the system which are left to workload implementers. Many elements of the design are largely inspired by popular systems including MapReduce and Apache Spark. The DDO interface exposes compute, memory, and storage mechanisms which are used by the execution and storage engine. This coupling of execution mechanisms gives the DDO implementer the ability to implement hidden internal optimizations which are mostly concerned with minimizing the volume of data transferred between different levels of the memory hierarchy and between workers. In addition to the DDO interface, a per-DDO controller module provides a logical view of data objects and controls their distribution across the cluster.

A per-DDO implementation physical planner complements the DDO and uses its compute primitives to create tasks for incoming jobs. The execution engine schedules the tasks produced by the query planner and prepares the DDOs accordingly. The DDO exposes memory control primitives such as *load()*, and *pin()*. The system can use these primitive to pre-load, cache, and evict data objects as needed. The system also uses a lineage DAG analyzer to optimize the order of tasks and minimize thrashing.

The execution engine has two executors: a compute executor and an I/O executor. This allows tasks marked as I/O tasks to be run asynchronously. When idle, the I/O executor will look ahead and pre-load the inputs of queued compute tasks, if needed. The two executor architecture establishes an asynchronous data pipeline that reduces scheduling delays.

The system is viewed as an infinitely long chain of virtual workers than can be scaled as needed. This view of the system allows failures and replication to be hidden easily. The *universe* object is responsible for

[1] Pandas DataFrames. `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`. Accessed April 2, 2021.

(1) maintaining the mapping from virtual to physical workers, (2) synchronizing global system state, and (3) connecting all the available physical workers together.

Minimizing data access requests, co-locating data objects, and maintaining performance-optimized replicas, as well as the overall modular design are the main differences between the design presented here and current state-of-the-art systems. These features improve the system efficiency and allow it to handle large volumes of data gracefully. For analytical tasks, this can improve query performance substantially as shall be demonstrated in Chapter 5.

4 Implementation

This chapter outlines important implementation details as well as present a DDO implementation called the *Relation* DDO. Section 4.1 presents an overview of the system implementation and discusses the rationale behind important implementation decisions. Section 4.2 discusses the implementation of DDO libraries. Section 4.3 presents the Relation DDO. The novel data distribution and query execution techniques are discussed in Section 4.4. An example query plan is shown in Section 4.6. Finally, Section 4.7 summarizes the chapter.

4.1 Overview of System Implementation

Implementing a new system starting from a clean slate makes it possible to test ideas that would normally be very difficult to integrate into existing designs. One of the main reasons behind this decision is that this design dictates strong coupling between the storage and compute layers. None of the existing big-data systems will allow introducing this concept easily, or at least without refactoring too much of their functionality. More time will be spent on understanding and refactoring than implementing new functionality. For this reason, implementing a new system was the clear choice. However, building a new system opens a lot of questions about many previous design choices. Although more time consuming, taking a first principles approach gives the opportunity to re-evaluate design choices common in big data systems and reason about their benefit to solving the problem at hand.

Apache Hadoop and Apache Spark both run on the JVM; with the former implemented in Java, and the latter in Scala. Since these platforms were meant to be general execute engines, code portability is a key concern. Users need to be able to write code defining one or more analytical tasks and/or custom data objects. The code will then need to be serialized to the workers for execution. The nature of the analytical task and the data object can change on a per-job basis. However, in a warehousing system where the analytical task and data object do not change often, the portability requirements can be relaxed.

Although C++ is a notoriously verbose language –and in many cases difficult to debug– the main reason behind the language choice is multi-core performance. Recent developments in the STL libraries and the introduction of parallel algorithms in C++17 makes the language a clear choice for leveraging strong multi-core performance. With increasing core per chip densities, workers having tens or hundreds of cores are becoming more and more common. Drocco *et al.* explored the use of C++ for parallel distributed algorithms and showed that current C++ parallel algorithms can use special iterator implementations that access

distributed containers and continue to show strong scalability [17]. OpenMP[1] also provides an easy-to-use interface for parallelizing existing C/C++ code.

The core part of the system is implemented using C++ in about 26 thousand lines of code. The system can accommodate heterogenous DDOs and run multiple concurrent jobs. As a proof-of-concept, an example DDO called the *Relation* was implemented to benchmark, study, and improve the design of the system. The system is intended to support a wide variety of analytical workloads on big data and is yet to be tested on other workloads.

The work done in this thesis was focused on developing the core execution and storage engine as well as one DDO implementation called the *Relation* DDO. For the Relation DDO, the intent is to provide a range of primitives that can support the functionality of most SQL statements. The implemented cluster client lacks a proper SQL interface or higher-level APIs. Currently, a few query lineage DAGs are hardcoded into the client for benchmarking purposes. Future work could focus on supporting client-side query languages as well as developing other DDOs for different types of workloads.

4.2 Implementing a DDO Library

DDOs and other accompanying modules are pluggable system components by design. In this implementation of the system, the modules are bundled into shared objects to be dynamically linked to the worker container process. DDO implementation libraries are linked during worker container startup just before the workers are started; after that the worker container process is ready and has all the implementations it needs to deal with the DDO types stored in the system. A DDO library should include implementations for the following:

- An implementation of the *DDO* interface,
- Extensions of the *LineageNode* class,
- An implementation of the *Job* and *Physical Planner* interfaces for physical query planning,
- An implementation of the *DDO Controller* interface, and
- One or more implementations of the *Loader* interface for ingesting data.

First, an extension of the DDO class needs to implemented. This includes implementing pure virtual functions as well as overriding some of the already existing functions. The DDO implementation is the developer's way to introduce compute primitives relevant to the analytical task they would like to perform. It also allows the developer to specify how the object will be serialized over network and written to secondary storage. An accompanying metadata file is provided to *load()/unload()* functions to help in object reconstruction.

Each of the compute primitives introduced in the new DDO will likely produce one or more new DDOs with changed data and lineage. Therefore, multiple extensions of the LineageNode class (not shown) must

[1]OpenMP. `https://www.openmp.org/`. Accessed January 29, 2021.

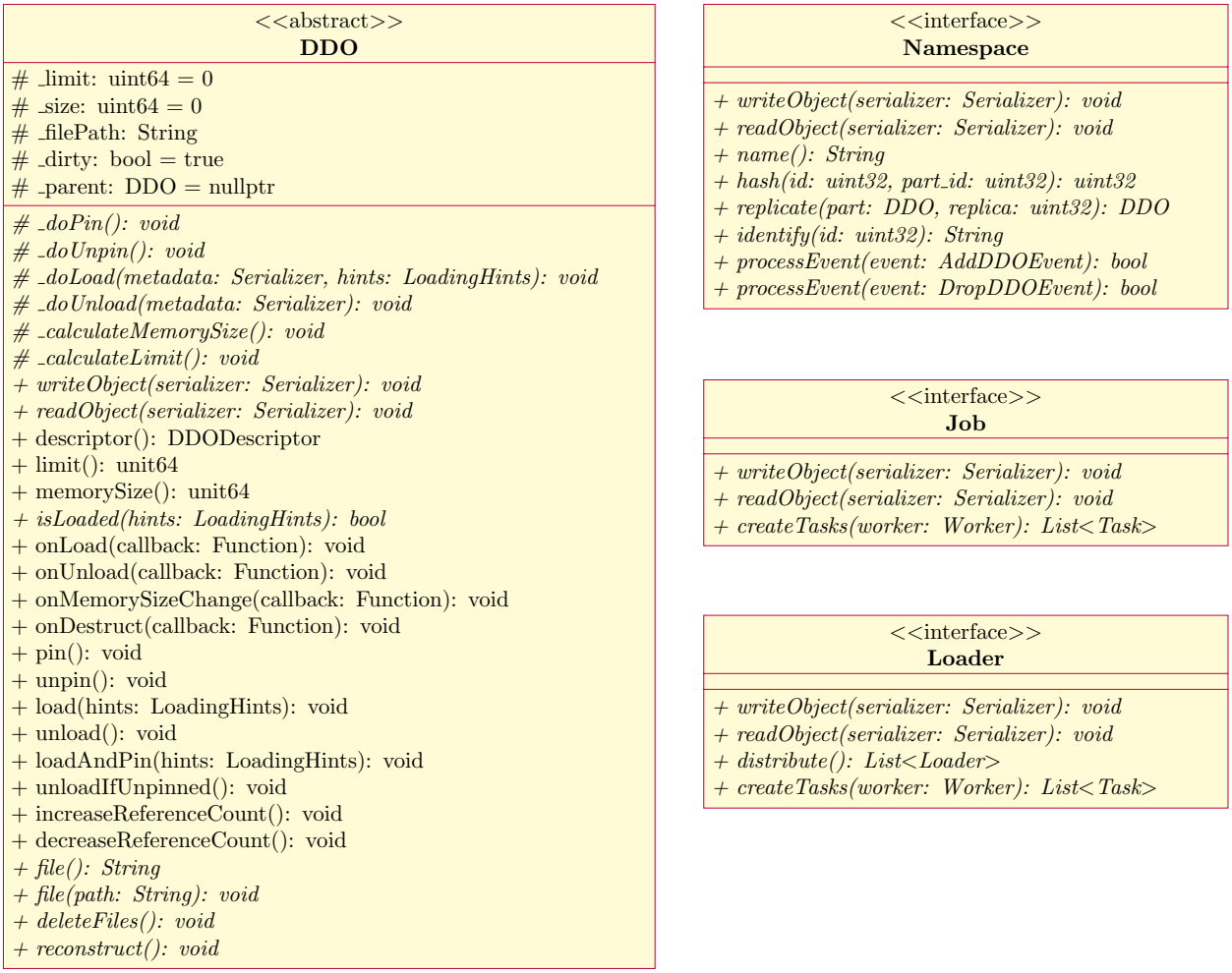


Figure 4.1: Class diagram of the main interfaces in a DDO library library. Virtual methods are in italics.

also be implemented. Each LineageNode contains information such as the corresponding compute primitive's arguments. LineageNode implementations are used to construct the lineage DAG of the DDO.

The newly introduced DDO also defines a way to serialize query jobs and data ingest jobs. For query jobs, the *Job* object usually includes a lineage DAG and provides a *createTasks()* function (which invokes the Physical Planner) to traverse the DAG and generate tasks for the job. Data ingest jobs, on the other hand, highly depend on the data source. Usually, *Loader* objects contain information such as the paths of the files to be ingested as well as an implementation of the *createTasks()* function. A typical implementation of the *createTasks()* function will (1) contact the data source to perform initial object discovery, (2) generate ingest and replication tasks, and (3) update the corresponding namespace. Figure 4.1 shows a simplified class diagram of the interfaces to be implemented in a DDO library.

4.3 The Relation DDO

The *Relation* DDO was designed to represent one or more columns in a table. The data object allows access to its constituent columns as well as the ability to add/drop columns in O(1) time. The object is also designed to be immutable; i.e. its data does not change after its construction and any transformation will produce a new copy. Depending on the transformation invoked, the copy may be either a deep copy or a shallow copy. Shallow copies allow simple lightweight transformations to be chained efficiently without consuming too much memory.

4.3.1 Compute Primitives

The data object supports basic primitives that allow most SQL queries to be executed. These primitives fall under two main categories: transformations and mutations. Transformations are methods that produce a new object with changed data. Mutations are methods that violate the immutability and alter the current object's data or the way it is viewed. Table 4.1 summarizes the most notable methods in the data object's interface. Mutations are uncommon and are considered to be an anti-pattern in this design. However, there are only two mutations and they are intended to simplify otherwise costly operations. These two mutations, *select()* and *mergeWith()*, are used carefully in very specific situations.

Mutations

The method *select()* is used to select a specific column in the relation temporarily for a subsequent single-column transformation. For instance, a call to *select(0)* followed by a call to *min()* will produce a new relation having a single column that is the minimum of the first column in the relation. This is to avoid creating a shallow copy of that particular column for the sole purpose of a single transformation. Shallow copies may seem like they are uncostly, which is true for the most part. However, registering too many data objects with the worker can lead to longer delays during *memory free cycles*, which evict or drop data objects.

The only other mutation, *mergeWith()*, is used to merge incoming shuffle blocks. In this design, the incoming shuffle blocks are merged as soon as they arrive. This approach is somewhat similar to the incremental reduction technique proposed by Elteir *et al.* [18]. While shuffle blocks are being merged, the incomplete result is hidden from other tasks (using the *hidden* storage level) until all the required blocks are merged. The process can be thought of as a prolonged object construction, since it only takes place at the beginning of the object's lifetime before any other methods are be invoked.

Transformations

Transformations, on the other hand, are the most frequently used methods. Typically, these methods have a 1-to-1 correspondence with nodes in the lineage DAG. The transformations presented in Table 4.1 comprise the set of primitives to support some of the most important SQL data processing features.

Table 4.1: Compute primitives of the Relation DDO

Transformations	Mutations
zip(list<DDO>)	select(uint32)
drop(list<uint32>)	mergeWith(DDO)
alias(uint32)	
filter(Expression)	
shuffle(Expression, Pivot)	
hashShuffle()	
join(DDO, Expression, JoinType)	
distinct()	
groupBy(list<uint32>)	
extractGroups()	
orderBy(list<uint32>)	
map(Function)	
aggregate(Function)	
count()	
sum()	
avg()	
min()	
max()	
sample()	

Two of the most notable and unique transformations in this design are the *zip()* and *drop()* transformations. These transformations allow columns to be added and removed from the relation in constant time. Having this kind of flexibility makes it possible to create query plans that bring needed columns later in the query plan, and drop columns that are no longer needed. This can substantially reduce the size of the data that needs to be loaded as well as reduce the size of intermediary objects.

Consider a query that filters a particular table by column a and then aggregates column b. The relation can start with column a, apply the filter, then zip column b for aggregation. The filtered-out records will not appear in column b. The first column in any relation is said to "lead" the relation. This means that the first column will control the records that can be seen in the relation. If column b was not previously loaded to memory, the relation may opt to load only the relevant regions of column b using the *LoadingHints* optional parameter discussed in Section 3.2.1. This largely depends on whether column a has an index that can be directly applied to b. In this case, a small number of contiguous blocks can be loaded from b, instead of loading all of b. This particular optimization is one major source of speed-up in this DDO design and is possible due to the tightly coupled compute and storage primitives of the DDO interface. Several performance-optimized

replicas are made to ensure that the majority of the expected queries will make use of optimizations such as this one.

4.3.2 The Schema: Relational DDO Controller and Namespace

The data object namespace is intended to provide a natural way to discover and access data objects. The namespace is responsible for providing the user a logical view of the data objects in the system. In this DDO implementation, the namespace is divided into schemas, tables, and columns. While the Relation DDO can hold more than one column at a time, the persisted data objects are the individual columns themselves. Each column has a system-wide unique identifier. These columns can have performance-optimized replicas or regular replicas.

The DDO controller can enforce co-location on particular sets of columns. These sets usually contain columns from the same table that appear frequently together in the expected queries. The co-location constraints can also be extended to include frequently joined tables to their corresponding DDO parts co-located and thus avoid shuffling large volumes of data over the network to compute a join.

The user-level application (through some client) can discover schema and table objects in the cluster warehouse by requesting a namespace manifest. The table objects contain references to their constituent column DDOs as well as their performance-optimized replicas. Not all columns of the base table need to be present in a performance-optimized replica. The user can request creation of optimized replicas on subsets of the full set of child columns. This is indicated in the *table ingest* job message. Depending on the query, the client may wish to construct a query DAG using different preferred replicas for different transformations. In the event that any DDO part of an indicated replica is unavailable, the system will use the DDO controller to look up a suitable replacement DDO part. This process is described in detail in Section 3.9.

4.4 Performance Improvements

As mentioned earlier, the system design allows for a number of replicas to be registered with the DDO controller. These replicas may be exact copies of the data or copies optimized for certain operations. Moreover, content-aware record placement and DDO part placement are used to reduce the amount of data shuffles. This section summarizes the novel techniques used by the Relation DDO to improve query execution.

4.4.1 Primary Key Index

A *primary key index* imposes the restriction that each DDO part will have records within a predefined range of values for the primary key of a given table. During data ingest the records are shuffled by the primary key of the table and an index is created to indicate the range of values for each DDO part. Only one primary key index may be chosen for a given table.

A primary key index can be useful in join transformations. The foreign table is shuffled by the foreign key using the same primary key index, thus guaranteeing that ranges of matching records end up at the same worker. This avoids having to shuffle both tables to perform a join, as the table having the primary key index is already distributed based on primary key values. Content-aware data distribution offers a substantial improvement over a system that processes records without any knowledge of data distribution.

Furthermore, a table may be distributed by another table's primary key index. One or more attributes from the ingested table will be chosen as the *distribution key*. The distribution key attributes are used to shuffle the records using the primary key index from a previously-ingested table. A co-location constraint on DDO parts having ranges of matching records will also be enforced. This optimization is useful for frequently-joined tables. The distribution key is usually one or more attributes from the composite key of a *fact* table referencing some *dimension* table's primary key.

Fact tables are tables that contain measurements (such as sales entries) and typically will reference one or more *dimension* tables (such as items, customers, orders, etc.). This is typically found in *star* schemas, which are used in many applications. The downside to using a distribution key is that a *fact* table can be optimized for only one of its *dimension* tables. Joining with other tables will require performing a shuffle operation in run-time. However, careful selection of the distribution key will usually result in favorable results.

4.4.2 Densely-Indexed Replicas

Densely indexed replicas are used to speed up transformations such as *filter* and *shuffle*. The DDO part, which is used to represent a horizontal shard of a given table, is indexed by a certain column. That column has record pointers so that the data of other columns can be fetched. The index is dense, meaning that there is a pointer for every record. A sparse variant of this index is possible, but is not yet implemented.

Dense indices can be used to filter a DDO part in O(1) time. The filter result is not evaluated by scanning all records. Instead, an index lookup will identify the regions of data that include the result. Furthermore, the index can be used to load only those regions (using a *LoadingHints* object) and avoid costly I/O operations.

The same can be applied for the *shuffle* transformation. When records are shuffled, they are sent to specific locations based on their value. Using an index can speed things up. Instead of having to scan all records and place them into buckets before sending them, the index can be used to split the DDO part into contiguous regions that can be sent directly to their destinations without having to scan individual records.

4.4.3 Grouped Densely-Indexed Replicas

In addition to densely indexed replicas which have the records indexed by the value of a given column, there is a grouped variant of this index. The index itself is split into buckets, each of which represent a group. The grouping clause is one that is expected to occur frequently in the workload. Having the records pre-bucketed and indexed makes it easy to filter and aggregate the data.

4.4.4 Broadcasted Tables

Tables having all DDO parts replicated on all workers are called broadcasted tables. This optimization is useful in cases where relatively small tables are frequently joined with one or more tables in the schema. This alleviates the barrier synchronization required to broadcast and collect all parts of a DDO during query execution. Instead the actual data broadcast is done once during ingest and persisted permanently by the workers. This can only be done if the table size is sufficiently small to be cheaply replicated on all workers.

4.5 Data Types and Serialization

Since the data object is concerned with a handful of data types, the data object can have special optimizations for efficient storage and management of every type. Each of the columns in a Relation object is backed by an array object. There are different types of arrays and each of them is specialized for a given data type. Array objects are mainly concerned with memory management and serialization. Arrays are built specifically for the purposes of fast serialization and providing primitives that leverage multi-core capabilities.

An array object does not require any processing during serialization/deserialization, thus eliminating the computational bottleneck of serialization algorithms that usually hinder the performance of general-purpose distributed engines (e.g. Apache Spark). This can help leverage the power of very fast I/O in powerful server machines (e.g. 40-100 Gbit ethernet and SSD arrays). Array objects can be serialized to/from network and persistent storage alike.

Apache Spark and other big data frameworks have to resort to complicated serialization algorithms in order to be able to deal with custom data objects. This is mainly because these frameworks were designed to deal with unstructured data that is usually parsed and converted into user-defined objects. This is not the case in this system. This system deals with structured data and a handful of known data types. By limiting the data types to a set of predefined types, nearly all computational work for the purpose of serialization could be eliminated entirely. This is achieved by enforcing the requirement that each array object is a contiguous, self-describing, block of data. This block of data can be transferred over the network or stored on persistent storage as-is.

Consider the case of shuffling a relation DDO part. The relation and all of its constituent columns will be split into buckets. Each one of those buckets contains a collection of arrays; an array for each column in the relation. The original source arrays are processed using the parallel computation primitives provided by the array object. The arrays are scanned in parallel and records are inserted into their corresponding buckets. These buckets, which are collections of arrays, will be transmitted over the network without any further processing. Thus, the computational work is only focused on the actual scanning and bucketing of records. Of course, having the relation indexed by the shuffle attribute will eliminate even this step. Taking this a step further, relations indexed by the shuffle attribute will produce shallow copies of different contiguous parts of the relation. If the target of some of these shallow copies is a worker under the same worker container, the

```
-- TPC-H Shipping Priority Query (Q3)

SELECT
    l_orderkey,
    SUM(l_extendedprice*(1-l_discount)) AS revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < date '1995-03-15'
    AND l_shipdate > date '1995-03-15'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue desc,
    o_orderdate;
```

Figure 4.2: SQL listing of an example query.

object is shared through memory.

4.6 Logical Query Planning

As discussed in Section 3.11, the cluster clients are capable of producing query job descriptions and sending them to the cluster. During job execution, the client observes the job's progress via the forwarded universe events. The entire process is fully decentralized and does not require any intervention or a central coordinator. The only control available to the client is the ability to kill the job.

A job message will typically contain a query plan expressed as a lineage DAG of the desired result. The lineage DAG is later interpreted by the physical planner on every worker. After execution, the client is given a set of descriptors for the result DDO parts. The client can then collect the DDO parts from one or more workers and perform a final step to merge the (relatively small) DDO parts. This can be also done on the cluster if the client machine has limited resources. In cases where a high-speed connection to the cluster and reasonable resources at the client exist, it is better to perform the final step at the client since data needs to be fetched from the cluster in all cases. Consider the example in the Figure 4.2 and its corresponding lineage DAG in Figure 4.3.

The lineage DAG represents the final (optimized) query plan produced by the client which can be sent to one of the cluster nodes. This will in turn invoke the physical job planning procedure described in Section 3.8. For the proof-of-concept system, a simple client was implemented with hardcoded lineage DAGs for a number of benchmark queries in order to assess the system's performance. The lineage DAG shown in Figure

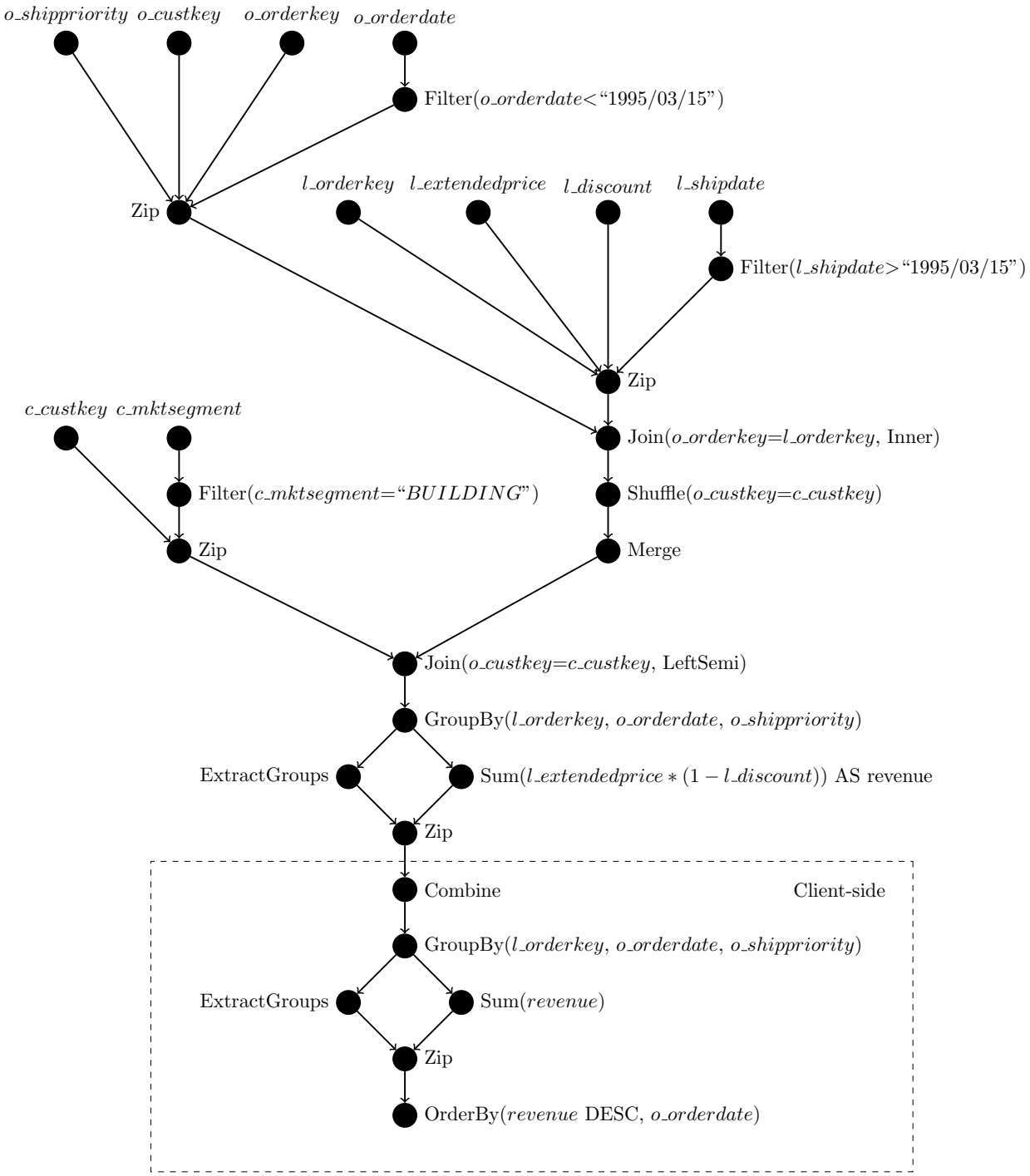


Figure 4.3: Example query plan as a lineage DAG. Note the first join between tables *lineitem* and *orders* is performed without needing to shuffle records due to co-location.

4.3 is equivalent to the listing in Figure 4.4.

Lines 1 through 17 obtain references to namespace objects representing tables in a namespace called "tpch". The table objects contain references to their child column DDOs. A table object will also contain references to its optimized replicas. By using one of the replicas in the lineage DAG, the query plan is implicitly specifying a preferred replica. In the case that the chosen replica is lost due to failure, the physical planner will produce replacement tasks and change the DDO references to one of the available replicas. The optimized replicas allow efficient data loading by performing an external search and retrieving slices of DDO parts that correspond to the filter predicates. Depending on the filter predicate, this can significantly reduce the I/O and compute time needed.

Another interesting optimization is DDO part co-location. Note how tables *lineitem* and *orders* are immediately joined in line 27 without having shuffle and merge nodes as in lines 39 and 40. This is because the records of both tables are shuffled during ingest by the *orderkey* attribute in addition to having a co-location constraint on their respective DDO parts. This constraint is enforced by the DDO controller implementation accompanying the Relation DDO to ensure that all matching records are persisted at the same workers. The same could be achieved with denormalization but with a greater storage footprint (and I/O time). Information about available replicas and co-location constraints are all accessible through the DDO controller object for query planning and optimization purposes as well as physical job planning purposes in the event of data loss. More query plan examples can be found in Appendix B.

4.7 Summary

This chapter presented an overview of the system implementation as well as the DDO implementation. The DDO and other modules are pluggable system components. Each DDO implementation library is concerned with a specific type of workload. The system is designed to store multiple heterogenous DDOs and allow multiple system tenants to run analytical tasks on shared data.

The Relation DDO was implemented to evaluate the system performance and help guide the design process. The implementation relies heavily on performance-optimized replicas and co-location to speed up certain compute operations in a given query plan. Data object co-location allows frequently joined tables to be joined immediately without the need to shuffle data records. The speed-up gained by employing these techniques largely depends on the nature of the executed queries as shall be demonstrated in Chapter 5.

```
auto lineitem = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("lineitem")
    .replicas()
    .indexedBy("l_shipdate");

auto orders = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("orders")
    .replicas()
    .indexedBy("o_orderdate");

auto customer = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("customer")
    .replicas()
    .indexedBy("c_mktsegment");

Lineage rel = {
    Lineage(orders["o_orderdate"])
    + Filter(c(orders["o_orderdate"]) < date("1995-03-15"))
    + Zip()({
        orders["o_shippriority"],
        orders["o_custkey"],
        orders["o_orderkey"]
    })
    + Join(
        c(orders["o_orderkey"]) == c(lineitem["l_orderkey"]),
        JoinType::Inner
    )(
        Lineage(lineitem["l_shipdate"])
        + Filter(c(lineitem["l_shipdate"]) > date("1995-03-15"))
        + Zip()({
            lineitem["l_orderkey"],
            lineitem["l_extendedprice"],
            lineitem["l_discount"]
        })
    )
    + Shuffle(c(orders["o_custkey"]) == c(customer["c_custkey"]))
    + Merge()
    + Join(
        c(orders["o_custkey"]) == c(customer["c_custkey"]),
        JoinType::LeftSemi
    )(
        Lineage(customer["c_mktsegment"])
        + Filter(c(customer["c_mktsegment"]) == l<String>("BUILDING"))
        + Zip()(
            customer["c_custkey"]
        )
    )
    + GroupBy({ "l_orderkey", "o_orderdate", "o_shippriority" })
};

Lineage queryPlan = {
    Zip()({
        rel + ExtractGroups(),
        rel + Aggregate("sum(l_extendedprice*(1-l_discount))")
    })
};
```

Figure 4.4: Listing of an example query plan in C++.

5 System Evaluation

The implementation presented in Chapter 4 serves as a proof-of-concept to show that this design can substantially speed up and improve the efficiency of analytical workloads on structured big data. Two industry-standard benchmarks will be used to evaluate the system's performance on relational query processing. This chapter presents the experimental results and an in-depth investigation of the system behaviour compared to the baselines.

Section 5.1 summarizes the experimental setup used for evaluation. Section 5.2 introduces the benchmarks used. A brief note on query optimization for the Relation DDO is found in Section 5.3. Section 5.4 introduces the baseline systems. Section 5.5 discusses the performance metrics used in this evaluation. A brief note of implementation correctness is presented in Section 5.6. The results and discussion are presented in Sections 5.7 and 5.8. Section 5.9 concludes the chapter.

5.1 Experimental Setup

The experiments were conducted on a cluster of ten commodity desktop computers. One machine is used as the master node and/or client. The remaining nine machines comprise the cluster workers and are configured as follows:

- CPU: Intel Core i7-2600 running at 3.40GHz,
- RAM: 16GB DDR3, 1333 MT/s,
- Storage: 8TB HDD, 14 ms average seek time, 140 MB/s average read speed,
- Network interface: 1Gbps ethernet,
- Operating system: Ubuntu 18.04.5 LTS, and
- Kernel: Linux 4.15.0-128-generic.

5.2 Benchmarks

5.2.1 TPC Benchmark H (TPC-H)

The TPC-H benchmark is a standard decision support benchmark [35]. The benchmark suite consists of a data generator and a set of 22 queries. This particular suite focuses on examining large volumes of data

and generating reports that answer important business questions. The benchmark is used to evaluate the system's compute performance, as most of the benchmark queries will force the system to aggregate large quantities of data.

The TPC-H data generator creates CSV (comma separated values) files containing the record data of 8 tables. The data generator is capable of sharding the dataset into any number of blocks and generating any specific block on demand. This allows the cluster nodes to synthesize the dataset in parallel. The CSV files are intended to be ingested into a warehousing system where the analytical work will take place.

5.2.2 TPC Benchmark DS (TPC-DS)

The TPC-DS benchmark [36], like the TPC-H, is also a decision support benchmark. However, this benchmark offers a more modern take on analytical decision support systems. It consists of 99 queries and a complex star schema. A star schema is a schema characterized by having a set of *fact* tables which reference a one or more (usually a handful or more) *dimension* tables. Dimension tables are usually small in size and can be cheaply replicated on all nodes using the fine-grained block placement mechanism. This makes most of the computations invoked by the queries embarrassingly parallel, a setting extremely suitable for distributed systems. For most queries, tiny amounts of data will need to be shuffled or broadcasted across the system. This benchmark is expected to highlight the benefits of fine-grained and workload-aware block placement.

It should be noted that development on the Relation DDO was halted as soon as full coverage of the TPC-H benchmark was reached. The TPC-DS benchmark also requires implementing numerous advanced SQL features which would require a substantial implementation effort and may not necessarily highlight any new performance characteristics. For these reasons, only a subset of the 99 TPC-DS queries was chosen to showcase the benefits of the block placement techniques.

5.3 Relation DDO Query Optimization

The Relation DDO can optimize the execution of expected workloads by organizing the data and using replicas to serve as optimized representations of the data for certain compute operations. The assumption is that there is a known workload that can be used to characterize the majority of expected query plans. This is used in important decisions such as choosing an appropriate *distribution key*, and creating one or more *densely-indexed replicas* as discussed in Section 4.4.

Applying this to the TPC-H and TPC-DS benchmarks was straight-forward in most cases. All tables have primary key indices based on the their primary key attributes. Some *fact* tables are organized using a *distribution key* that corresponds to the primary key of one of their most-frequently-joined *dimension* tables. Finally, *densely-indexed replicas* and *grouped densely-indexed replicas* are created based on frequent filter and grouping clauses in the benchmark queries. These optimizations cannot be applied to all queries as some may have less-frequent filter/grouping clauses that do not justify the creation of a dedicated replica, or require

a join between tables that are not co-located. In these cases, the query plans force the system to operate similar to the baselines. However, this was only found in a few queries and due to the overall efficiency of the data processing pipeline these queries may gain a moderate speed-up.

5.4 Baselines

5.4.1 SparkSQL

SparkSQL is Apache Spark's SQL interface implementation. It is widely used in industry and is mainly intended to be used in the early stages of larger processing pipelines [4]. It can be used to augment the data and/or perform some initial aggregation. Nearly 35% of organizations use Apache Spark in machine learning pipelines [5], and those pipelines are very likely to have SQL in them. However, due to its performance and open-source nature, SparkSQL has become one of the most commonly deployed distributed SQL analytics solutions and it is sometimes used in SQL-only workloads.

The use of *whole-stage code generation* in SparkSQL largely improves its performance.[1] Using code generation, SparkSQL is able to pipeline entire compute stages (successive tasks having narrow dependencies that do not need shuffles or external inputs). These compute stages are converted into single tasks running dynamically generated code. For example, three successive tasks performing filter, grouping, and aggregation can be all converted into a single loop having three steps: (1) an if-statement for the filter, (2) computing the hash value of the grouping attributes, and (3) performing a hash table lookup to update the aggregate value directly. In this case, there is no need to keep actual buckets of grouped records. Only aggregated values are kept in a hash table (or some other data structure) and the original records are scanned and dropped immediately.

For these reasons, SparkSQL is considered to be a good representative of state-of-the-art in big data SQL. The configuration parameters of Spark were carefully optimized for the performance experiments. Spark has hundreds of configuration parameters and the process can take hundreds, if not thousands, of experiments to explore the entire configuration space. The work done by Adekoya *et al.* on the same cluster environment provided valuable insight in creating an initial configuration [1]. The initial configuration performed well, but it was found –due to workload differences– that SQL workload performance could be improved by adjusting some of the parameter values.

The data is ingested into Parquet format[2] and stored in HDFS. Apache Parquet is a columnar data format and it is one of the most commonly used data formats for SparkSQL. The binary format alleviates the need to parse text records at query runtime. The format also uses robust bit-packing and dictionary encoding techniques to minimize file sizes and reduce I/O volume. On top of that, the data pages themselves

[1] Project Tungsten. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`. Accessed January 29, 2021.

[2] Apache Parquet. `https://parquet.apache.org/`. Accessed January 29, 2021.

can also be compressed using lightweight compression algorithms such as Snappy[3] to further improve I/O performance. Apache Spark has built-in Parquet readers and writers which are highly optimized for the compute engine. Since the Relation DDO does not use any compression or encoding techniques to reduce file sizes, the experiments were repeated using two configurations:

- Snappy-compressed Parquet with default dictionary and bit-packing parameters, and
- Uncompressed Parquet with dictionary encoding and bit-packing disabled.

Apache Spark 3.0 offers SparkSQL improvements over Spark 2.0, largely due to dynamic execution and optimization. Spark 3.0 allows query plans to be re-evaluated mid-execution and optimized again using actual statistics about the intermediary results. For example, a *sort-merge join* could be converted into a *broadcast join* if one of the join inputs is smaller in size than a certain threshold value. Small shuffle blocks could also be coalesced to reduce task overheads. While this may seem to add latency for smaller workloads, the benefit for larger workloads is evident and the impact on small scale workloads is not substantial. Spark versions 2.4.5 and 3.0.2 (in standalone mode) were both evaluated to highlight the significance of these improvements.

5.4.2 Hive

Hive is a structured data warehousing and query engine, originally developed for MapReduce [34]. Tables in Hive are stored in HDFS and metadata is stored separately in a lightweight central database. Hive supports three different compute engines: Hadoop MapReduce, Spark, and Tez. The MapReduce configuration is deprecated in the current version and could be removed in the next major release of Hive. For Spark, Hive only supports Spark 2.x. Hive version 2.3.8 was used for evaluation. Spark version 2.4.5 and Tez version 0.9.2 were used as execution engines. YARN version 2.7.6 was used as this is the only supported configuration for the Hive-on-Spark configuration, and Tez exclusively runs on YARN.

The configuration parameters were carefully optimized. In particular, *vectorization* and *cost-based optimizations* proved to substantially improve performance. *Vectorization* allows columnar aggregations to be computed more efficiently using batches of column values. *Cost-based optimizations* are applied to query plans based on statistics stored in the Hive *metastore*.

For comparable results with the Spark configurations, the Parquet format was also used to persist the data. Partitioning and bucketing was not used in the performance experiments for the following reasons:

- Most of the TPC-H and TPC-DS queries use filters on key columns (filter by date key for example) which require creating a huge number of files, one for each unique key value, and most likely will have a negative impact on performance. An attempt was made to partition the tables but the job would usually run out of memory while trying to shuffle and re-organize the entire dataset.

[3]Snappy. `https://github.com/google/snappy`. Accessed May 1, 2021.

- The support for bucket joins and other optimization techniques related to bucketing are not fully implemented as noted by Costa *et al.* [13].

The experiments were focused on highlighting differences in query planning and optimization between SparkSQL and Hive. For the Spark engine, Hive uses a handful of simple primitives such as *foreachPartition* and *union* to implement the data processing logic. This makes it difficult for the compute engine to optimize the underlying logic as it is not exposed to the engine. For the Tez configuration, Hive has more control over the compute and data planes due to the bare-bones nature of Tez.

5.4.3 SparkSQL on Hive

SparkSQL can interface with the Hive *metastore* and obtain information about the location of data files and their structure. The data is ingested by Hive and is later used by SparkSQL to compute the results. The results are expected to be similar to Spark 2.0, with minor differences mainly due to different data ingest procedures.

5.5 Performance Metrics

The performance metrics used in this evaluation attempt to measure system efficiency. The scope of efficiency in this thesis is concerned with how well compute resources are utilized. For big data clusters, it is common to allocate resources spatially; e.g. CPU cores [21, 37]. Efficient resource utilization, in this scope, can be defined as the amount of useful work done in unit time using some allocation of resources, relative to some theoretical upper bound. According to this definition, if a system a is able to finish a given workload in less time (or using a "smaller" allocation of resources) than a system b, then it can be deduced that system a is more efficient than system b, since a unit of resource allocation under system a has higher throughput of useful work.

The experiments conducted in this evaluation were performed on a fixed-size cluster; i.e. the resource allocation is constant. The main metric used is query time. Other derived metrics such as speed-up, mean (geometric) query time, and total benchmark time are also used.

5.6 Implementation Correctness

The base system and Relation DDO implementation were tested using unit tests and integration tests on many parts of the system. However, these tests do not provide full coverage of the entire code. The results of the benchmark queries were compared against the published answer sets for both benchmarks. The published answer sets are for the 1 GB "validation" dataset scale. Further attempts to validate implementation correctness on large scales was also done by comparing the query answers produced by the Relation DDO against those produced by SparkSQL.

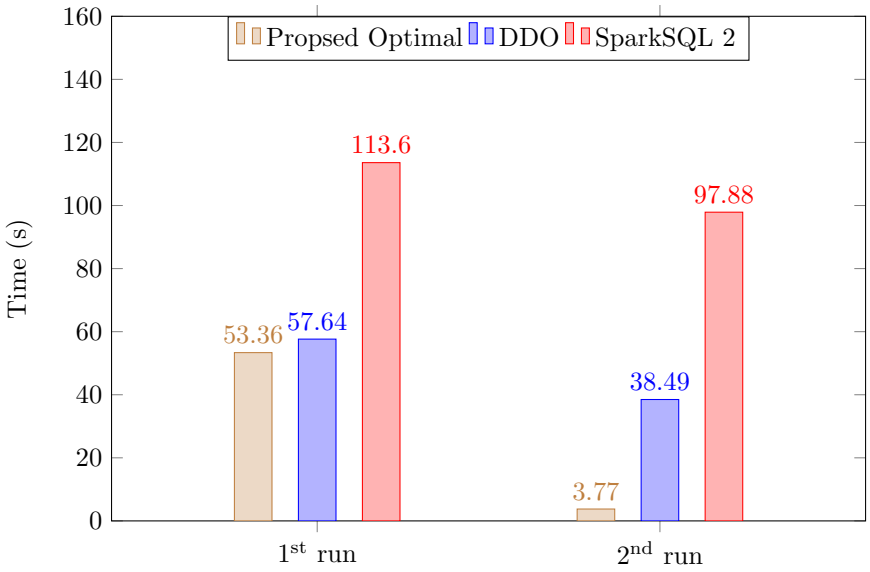


Figure 5.1: Comparison of system efficiency. The dataset size is 300GB and contains 3 billion records. The query filters out roughly half the dataset and performs many aggregations over a few attributes to test code generation optimizations. Comparing the DDO system to the optimal shows how well the design performs just by addressing inefficiencies in the data model alone. The system buffers were cleared before the first run but not before the second. Legend entries correspond to bar order. Best viewed in color.

At the time of writing, there are a few known (and possibly some unknown) bugs in the base system and the Relation DDO that affected some queries from the TPC-DS suite. The running times of these queries are not documented in this thesis. Only the running times of queries producing correct answers are considered. It is also assumed that fixing these bugs will not affect the results documented here.

5.7 Results

5.7.1 Evaluation of System Efficiency

A special query was used to highlight the inefficiencies in current state-of-the-art systems. The query was run on the SparkSQL baseline as well as the DDO system. In addition to that, a highly optimized, hardcoded version of the query was written. The "proposed optimal" query is intended to represent the performance of an ideal system, with respect to the issues discussed. Simple experiments such as this can show the upper bounds of performance while also examining what an ideal system design may need to support in terms of data distribution and execution. Figure 5.1 shows the running times for the proposed optimal implementation and the SQL-equivalent of this query (described in Figure 5.2) using SparkSQL and the Relation DDO.

Both the Relation DDO and the proposed optimal cases can minimize data access requests. However, the proposed optimal query is much more computationally efficient than the DDO case. Implementing code generation can solve this problem. When the main bottleneck is disk access (1st run), the Relation DDO is

```
SELECT
    a,
    b,
    COUNT(*),
    SUM(attr1),
    SUM(attr2),
    SUM(attr3),
    AVG(attr1),
    AVG(attr2),
    AVG(attr3),
    MIN(attr1),
    MIN(attr2),
    MIN(attr3),
    SUM(attr1 + attr2),
    SUM(attr2 + attr3),
    SUM(attr1 + attr3),
    SUM(attr1 + attr2 + attr3)
FROM
    t1
WHERE
    r100000 < 50000
GROUP BY
    a, b
ORDER BY
    a, b;
```

Figure 5.2: SQL listing of a query used to highlight efficiency issues.

similar to the proposed optimal case, showing that disk access requests were minimal in both cases. When the disk bottleneck is removed (via system buffers in the 2nd run), the difference in computational efficiency is revealed. Apache Spark scans the entire dataset in both runs.

This experiment was crafted to highlight two main issues: (1) inefficiencies in the data model which are largely addressed in the DDO system, and (2) computational inefficiencies. The latter is even more surprising. Even though SparkSQL implements code generation and should be the more computationally efficient alternative compared to the Relation DDO from a theoretical standpoint, it is evident that an inefficient data model has the potential to make computational optimizations largely imperceptible. I/O components are the slowest components in a computer system, and efficient use of these critical resources should be the main concern of a data processing framework. This is especially true in distributed big data as it involves storing huge quantities of data on slow persistent storage and transferring huge quantities of data over slow mediums.

5.7.2 TPC-H Benchmark

The benchmark queries were run at different scales of the dataset. Scale factors 10, 30, 100, and 300 were used to evaluate the systems. A scale factor of 1 roughly corresponds to 1 GB of data. For all systems, the data was distributed evenly (in terms of size) among all worker nodes. A summary of the results is presented in Table 5.1. Note that for both the Hive-on-Spark and Hive-on-Tez configurations, queries 2, 8, 9, 11, 13, 15, 16, 21 and 22 were not run due to limitations in the query engine. Detailed results can be found in Appendix

Table 5.1: Summary of TPC-H results. Values are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Mean is calculated using geometric mean.

		DDO	SparkSQL 2	SparkSQL 2 (compressed)	SparkSQL 3	SparkSQL 3 (compressed)	Hive on Spark 2	SparkSQL 2 on Hive	Hive on Tez
SF-10	Mean	4.24	6.45	5.04	4.87	**3.51**	20.60	8.22	35.39
	Total	148.32	179.39	135.73	126.65	**89.97**	N/A	229.48	N/A
	STDEV	6.40	6.41	3.86	3.33	2.34	15.85	6.95	44.12
SF-30	Mean	**6.30**	14.69	10.33	10.23	7.31	30.13	14.41	50.79
	Total	228.80	385.36	300.34	253.49	**196.06**	N/A	381.59	N/A
	STDEV	9.55	11.11	9.93	5.15	6.04	27.28	9.97	63.04
SF-100	Mean	**16.62**	57.89	35.32	46.01	27.69	87.64	46.46	85.14
	Total	**574.30**	1610.92	1040.25	1236.10	825.42	N/A	1295.51	N/A
	STDEV	21.12	50.15	39.76	36.56	32.86	103.39	40.24	96.29
SF-300	Mean	**44.12**	261.59	165.44	227.70	133.65	279.98	175.69	194.12
	Total	**1550.23**	8100.52	5656.24	6793.76	4360.86	N/A	5966.24	N/A
	STDEV	58.25	287.68	233.99	239.75	174.40	313.17	250.19	237.63

A.

Small Scale Results

For the 10 GB and 30 GB data sizes, the intent is to show the responsiveness of each system and highlight its suitability for interactive workloads in a resource-abundant environment. SparkSQL 3 using compressed Parquet offers the smallest mean (geometric) query time as well as total benchmark time for the 10 GB data size. At 30 GB, the Relation DDO offers the least mean query time but SparkSQL 3 on compressed Parquet still holds the minimum for total benchmark time.

The Relation DDO lacks the compute efficiency of code generation as well as having an overall preference for algorithms that perform better on large data sizes. The latter is evident in TPC-H queries 7 and 8, for which the Relation DDO creates a large number of shuffle blocks and performs substantially worse than the majority of baselines. Using the namespace metadata, the shuffle operation could be configured to produce a smaller number of shuffle blocks corresponding to the size of the DDO. This is not currently implemented in the Relation DDO. The shuffle operation will always split a DDO part into a fixed number of blocks depending on the number of configured workers.

Medium and Large Scale Results

For the 100 GB data size, the dataset can fit entirely in memory and the major bottlenecks are expected to be compute and network related. Performance improvements mainly due to DDO part co-location and optimized replicas start to show in the 100 GB case. The Relation DDO shows the minimum in both mean query time and total benchmark time.

For the 300 GB data size, the dataset is almost twice the size of total cluster memory. The benchmark at this scale highlights the system's ability to use I/O channels efficiently. Figure 5.3 shows the relative speed-up of the Relation DDO compared to all baselines for the 300 GB case.

Compared to SparkSQL 3 on compressed Parquet (the second best alternative), the Relation DDO offers 3x improvement in mean query time and 2.8x improvement in total benchmark time. For TPC-H queries 6, 15, 16, and 18 the Relation DDO achieves more than an order of magnitude improvement compared to SparkSQL 3 on compressed Parquet. The benefits due to DDO part co-location and optimized replicas continue to show scalability as data volumes continue to grow. This trend is expected to continue with increasing data sizes. Figure 5.4 shows the scalability trends for all systems.

Effect of Data Compression

The architectural improvements offered by the DDO system focus mainly on reducing the imposed load on bandwidth-constrained I/O channels. In Spark, the same could be achieved (to a limited extent) by using data compression. Compressed data can be transferred more efficiently at the expense of the extra compute work required to compress and decompress the data. To test this effect, SparkSQL was evaluated using both compressed and uncompressed Parquet as noted in Section 5.4.1.

For the 100 GB and 300 GB data sizes, the use of compressed Parquet improves both mean query time and total benchmark time by roughly 1.5x. The Relation DDO on the other hand does not use any encoding or data compression mechanisms. However, these techniques are still applicable to the Relation DDO and could further improve its performance.

SparkSQL 2 vs. SparkSQL 3

SparkSQL 3 offers improvements over SparkSQL 2 mainly due to dynamic execution and optimization techniques as discussed in Section 5.4.1. The use of these options gives a clear advantage for SparkSQL 3 as seen in Table 5.1. For the 100 GB and 300 GB data sizes, the improvement over SparkSQL 2 is $\approx$ 15%-27% in mean query time and $\approx$ 19%-30% in total benchmark time.

SparkSQL 2 on Hive vs. Hive on Spark 2

Comparing SparkSQL and Hive query engines reveals differences in query planning, optimization, and execution styles. In both cases, the data was previously ingested into Parquet by Hive. Using the Hive *metastore*, SparkSQL can discover the schema and table objects and run queries on them as in the SparkSQL standalone cases. SparkSQL is able to encode the needed transformations using a wide variety of RDD compute primitives. This allows the Spark execution engine to optimize compute operations and pipeline them using code generation when possible.

On the other hand, Hive's query engine manually implements most of the compute logic through simple RDD primitives such as *foreachPartition* and *union*. This obscures the query plan and makes it difficult for

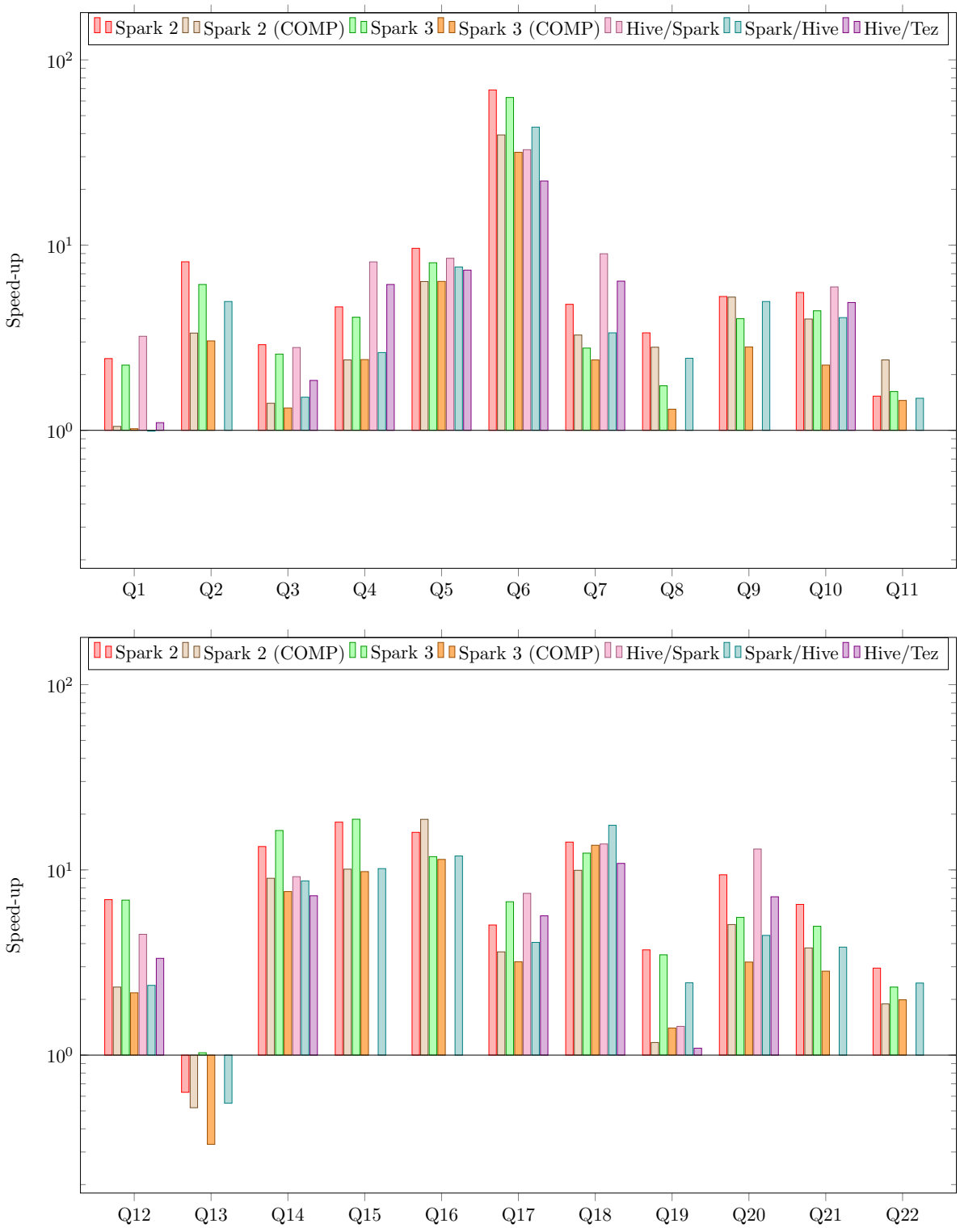


Figure 5.3: Relative speed-up of the TPC-H queries using the Relation DDO. Speed-up = baseline time / Relation DDO time. Only the query times for the 300 GB scale are considered. Note that queries 2, 8, 9, 11, 13, 15, 16, 21, and 22 are not available in the Hive configurations. Legend entries correspond to bar order. Best viewed in color.

the execution engine to perform optimizations. Furthermore, Hive allocates YARN containers based on data size alone; i.e. not all compute nodes are used in the 10 GB and 30 GB cases. Compared to Hive's query engine, SparkSQL offers a 2.5x and 1.6x improvement in mean query time for the 100 GB and 300 GB data sizes, respectively.

SparkSQL offers a wider coverage of SQL features than Hive's query engine. SparkSQL manages to run all 22 queries of the TPC-H benchmark, while Hive is unable to support 5 queries and mistakenly decides to perform a cartesian product in 4 other queries. An attempt was made to run queries containing cartesian product. However, the Spark YARN containers would fail producing an out-of-memory error. SparkSQL manages to correctly identify the join operations and/or flatten inner queries, avoiding the need for a cartesian product.

Hive on Spark 2 vs. Hive on Tez

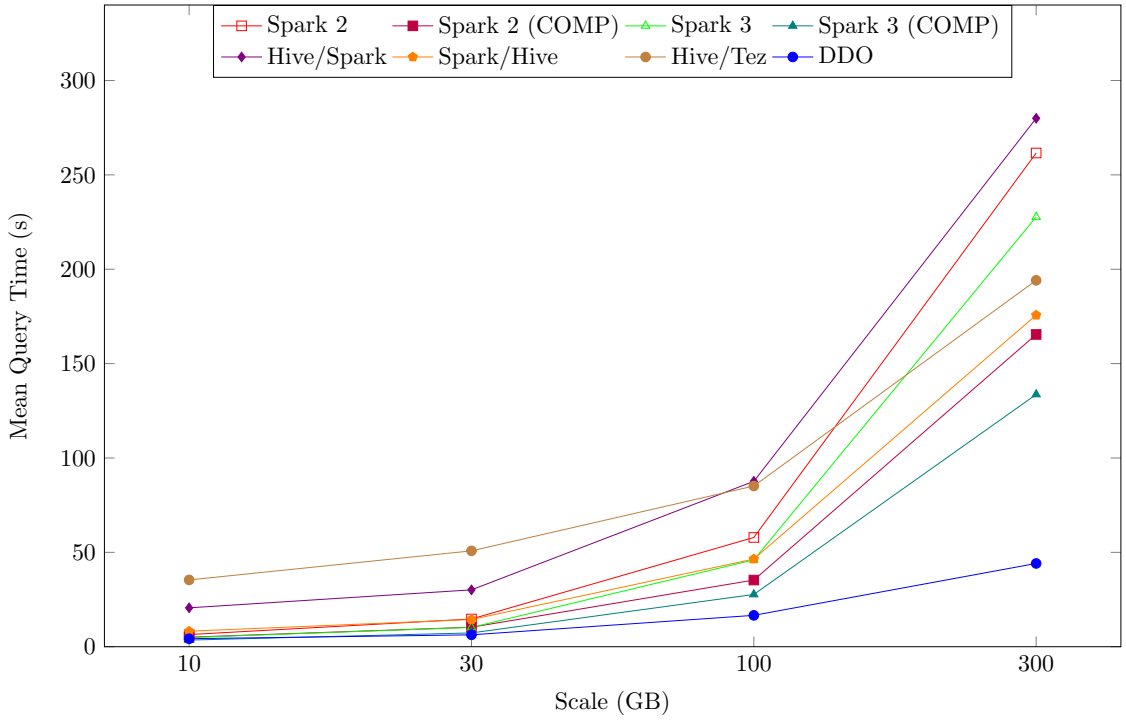
Hive supports both Spark 2 and Tez as execution engines. Using Tez, Hive implements most of the functionality using the DAG API. Hive on Tez improves mean query time by $\approx 44\%$ compared to Hive on Spark 2 at the 300 GB data scale. For the 100 GB data size the results are similar for the majority of queries and mean query time is almost identical. Hive on Tez performs worse than Hive on Spark 2 in the 10 GB and 30 GB cases.

It is difficult to speculate exactly why these performance differences exist. However, one important feature of the Tez framework is the ephemeral nature of its containers. Spark allocates YARN containers that are used for the entire duration of the Hive session. The containers are allocated once just before executing the first query. The allocated containers are then reused for subsequent queries and until the session terminated. In the Tez configuration, containers are allocated by the framework on a per-task basis. This could lead to some scheduling delays in YARN. The ephemeral nature of the containers, however, is more suited for multi-tenant environments.

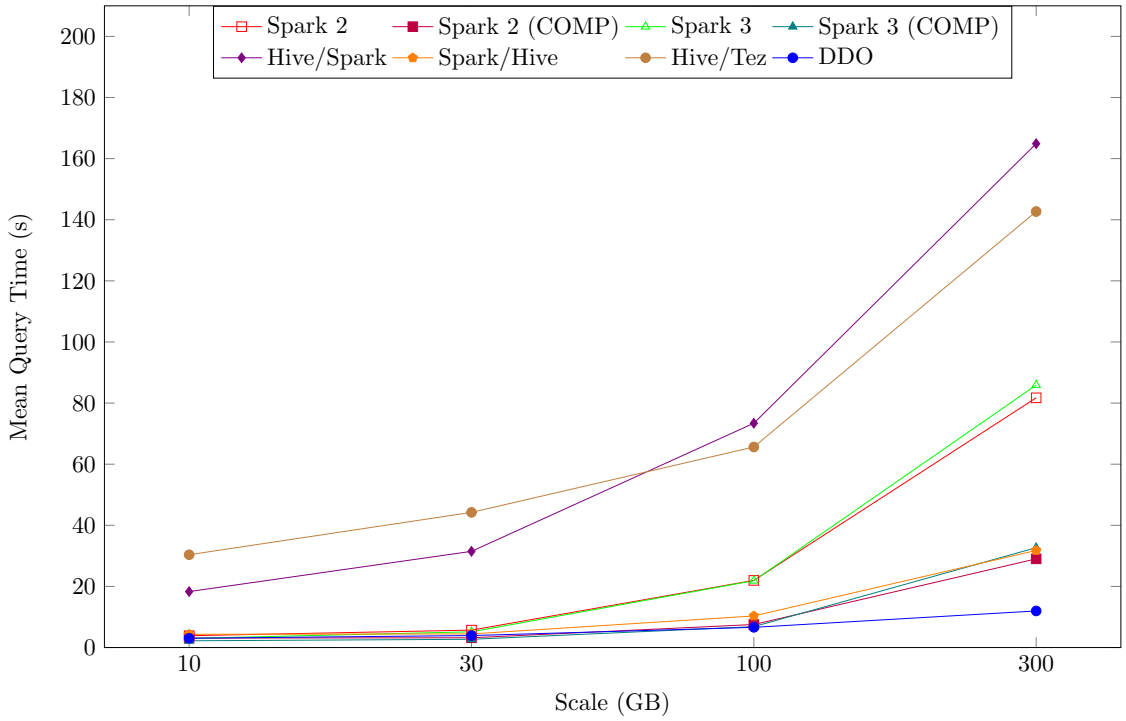
5.7.3 TPC-DS Benchmark

The benchmark queries were run for the same scales of the dataset as the TPC-H benchmark and using an even distribution (in terms of size). Compared to the TPC-H benchmark, TPC-DS queries focus on answering lighter, more-specific questions; e.g. aggregating sales from a particular quarter in a particular year. The benchmark focuses on the ability of the system to extract slices of data efficiently.

Only 18 out of the 99 queries were evaluated in the TPC-DS benchmark. SparkSQL was able to support 73 out of the 99 queries, while the Hive query engine was only able to support 37 queries. The features implemented in the Relation DDO for the TPC-H benchmark were able to support 18 of the TPC-DS queries. The Relation DDO and the majority of baselines show features comparable to the TPC-H benchmark. A summary of the results is presented in Table 5.2. Note that for both Hive-on-Spark and Hive-on-Tez configurations, queries 41, 62, 96, and 99 were not run due to limitations in the query engine. Detailed



(a) Mean TPC-H query time



(b) Mean TPC-DS query time

Figure 5.4: Mean query time for TPC-H and TPC-DS benchmarks. The scalability trends highlight each system's ability to handle larger volumes of data gracefully. Mean is calculated using geometric mean. Best viewed in color.

Table 5.2: Summary of TPC-DS results. Values are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Mean is calculated using geometric mean.

		DDO	SparkSQL 2	SparkSQL 2 (compressed)	SparkSQL 3	SparkSQL 3 (compressed)	Hive on Spark 2	SparkSQL 2 on Hive	Hive on Tez
SF-10	Mean	3.04	3.87	3.09	2.99	**2.21**	18.31	4.33	30.37
	Total	133.20	84.78	68.00	62.62	**45.22**	N/A	98.13	N/A
	STDEV	8.72	2.88	2.00	1.97	1.13	13.22	3.37	10.17
SF-30	Mean	3.92	5.70	3.20	5.15	**2.70**	31.48	4.46	44.23
	Total	163.20	127.03	71.94	130.74	**61.20**	N/A	113.47	N/A
	STDEV	9.35	4.32	2.39	7.97	2.37	17.40	5.90	17.58
SF-100	Mean	**6.61**	21.97	7.54	21.87	6.84	73.40	10.34	65.63
	Total	233.18	512.59	**179.60**	550.86	182.31	N/A	299.02	N/A
	STDEV	13.93	18.02	8.42	22.82	10.71	37.50	22.02	27.38
SF-300	Mean	**11.97**	81.72	29.02	85.89	32.68	164.89	31.78	142.68
	Total	**377.76**	2306.69	911.47	2506.55	1149.44	N/A	1001.55	N/A
	STDEV	16.17	110.34	62.28	129.46	85.81	95.81	67.60	86.41

results be found in Appendix A.

Small and Medium Scale Results

For the 10 GB and 30 GB data sizes, SparkSQL 3 using compressed Parquet shows the best overall mean query time and total benchmark time. The Relation DDO offers a slight improvement in mean query time at the 100 GB data size. Surprisingly, the best total benchmark time is achieved by SparkSQL 2 on compressed Parquet, not SparkSQL 3. However, the difference between the two is negligible at this scale.

Large Scale Results

For the 300 GB data size, the Relation DDO shows a substantial improvement over all baselines. Figure 5.5 shows the relative speed-up of the Relation DDO compared to all baselines for this data scale. Compared to SparkSQL 2 on compressed Parquet (the second best alternative), the Relation DDO shows roughly 2.4x improvement in both mean query time and total benchmark time. Comparing to the uncompressed case, however, reveals a much more substantial speed-up of more than 6x in both mean query time and total benchmark time. Figure 5.4 shows the overall scalability trends for all systems.

Penalty of Dynamic Execution

Dynamic execution and optimization fails to improve the query times of SparkSQL 3 when compared to SparkSQL 2 in both the compressed and uncompressed cases. This demonstrates the adverse effects imposed by the barrier synchronization required to inspect and optimize query plans mid-execution. In dynamic

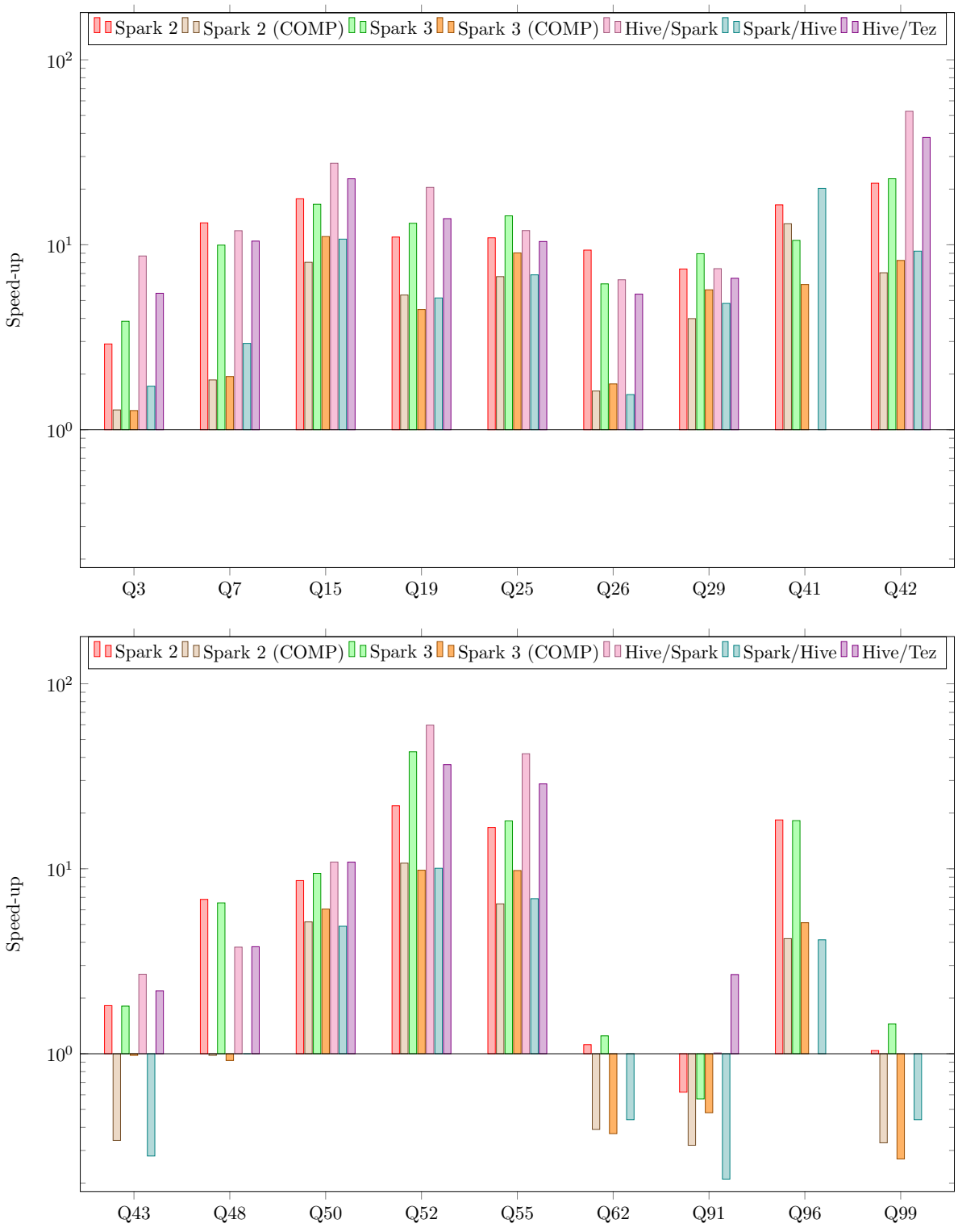


Figure 5.5: Relative speed-up of the TPC-DS queries using the Relation DDO. Speed-up = baseline time / Relation DDO time. Only the query times for the 300 GB scale are considered. Note that queries 41, 62, 96, and 99 are not available in the Hive configurations. Legend entries correspond to bar order. Best viewed in color.

execution mode, Spark jobs have extra stages that are executed only at the *driver* node. These extra stages collect measurements from the intermediary results to decide whether the current query plan will need to be modified.

5.8 Discussion

5.8.1 Significant Speed-ups

This section discusses the reasons behind significant speed-ups offered by the Relation DDO. These reasons can be summarized as follows:

- **DDO part co-location** helps reduce the compute work and network delays by eliminating the need to shuffle data due to workload-aware data placement that was done during data ingest.
- **Optimized replicas** can be used to optimize individual queries, as each query plan may choose its preferred replicas.
- **Broadcasted Tables** are useful in cases where small DDOs are replicated on all nodes to reduce latency.
- **Streamlined execution** by first analyzing the query plan to determine an optimal task execution order, and later allowing flexible scheduling to reorder tasks based on run-time data availability.

DDO Part Co-location

One of the main improvements offered by the DDO system is the ability to delegate the placement of DDO parts to their respective DDO controller implementations. This allows each DDO implementation, which is aware of its workload needs, to decide how it wants to replicate and co-locate any blocks from any of its DDOs. In the Relation DDO this is used to optimize frequently joined tables. During ingest, each table has its records shuffled by some user-defined key attribute. DDO parts containing records having possibly matching key values are guaranteed to be co-located, and DDO parts containing records that are guaranteed not to have any joinable records are not necessarily co-located; i.e. the constraint is only enforced on DDO parts with a possibility of having "joinable" records. This allows joins to be performed without having an initial pass to scan and shuffle the records.

TPC-H query 18 benefits greatly from this technique as shown in Figure 5.7. The inner query shown in the listing in Figure 5.6 can be converted to a join between tables *lineitem* and *orders*. If all records with matching keys are co-located, then the join and filter can be computed immediately without any preparation. Similar benefits apply to TPC-H queries 3, 4, 7, 8, 9, 10, and 12 with varying effects. For example, TPC-H query 4 does not show as impressive a speed-up as query 18 due to the fact that it has filters on records input to the join operation. These two filters greatly reduce the size of the data that needs to be shuffled,

```
-- TPC-H Large Volume Customer Query (Q18)

SELECT
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    SUM(l_quantity)
FROM
    customer,
    orders,
    lineitem
WHERE
    o_orderkey IN (
        SELECT
            l_orderkey
        FROM
            lineitem
        GROUP BY
            l_orderkey HAVING
                SUM(l_quantity) > 300
    )
    AND c_custkey = o_custkey
    AND o_orderkey = l_orderkey
GROUP BY
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
ORDER BY
    o_totalprice desc,
    o_orderdate;
```

Figure 5.6: SQL listing of TPC-H query 18. The inner query is converted to a join between tables *lineitem* and *orders*. DDO parts of both tables are co-located for optimized join computation.

obscuring the benefits of object co-location. Query 4 is also much simpler, having only one join and a *count* aggregation.

Optimized Replicas

Optimized replicas are replicas of DDO parts having the same data but not necessarily the same data representation. The Relation DDO makes use of this by having replicas of DDO parts, each sorted by a different user-defined attribute. This was referred to as *densely-indexed replicas* in Chapter 4. These replicas can be used to optimize a *sort-merge join* by skipping the sorting step, or they can be used to perform a binary search on the sorted attribute to determine a range of records instead of scanning the entire DDO part.

For TPC-H query 6 (described in Figure 5.8), the Relation DDO achieves a reduction in running time by more than an order of magnitude compared to all baselines. Using a replica indexed by filter attributes allows the DDO to read only the records that are needed for the remainder of the query; i.e. only the records from the indicated date range are considered. This reduces both the disk access and compute work. Larger

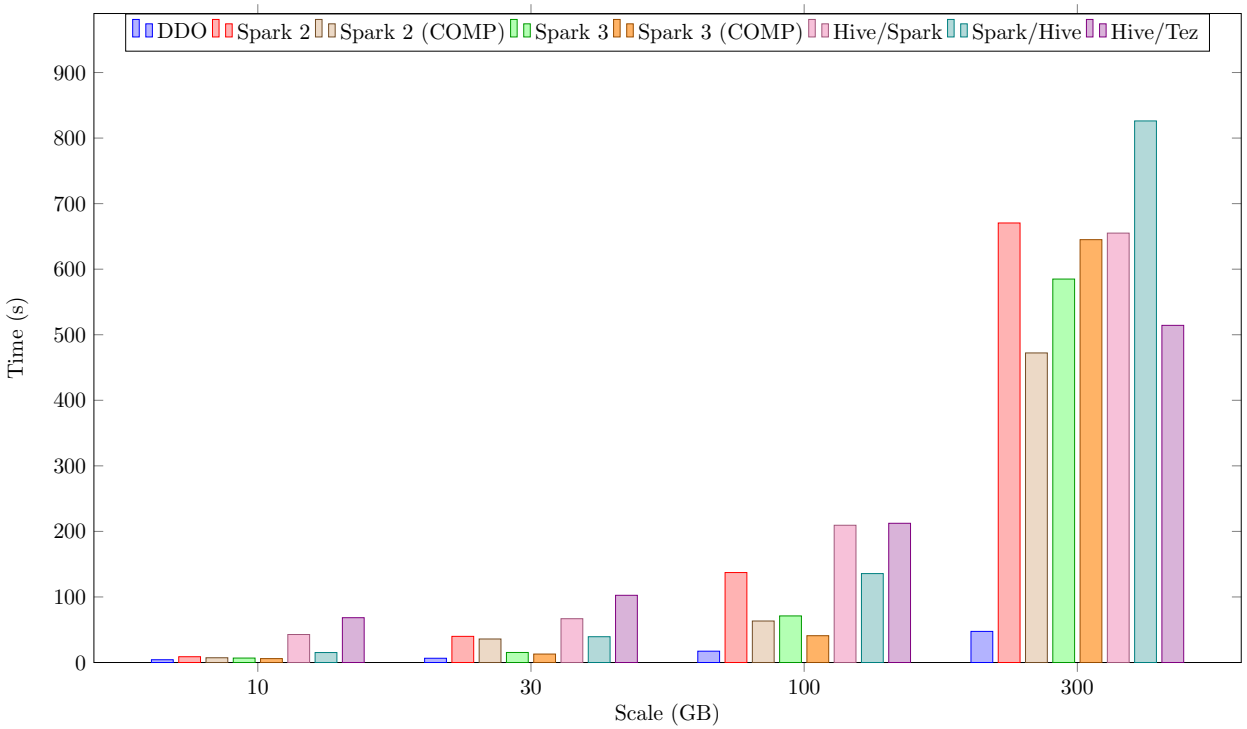


Figure 5.7: Effect of DDO part co-location on TPC-H query 18. The DDO system manages to keep execution times significantly lower due to reduced network usage. Legend entries correspond to bar order. Best viewed in color.

data scales further highlight the significance of this optimization as shown in Figure 5.9. The majority of the TPC-H and TPC-DS query plans heavily utilize this technique to keep I/O utilization at a minimum. This is applicable to many analytical SQL queries as they usually investigate slices of data and not the entire dataset. While this could be implemented on other systems by using clever data formats, having fine-grained control over storage allows multiple replicas of the same DDO (each suited for a range of queries) to be stored and used interchangeably in case of node failures.

Broadcasted Tables

Not all DDOs are large. Some are reasonably small and used frequently. The Relation DDO allows ingesting and replicating some tables on all workers. For example, the TPC-DS schema contains 14 dimension tables, each contains a few thousand records. Some are relatively large (1 or 2 million) but they are still manageable by a single worker. Almost all of the implemented TPC-DS queries benefit from this. In fact, this makes entire queries embarrassingly parallel, as no co-operation at all is needed from other workers.

A good example of this is TPC-DS query 15. Tables *customer*, *customer_address*, and *date_dim* are replicated on all workers. As seen in the listing in Figure 5.10, the join and aggregation done over the DDO parts of *catalog_sales* can be computed in parallel on all workers, independently of each other. The results are shown in Figure 5.11. The query plan also selects an optimized replica of table *catalog_sales* to further

```
-- TPC-H Forecasting Revenue Change Query (Q6)

SELECT
    SUM(l_extendedprice*l_discount) AS revenue
FROM
    lineitem
WHERE
    l_shipdate >= date '1994-01-01'
    AND l_shipdate < date '1994-01-01' + interval '1' year
    AND l_discount between 0.06 - 0.01 AND 0.06 + 0.01
    AND l_quantity < 24;
```

Figure 5.8: SQL listing of TPC-H query 6. The filter predicate is optimized using a densely-indexed replica to reduce I/O and compute time.

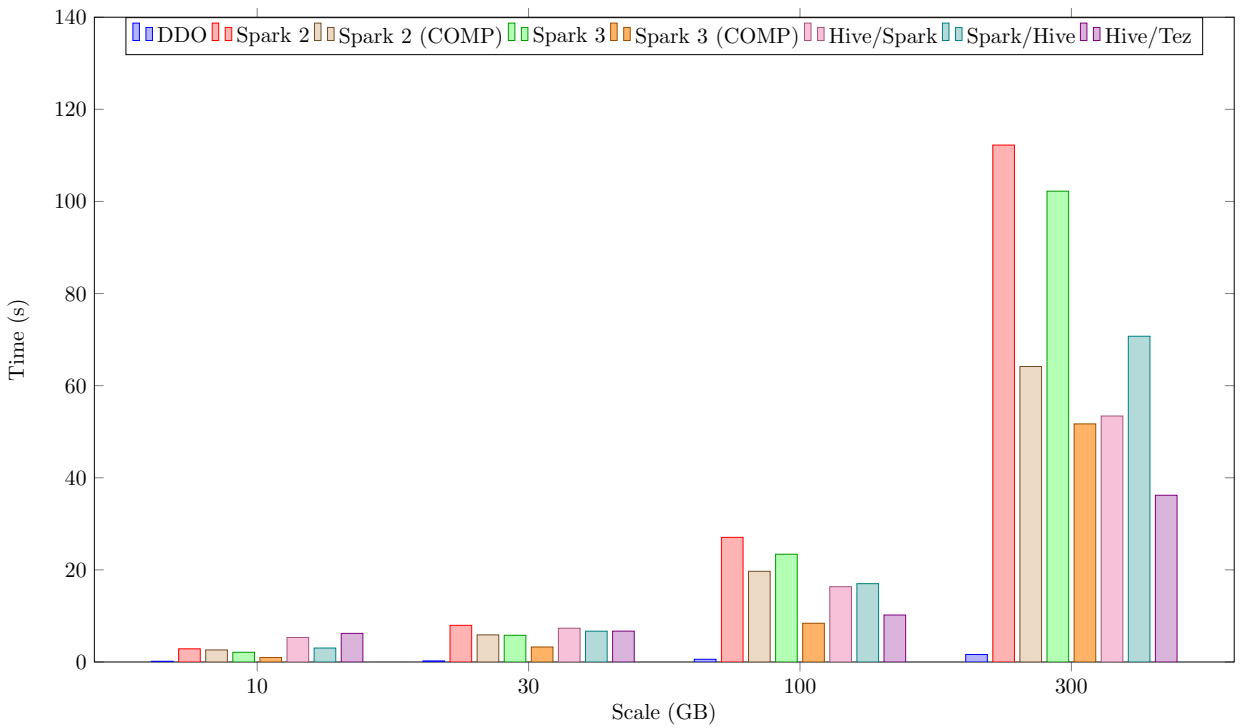


Figure 5.9: Effect of replica optimization on TPC-H query 6. The Relation DDO considers a limited range of records from a suitable replica to reduce disk and compute activity. Legend entries correspond to bar order. Best viewed in color.

```
-- TPC-DS Query 15

SELECT TOP 100
    ca_zip,
    SUM(cs_sales_price)
FROM
    catalog_sales,
    customer,
    customer_address,
    date_dim
WHERE
    cs_bill_customer_sk = c_customer_sk
    AND c_current_addr_sk = ca_address_sk
        AND c_current_addr_sk = ca_address_sk
    AND c_current_addr_sk = ca_address_sk
    AND (SUBSTR(ca_zip,1,5) IN ('85669', '86197', '88274', '83405', '86475',
                                '85392', '85460', '80348', '81792')
        or ca_state IN ('CA','WA','GA')
        or cs_sales_price > 500
    )
    AND cs_sold_date_sk = d_date_sk
    AND d_qoy = 2 AND d_year = 2000
GROUP BY
    ca_zip
ORDER BY
    ca_zip;
```

Figure 5.10: SQL listing of TPC-DS query 15. The query benefits from broadcasted dimension tables *customer*, *customer_address*, and *date_dim*.

reduce query time by only considering sales from the second quarter of the year 2000.

Streamlined Execution

The system performs an initial DAG analysis to produce a favorable task execution order. However, the system can reorder tasks during runtime as shuffle and input data blocks become available at arbitrary times. Due to the decentralized nature of task scheduling, each worker is able to reorder tasks based on data availability and age. Tasks using newly produced data blocks are scheduled to the front of the ready queue since their inputs are more likely to be available in memory or even processor cache.

TPC-H query 5 is extremely complex and joins data from six different tables. The query plan consists of multiple successive stages each relying on the output of the previous stage. Only one input table is filtered and a suitable densely-indexed replica is chosen to perform index-assisted filter. Other than the first filter operation, there seems to be no other optimizations that can be done. However, experimental results show the system's ability to handle large intermediate results gracefully, even at the 300 GB scale. This can be attributed to the overall efficiency of the asynchronous data pipeline and flexible execution policies.

5.8.2 Moderate Speed-ups

More moderate speed-ups are observed in some queries. In most cases, this is due to improvements that are obscured by other (unoptimized) components of the system and/or hardware limitations. This results in

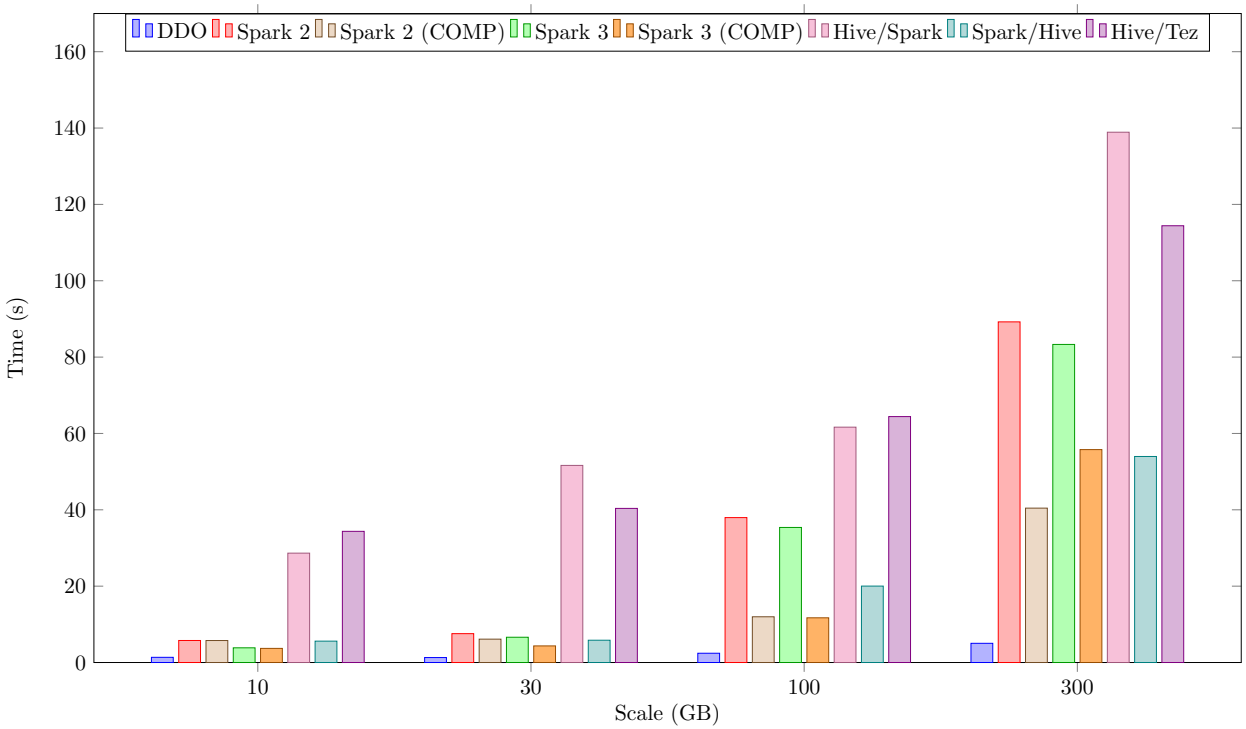


Figure 5.11: TPC-DS query 15 results. The query benefits from both an optimized replica to assist with the filter predicates, and broadcasted DDO part replicas to avoid shuffling data. Legend entries correspond to bar order. Best viewed in color.

behaviours that are not immediately obvious. This section discusses the reasons behind these observations and how they can be addressed in the future.

Hardware Limitations

Another possible performance improvement due to replica optimization is record pre-grouping. The Relation DDO allows replicas to be grouped by one or more user-defined attributes. This alleviates the need to perform a grouping step before computing the aggregation. However, with code generation the grouping and aggregation steps could be merged together in one compute step.

TPC-H query 1 runs on the largest table in the schema (*lineitem*) and has a filter predicate that removes less than 5% of the data. The query has no joins and therefore does not require a shuffle. In SparkSQL, the entire query is a single stage and benefits greatly from code generation in the smaller scales. The DDO system uses a *grouped densely-indexed* replica to improve the execution time of this query. However, the advantage slowly diminishes with increasing data sizes as seen in Figure 5.12. This is due to the nature of the task which is mostly limited by processor performance and not software architecture. In both cases, 95% of the records from the table *lineitem* will have to be scanned and aggregated.

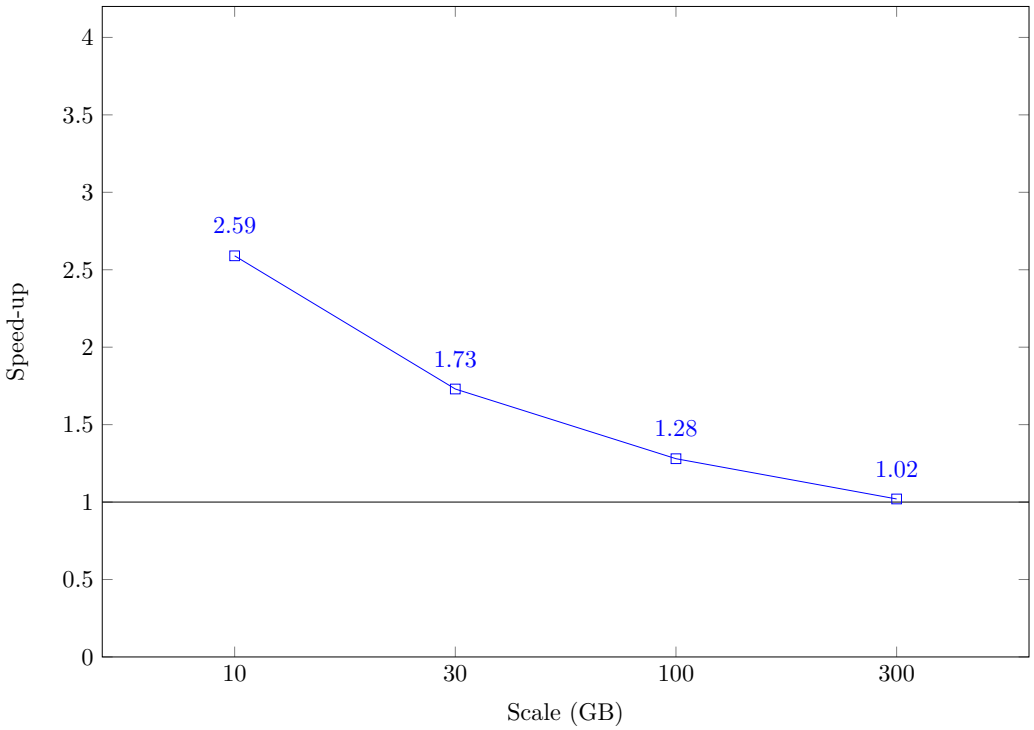


Figure 5.12: Speed-up of TPC-H query 1 due to pre-grouping. The speed-up is in comparison to SparkSQL 3 using compressed Parquet (the best baseline for the TPC-H suite). Speed-up = baseline time / Relation DDO time.

Competing Factors

Another query for which the Relation DDO exhibits decreasing speed-up as the data size increases is TPC-H query 3. Query 3 is somewhat similar to query 1 in that it operates on large inputs. The main difference is that the query has two joins, one of which requires shuffling while the other shuffle is optimized away due to DDO part co-location. The remaining shuffle step somewhat obscures the benefit at the 10 GB and 30 GB scales as there are tens of thousands of small shuffle blocks that need to be collected. Compared to the Relation DDO, SparkSQL 3 can dynamically reduce the number of shuffle blocks by coalescing sufficiently small blocks. The speed-up is maximized at the 100 GB scale as the shuffle blocks are larger than the coalesce threshold set for Spark. However, the speed-up decreases significantly at the 300 GB scale as both systems are impeded by the increasing size of data transfer over the network. The speed-up is expected to continue to decrease with increasing data sizes. It should be noted that the asynchronous operation of the DDO system can hide network delays in cases where the shuffle size is sufficiently small and there are available compute tasks to execute while DDO parts are being transferred, which is most evident in the 100 GB scale as shown in Figure 5.13.

TPC-H query 16 has an relatively complex filter predicate. Only one of the filter attributes matches an optimized replica which can be used to reduce data loading. However, due to the complexity of the remaining parts of the filter predicate, the overhead of virtual methods (which are used to build expressions) starts to

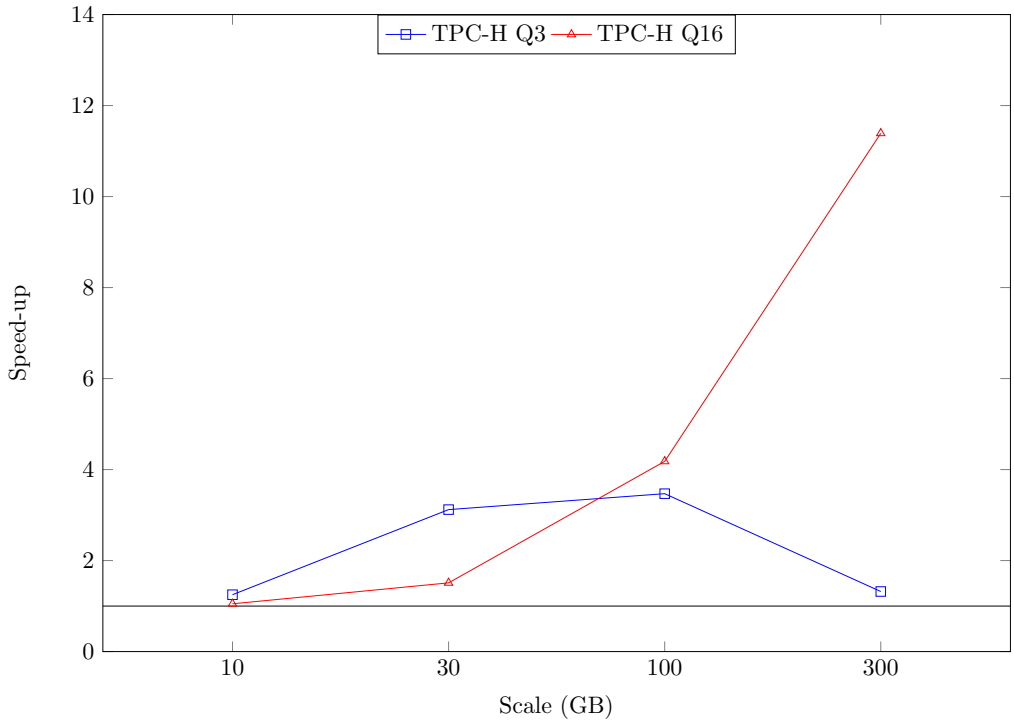


Figure 5.13: Speed-up of TPC-H queries 3 and 16 compared to SparkSQL 3 using compressed Parquet. The speed-up varies as different aspects of both the DDO system and Spark become the limiting factors at different data scales. Speed-up = baseline time / Relation DDO time.

be the main bottleneck. SparkSQL inlines the filter predicate using code generation. The relative benefit of DDO part co-location and optimized replica manifests more clearly with increasing data size as seen in Figure 5.13.

5.8.3 Slowdowns and Drawbacks

At the largest data scale, there are only a handful of queries for which the Relation DDO implementation shows slowdowns compared to the baselines. TPC-DS queries 43, 62, 91, and 99 in particular deal with small intermediary results. One of the most important reasons explaining the slowdown is the fact that most compute primitives (such as *shuffle* and *groupBy*) are implemented with the assumption that the input dataset is large. For example, the compute primitive *groupBy()* only uses a parallel implementation. In cases where the data size is too small, the overhead of splitting the data, waiting for barrier synchronization, and combining the results of multiple threads is greater than immediately solving the problem in the current thread using a single-threaded implementation. This can be mitigated in the future by providing implementations that work better with smaller data and choosing the proper implementation at runtime.

On the other hand, the *shuffle()* primitive creates a large number of intermediary DDO parts that need to be transferred over the network. This works well for large DDO parts as it leverages the entire asynchronous data pipeline of the system. DDO parts are read from disk asynchronously by the I/O executor, the compute

executor splits each part into a specific number (called the *split factor*) of blocks and hands them to an asynchronous sending thread. For smaller DDO parts, the fixed *split factor* yields many DDO parts, some of which may contain very few records or none at all. The DDO part metadata in this case is greater than the actual useful data. The current implementation does not allow the *split factor* to be changed after query planning. However, the query plan could be optimized by choosing an appropriate *split factor* before execution. The namespace contains information about the DDO parts and their sizes which can be used in cost-based optimizations.

For TPC-H query 13 the Relation DDO shows significant slowdowns due to both an unavoidable shuffle step to join two tables, and two successive grouping and aggregation steps that could be pipelined if code generation was implemented. In fact, this query alone highlights the two major areas of the system that require improvement: (1) shuffle performance for small DDOs, and (2) code generation. However, known solutions to these problems are applicable to the Relation DDO but have not been implemented in this proof-of-concept. The overall goal is to showcase sizable improvement that is mainly driven by architectural improvements.

5.9 Summary

The overall average speed-up increases as the data size increases. This characteristic is the main goal the system design was meant to realize: handling large volumes of data gracefully and efficiently using a unified compute and storage interface. At the 300 GB scale, mean query time is improved by at least 3x for the TPC-H benchmark, and 2.4x for the TPC-DS benchmark. For a considerable portion of queries from both benchmarks, execution time was improved by an order of magnitude or more. A considerable portion of the speed-up is driven by the use of indices which may not be feasible for some workloads. However, other architectural improvements, such as the customizable DDO part placement, may still be leveraged to implement optimizations for these workloads.

An entirely new system was developed and is not yet optimized and perfected, as is the case with current state-of-the-art baselines. As concluded in Section 5.7.1, an inefficient data model has the potential to make computational optimizations largely imperceptible. The inverse is also relatively true for complex workloads. Without having all components in the system performing optimally, a novel data management technique may not be easily comparable to systems that have undergone rigorous optimizations. However, even without advanced features such as code generation, the results show substantial improvement both to individual queries and the overall mean query time. Other optimizations such as data compression, using vector instructions for aggregation, cache locality optimizations, and optimized memory allocation and management are all still possible.

6 Conclusions and Future Work

6.1 Thesis Summary

The overall general-purpose nature of distributed architectures is not suitable for specialized applications such as data warehousing. In this thesis, a modular distributed architecture was shown to mitigate this issue through the use of pluggable and highly specialized system modules that allow workloads to customize system behaviour. This was shown to have a substantial performance improvement over general-purpose architectures. However, the drawback of this approach is the amount of development work needed to introduce support for new workloads as many system modules will need to be implemented.

Furthermore, efficient use of node-local resources requires some form of resource coupling. Distributed file systems usually do not allow workloads to specify data placement preferences. Resource coupling can address this issue and avoid over-utilizing the cluster network as many distributed algorithms are sensitive to data placement (e.g. SQL joins). Previous works have also hinted at the possibility of various performance improvements through some resource coupling. For example, in-memory caches that are managed by the execution system (which knows about data dependencies through object lineage or other mechanism) perform better compared to generic cache management policies that try to discover access patterns. Moreover, coupling the storage element with the rest of the system can enable efficient fine-grained data access requests.

The architecture proposed in this thesis tightly couples **storage**, **memory**, and **compute** resources to create an efficient asynchronous data processing pipeline. The data pipeline is available for use by many workloads through a unified interface called the *DDO*. In general, the system allows pluggable workload implementation modules to define custom data objects, replication and placement strategies, logical namespaces, and physical query planners.

Using this novel architecture, the *Relation* DDO was implemented to evaluate the system's performance using the TPC-H and the TPC-DS benchmarks. The Relation DDO uses multiple techniques to improve query performance such as *primary key indices*, *densely indexed replicas*, *grouped replicas*, and *broadcasted tables*. Moreover, the asynchronous nature of the data pipeline and the flexibility of task scheduling mechanisms have enabled moderate improvements over the baselines in cases where these optimizations could not be applied.

The system and DDO implementation presented here is only a proof-of-concept and there is still plenty of opportunities for optimization and design improvements. Currently implemented components such as the task scheduler, lineage-based cache manger, and executors could be further optimized. For future work,

the DDO and modular design concepts will likely remain unchanged. Further work will mainly focus on improvements in the design of the base system and implementing new DDOs for a variety of workloads.

6.2 Summary of Contributions

A review of previous works in the literature revealed that current systems lack a few important features:

1. The ability to control system behaviour for specialized workloads,
2. A data object interface that can minimize data access requests across all levels of memory hierarchy, and
3. The ability to indicate co-location and replication constraints to the storage layer in order to minimize data movement for the majority of expected query plans.

Experiments show that resolving these issues can result in roughly 3-6x improvement in mean query time and more than an order of magnitude improvement for some queries. A considerable portion of these improvements were possible due to the structured nature of the data. The benchmarked DDO implementation makes use of an index to optimize data access requests at both the disk and memory levels. This requires the data to be first ingested into the system and prepared (indexed, replicated, etc.). The system offers no added benefit in the cases of one-time jobs. In such cases, it is better to start working on the analytical task immediately rather than waste time ingesting and organizing data for long-term storage. A framework such as Apache Spark would be more suitable in terms of performance, flexibility, and ease-of-use for such use cases.

6.3 Future Work

6.3.1 Columnar Compaction

Columns in the Relation DDO are backed by plain arrays. Using plain arrays can sometimes be wasteful in terms of memory and storage. This is especially true in cases where the number of distinct column values is substantially smaller than the length of the column; e.g. *enum* type columns. Using compaction methods such as dictionary encoding combined with bit-packing and/or run-length encoding can save valuable memory, reduce disk and network delays, and improve processor cache hit rates.

6.3.2 Memory Compression

Each worker has an assigned quota of memory. Whenever the worker memory is full, a *memory free cycle* is triggered to unload some DDOs from memory. However, it might be beneficial to compress the data of some DDOs instead. The memory will only be compressed if no other unneeded DDOs can be unloaded and

the only remaining choice is to unload DDOs that will most likely be loaded again in the near future. This information can be obtained by inspecting the tasks in the blocked set. The advantage/disadvantage of this technique is yet to be studied.

6.3.3 Code Generation

Currently, a query is executed by running successive tasks that transform and shuffle the data multiple times. Each node in the lineage DAG corresponds to a hardcoded function which is then wrapped in a *Task* object and submitted to the worker. Chains of narrow-dependency tasks often occur. These tasks are characterized by three features: (1) producing outputs that are consumed by exactly one task, (2) consuming inputs produced by exactly one task, and (3) the intermediate data does not need to be shuffled. In Apache Spark, code generation has been used to merge narrow-dependency tasks into larger, more complex, and more efficient tasks.[1]

Another advantage to using code generation is related to the Relation DDO in particular. The Relation DDO uses an *Expression* object to evaluate filter, join, and shuffle predicates. The Expression object is a nested tree of objects. Each node in the tree has a virtual *eval()* method. With each record, the *eval()* method is invoked on the current record pointer and a boolean value is retrieved. Virtual functions in C++ are essentially functions pointers. With complex enough expressions, the overhead of following these function pointers starts to dominate. The traversed path in the tree (which does not change at all during run-time) can be flattened to a few lines of dynamically generated code that perform the same data checks without the need to use function pointers.

6.3.4 Efficient Resource Utilization

The performance evaluation performed in this thesis only considered a fixed spatial allocation of resources. This is a valid assumption for most cluster environments. For example, YARN [37] allows applications to request spatial quotas of certain resources such as CPU cores, RAM megabytes, etc. However, the utilization of these allocated resources depends on the application. Further performance evaluations considering variable resource allocations are also needed.

Moreover, in order to utilize the allocated resources efficiently, the system must also consider optimal scheduling of concurrent jobs. The system is intended to support multiple concurrent users or even allow a single user to submit multiple concurrent jobs. Different query jobs may have different resource demands. An efficient scheduler should try to schedule tasks from more than one job to keep all allocated resources (e.g. CPU, disk, network, etc.) utilized.

[1] Project Tungsten. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`. Accessed January 29, 2021.

6.3.5 Straggler Mitigation

Straggler mitigation is a technique commonly used in distributed systems to reduce the makespan of jobs. Towards the end of a job straggler nodes may take longer than usual trying to finish the last few tasks. The common mitigation strategy is to submit backup tasks to idle workers and terminate the job as soon as one of the backup tasks finish.

6.3.6 Integration with Resource Managers

The current implementation of the system uses assigned resources on each node according to a configuration file. The system is intended to be a long-running server for serving raw data and/or executing analytical queries. The resources (CPU cores and memory) can, in theory, be resized while the worker is running. However, the mechanism to trigger the resize is not implemented. Integrating with a cluster resource manager such as YARN [37] or Mesos [21] can address this issue.

6.3.7 Machine Learning

The system is intended to serve as a storage component and an execution engine for a wide variety of analytical tasks. In the recent years, machine learning and deep learning have been extensively used to predict patterns and extract non-trivial information from raw data. Integrating with a machine learning framework can be valuable to the usability of the system. A suitable candidate is Tensorflow. The core of Tensorflow is written in C++ and can be integrated directly with the DDO.

6.3.8 Expanding DDO support to Other Programming Languages

Implementing a DDO requires implementing the interface in C++ and compiling a shared object library. However, the language requirement (in theory) can be relaxed using a special type of DDO with implementations of all its primitives that can look up any arbitrary block of code and an appropriate virtual machine implementation to execute it. Programming languages such as Python and Java are much more popular and easier to use than C++. Implementing a DDO in Python is possible if the Python interpreter is integrated with the system. The same can also be done for the Java virtual machine.

6.3.9 Semi-structured and Unstructured Data

The work done in this thesis only examined a workload dealing with structured data. An important optimization that is possible due to resource coupling is the efficient use of I/O and compute resources to retrieve and process portions of an indexed list of records. The applicability of this technique to semi-structured and unstructured data was not studied.

The system offers other design improvements over current state-of-the-art systems such as customizable block placement and co-location, which can be used to reduce the amount of shuffled data during query

execution. Adaptive query planning, task reordering, and asynchronous data pipelines are also features that can be leveraged in other workloads dealing with semi-structured and unstructured data.

References

[1] O. Adekoya, H. Sabiu, D. Eager, W. Grassmann, and D. Makaroff. A Case Study of Spark Resource Configuration and Management for Image Processing Applications. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 18–29, Markham, Canada, October 2018.

[2] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. NICE: Network-Integrated Cluster-Efficient Storage. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 29–40, Washington, DC, June 2017.

[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proceedings of the 6th European Conference on Computer Systems*, pages 287–300, Salzburg, Austria, April 2011.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM International Conference on Management of Data*, pages 1383–1394, Melbourne, Australia, May 2015.

[5] Atscale, Cloudera, and ODPi.org. Big Data & Analytics Maturity Survey Report. `https://www.atscale.com/wp-content/uploads/2020/02/2020-Big-Data-Analytics-Survey-Results.pdf`, 2020. Accessed December 10, 2020.

[6] M. Bakratsas, P. Basaras, D. Katsaros, and L. Tassiulas. Hadoop MapReduce Performance on SSDs for Analyzing Social Networks. *Big Data Research*, 11:1–10, March 2018.

[7] S. Behera, L. Wan, F. Mueller, M. Wolf, and S. Klasky. Orchestrating Fault Prediction with Live Migration and Checkpointing. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 167–171, Stockholm, Sweden, June 2020.

[8] A. Behrouzi-Far and E. Soljanin. Data Replication for Reducing Computing Time in Distributed Systems with Stragglers. In *Proceedings of the 2019 IEEE International Conference on Big Data*, pages 5986–5988, Los Angeles, CA, December 2019.

[9] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203, Ottawa, Canada, September 2000.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, June 2008.

[11] A. Cheptsov. HPC in Big Data Age: An Evaluation Report for Java-Based Data-Intensive Applications Implemented with Hadoop and OpenMPI. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 175–180, Kyoto, Japan, September 2014.

[12] H. E. Ciritoglu, J. Murphy, and C. Thorpe. HaRD: a heterogeneity-aware replica deletion for HDFS. *Journal of Big Data*, 6(1):1–21, October 2019.

[13] E. Costa, C. Costa, and M. Y. Santos. Evaluating partitioning and bucketing strategies for Hive-based Big Data Warehousing systems. *Journal of Big Data*, 6(34):1–38, May 2019.

[14] Databricks. Apache Spark Survey. `https://pages.databricks.com/rs/094-YMS-629/images/2016_Spark_Survey.pdf`, 2016. Accessed June 6, 2021.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, CA, December 2004.

[16] B. Dong, X. Zhong, Q. Zheng, L. Jian, J. Liu, J. Qiu, and Y. Li. Correlation Based File Prefetching Approach for Hadoop. In *Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science*, pages 41–48, Indianapolis, IN, November 2010.

[17] M. Drocco, V. G. Castellana, and M. Minutoli. Practical Distributed Programming in C++. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 35–39, Stockholm, Sweden, June 2020.

[18] M. Elteir, H. Lin, and W. Feng. Enhancing MapReduce via Asynchronous Data Processing. In *Proceedings of the IEEE 16th International Conference on Parallel and Distributed Systems*, pages 397–405, Shanghai, China, December 2010.

[19] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. *SIGOPS Operating Systems Review*, 37(5):29–43, October 2003.

[20] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, February 2009.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, Boston, MA, March 2011.

[22] X. Hua, H. Wu, and S. Ren. Enhancing Throughput of Hadoop Distributed File System for Interaction-Intensive Tasks. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 508–511, Turin, Italy, April 2014.

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, pages 59–72, Lisbon, Portugal, March 2007.

[24] P. Liu, A. Maruf, F. B. Yusuf, L. Jahan, H. Xu, B. Guan, L. Hu, and S. S. Iyengar. Towards Adaptive Replication for Hot/Cold Blocks in HDFS using MemCached. In *Proceedings of the 2nd International Conference on Data Intelligence and Security*, pages 188–194, South Padre Island, TX, June 2019.

[25] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. DataMPI: Extending MPI to Hadoop-Like Big Data Computing. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, pages 829–838, Phoenix, AZ, May 2014.

[26] G. Mackey, S. Sehrish, and J. Wang. Improving metadata management for small files in HDFS. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–4, New Orleans, LA, August 2009.

[27] H. Miller, P. Haller, N. Müller, and J. Boullier. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 82–97, Amsterdam, Netherlands, October 2016.

[28] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, 53:121–130, August 2015.

[29] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM International Conference on Management of Data*, pages 1357–1369, Melbourne, Australia, May 2015.

[30] J. Shafer, S. Rixner, and A. L. Cox. The Hadoop distributed filesystem: Balancing portability and performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 122–133, White Plains, NY, March 2010.

[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, Incline Village, NV, May 2010.

[32] S. Sur, M. J. Koop, and D. K. Panda. High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An in-Depth Performance Analysis. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 105–117, Tampa, FL, November 2006.

[33] A. S. Tanenbaum and M. van Steen. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the IEEE 26th International Conference on Data Engineering*, pages 996–1005, Long Beach, CA, March 2010.

[35] Transaction Processing Performance Council (TPC). TPC Benchmark H Standard Specification Version 2.18.0. `http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`, 2018. Accessed December 16, 2020.

[36] Transaction Processing Performance Council (TPC). TPC Benchmark DS Standard Specification Version 2.13.0. `http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.13.0.pdf`, 2020. Accessed April 15, 2021.

[37] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 1–16, Santa Clara, CA, October 2013.

[38] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, pages 1–17, Bordeaux, France, April 2015.

[39] C. Vorapongkitipun and N. Nupairoj. Improving performance of small-file accessing in Hadoop. In *Proceedings of the 11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 200–205, Chon Buri, Thailand, May 2014.

[40] W. Xu, W. Luo, and N. Woodward. Analysis and Optimization of Data Import with Hadoop. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1058–1066, Shanghai, China, May 2012.

[41] X. Yu and B. Hong. Grouping Blocks for MapReduce Co-Locality. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, pages 271–280, Hyderabad, India, May 2015.

[42] Y. Yu, W. Wang, J. Zhang, and K. Ben Letaief. LRC: Dependency-aware cache management for data analytics clusters. In *Proceedings of the 2017 IEEE Conference on Computer Communications*, pages 1–9, Atlanta, GA, May 2017.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 15–28, San Jose, CA, April 2012.

[44] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Communincations of the ACM*, 59(11):56–65, October 2016.

[45] H. Zhang, L. Wang, and H. Huang. SMARTH: Enabling Multi-pipeline Data Transfer in HDFS. In *Proceedings of the 43rd International Conference on Parallel Processing*, pages 30–39, Minneapolis, MN, September 2014.

Appendix A

TPC-H & TPC-DS Benchmark Results

Table A.1: TPC-DS benchmark results (scale factor = 10 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	193.32	116.85	0.60	98.72	0.51	117.59	0.61	126.17	0.65
Q 03	0.95	4.11	4.34	3.75	3.96	2.56	2.70	2.41	2.54
Q 07	28.52	7.21	0.25	4.61	0.16	5.62	0.20	3.21	0.11
Q 15	1.36	5.77	4.24	5.76	4.23	3.85	2.83	3.70	2.72
Q 19	2.70	6.36	2.36	5.25	1.95	4.30	1.59	3.27	1.21
Q 25	14.20	5.71	0.40	4.56	0.32	5.70	0.40	3.90	0.27
Q 26	18.99	5.57	0.29	7.19	0.38	3.74	0.20	3.28	0.17
Q 29	13.26	13.58	1.02	4.26	0.32	5.60	0.42	3.90	0.29
Q 41	0.12	1.31	10.99	1.41	11.82	1.31	10.99	1.29	10.85
Q 42	0.92	1.35	1.47	0.99	1.08	1.51	1.64	1.07	1.17
Q 43	3.93	1.56	0.40	1.02	0.26	1.81	0.46	1.07	0.27
Q 48	3.50	5.55	1.59	4.27	1.22	4.15	1.19	3.87	1.11
Q 50	9.93	4.45	0.45	8.03	0.81	8.98	0.90	4.26	0.43
Q 52	1.14	3.85	3.38	3.56	3.13	2.59	2.28	1.72	1.51
Q 55	1.14	4.47	3.91	3.32	2.90	2.62	2.29	1.74	1.53
Q 62	0.97	3.87	3.99	3.92	4.04	2.20	2.27	2.09	2.15
Q 91	25.01	1.92	0.08	2.14	0.09	1.41	0.06	2.03	0.08
Q 96	0.49	1.38	2.84	0.54	1.11	1.41	2.89	0.63	1.30
Q 99	6.10	6.75	1.11	3.43	0.56	3.27	0.54	1.78	0.29
Mean	**3.04**	**3.87**	**1.27**	**3.09**	**1.02**	**2.99**	**0.98**	**2.21**	**0.73**
Total	**133.20**	**84.78**	**0.64**	**68.00**	**0.51**	**62.62**	**0.47**	**45.22**	**0.34**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	170.36	0.88	170.36	0.88	284.28	1.47
Q 03	56.28	59.49	6.22	6.57	36.18	38.25
Q 07	43.97	1.54	9.95	0.35	28.58	1.00
Q 15	28.65	21.05	5.60	4.11	34.36	25.25
Q 19	17.57	6.52	5.57	2.07	30.62	11.36
Q 25	22.75	1.60	12.23	0.86	48.53	3.42
Q 26	14.50	0.76	5.37	0.28	30.34	1.60
Q 29	21.67	1.63	12.77	0.96	42.53	3.21
Q 41	UNS	N/A	1.76	14.76	UNS	N/A
Q 42	9.41	10.27	1.60	1.75	20.24	22.09
Q 43	9.36	2.38	1.47	0.37	24.27	6.18
Q 48	21.49	6.14	4.54	1.30	40.34	11.52
Q 50	16.22	1.63	7.55	0.76	50.34	5.07
Q 52	9.52	8.38	3.95	3.47	22.28	19.59
Q 55	19.37	16.95	4.91	4.30	22.21	19.43
Q 62	UNS	N/A	6.34	6.53	UNS	N/A
Q 91	9.46	0.38	2.50	0.10	16.86	0.67
Q 96	UNS	N/A	0.80	1.65	UNS	N/A
Q 99	UNS	N/A	5.01	0.82	UNS	N/A
Mean	**18.31**	**6.02**	**4.33**	**1.42**	**30.37**	**9.99**
Total	**N/A**	**N/A**	**98.13**	**0.74**	**N/A**	**N/A**

Error	**Meaning**
UNS	Unsupported or unrecognized feature

Table A.2: TPC-DS benchmark results (scale factor = 30 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	163.84	207.18	1.26	143.50	0.88	241.27	1.47	154.76	0.94
Q 03	0.94	4.19	4.47	2.93	3.13	3.84	4.10	2.83	3.02
Q 07	24.81	13.67	0.55	4.77	0.19	7.88	0.32	4.05	0.16
Q 15	1.30	7.56	5.82	6.12	4.71	6.62	5.10	4.35	3.35
Q 19	3.90	8.13	2.08	5.45	1.40	6.83	1.75	3.72	0.95
Q 25	15.88	13.22	0.83	8.97	0.56	37.76	2.38	9.29	0.59
Q 26	25.87	16.69	0.65	5.10	0.20	6.92	0.27	3.06	0.12
Q 29	17.83	12.77	0.72	9.07	0.51	13.24	0.74	8.57	0.48
Q 41	0.11	0.87	7.87	0.84	7.64	0.88	8.02	0.79	7.19
Q 42	0.96	3.40	3.55	1.47	1.54	3.24	3.38	1.65	1.72
Q 43	5.87	3.29	0.56	1.29	0.22	2.87	0.49	1.36	0.23
Q 48	5.29	7.82	1.48	4.29	0.81	9.29	1.76	4.47	0.84
Q 50	15.70	8.71	0.55	5.23	0.33	8.54	0.54	5.89	0.37
Q 52	1.32	5.39	4.09	3.42	2.60	4.21	3.20	2.31	1.75
Q 55	1.02	4.33	4.26	3.14	3.09	3.81	3.74	2.27	2.24
Q 62	8.28	5.25	0.63	3.65	0.44	3.48	0.42	2.00	0.24
Q 91	27.65	3.09	0.11	1.81	0.07	2.09	0.08	2.00	0.07
Q 96	0.59	2.73	4.60	0.66	1.11	2.82	4.75	0.67	1.12
Q 99	5.88	5.94	1.01	3.72	0.63	6.43	1.09	1.92	0.33
Mean	**3.92**	**5.70**	**1.45**	**3.20**	**0.82**	**5.15**	**1.31**	**2.70**	**0.69**
Total	**163.20**	**127.03**	**0.78**	**71.94**	**0.44**	**130.74**	**0.80**	**61.20**	**0.38**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	226.00	1.38	226.00	1.38	342.85	2.09
Q 03	47.90	51.12	4.56	4.87	42.20	45.04
Q 07	35.82	1.44	7.25	0.29	50.56	2.04
Q 15	51.63	39.75	5.84	4.49	40.37	31.07
Q 19	32.53	8.34	6.29	1.61	46.43	11.90
Q 25	65.28	4.11	21.54	1.36	80.54	5.07
Q 26	51.88	2.01	5.30	0.21	52.35	2.02
Q 29	67.57	3.79	21.22	1.19	70.49	3.95
Q 41	UNS	N/A	1.07	9.69	UNS	N/A
Q 42	20.49	21.36	2.41	2.51	30.26	31.55
Q 43	20.54	3.50	1.71	0.29	30.25	5.15
Q 48	16.48	3.11	4.78	0.90	60.35	11.40
Q 50	25.99	1.66	12.22	0.78	74.29	4.73
Q 52	19.90	15.11	5.05	3.84	30.28	22.99
Q 55	19.32	19.00	3.43	3.38	26.23	25.79
Q 62	UNS	N/A	4.00	0.48	UNS	N/A
Q 91	20.55	0.74	2.51	0.09	28.35	1.03
Q 96	UNS	N/A	1.19	2.00	UNS	N/A
Q 99	UNS	N/A	3.11	0.53	UNS	N/A
Mean	**31.48**	**8.03**	**4.46**	**1.14**	**44.23**	**11.29**
Total	**N/A**	**N/A**	**113.47**	**0.70**	**N/A**	**N/A**

Error	**Meaning**
UNS	Unsupported or unrecognized feature

Table A.3: TPC-DS benchmark results (scale factor = 100 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	516.56	735.61	1.42	490.25	0.95	729.06	1.41	485.51	0.94
Q 03	5.09	20.42	4.02	9.33	1.84	16.55	3.25	7.90	1.55
Q 07	56.41	59.02	1.05	11.59	0.21	49.35	0.87	8.40	0.15
Q 15	**2.43**	**37.96**	**15.65**	**11.98**	**4.94**	**35.38**	**14.58**	**11.71**	**4.83**
Q 19	**3.35**	**30.71**	**9.16**	**13.02**	**3.88**	**26.05**	**7.77**	**10.12**	**3.02**
Q 25	**14.29**	**61.65**	**4.32**	**36.57**	**2.56**	**97.27**	**6.81**	**43.82**	**3.07**
Q 26	33.40	35.34	1.06	10.34	0.31	34.52	1.03	8.04	0.24
Q 29	18.36	55.12	3.00	24.12	1.31	55.87	3.04	30.61	1.67
Q 41	0.62	3.34	5.35	2.70	4.32	2.35	3.76	1.77	2.83
Q 42	**1.38**	**11.87**	**8.58**	**3.62**	**2.62**	**11.62**	**8.40**	**3.58**	**2.59**
Q 43	16.68	16.81	1.01	3.60	0.22	31.43	1.88	3.66	0.22
Q 48	12.41	48.96	3.95	7.31	0.59	48.52	3.91	7.58	0.61
Q 50	12.64	39.70	3.14	16.40	1.30	38.22	3.02	20.46	1.62
Q 52	**1.32**	**15.88**	**12.04**	**6.09**	**4.62**	**45.61**	**34.58**	**5.02**	**3.81**
Q 55	**2.03**	**10.30**	**5.07**	**6.33**	**3.12**	**11.24**	**5.54**	**5.18**	**2.55**
Q 62	9.29	14.11	1.52	4.63	0.50	8.53	0.92	3.81	0.41
Q 91	26.28	5.41	0.21	3.19	0.12	4.73	0.18	3.16	0.12
Q 96	1.22	25.91	21.27	2.82	2.32	17.74	14.56	2.06	1.69
Q 99	15.99	20.07	1.26	5.95	0.37	15.86	0.99	5.43	0.34
Mean	**6.61**	**21.97**	**3.32**	**7.54**	**1.14**	**21.87**	**3.31**	**6.84**	**1.03**
Total	**233.18**	**512.59**	**2.20**	**179.60**	**0.77**	**550.86**	**2.36**	**182.31**	**0.78**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	490.67	0.95	490.67	0.95	633.94	1.23
Q 03	104.88	20.63	17.67	3.48	62.20	12.23
Q 07	92.08	1.63	15.84	0.28	90.59	1.61
Q 15	**61.65**	**25.41**	**20.02**	**8.25**	**64.42**	**26.56**
Q 19	**68.91**	**20.55**	**12.42**	**3.70**	**60.45**	**18.03**
Q 25	**138.49**	**9.69**	**100.42**	**7.03**	**120.53**	**8.44**
Q 26	48.95	1.47	10.75	0.32	60.35	1.81
Q 29	152.57	8.31	31.16	1.70	110.69	6.03
Q 41	UNS	N/A	4.90	7.86	UNS	N/A
Q 42	**55.59**	**40.17**	**5.11**	**3.69**	**48.28**	**34.89**
Q 43	67.30	4.03	3.91	0.23	46.27	2.77
Q 48	42.58	3.43	8.47	0.68	74.36	5.99
Q 50	140.07	11.08	31.81	2.52	118.33	9.36
Q 52	**75.46**	**57.21**	**9.04**	**6.86**	**44.25**	**33.55**
Q 55	**74.51**	**36.71**	**8.89**	**4.38**	**42.26**	**20.82**
Q 62	UNS	N/A	6.09	0.66	UNS	N/A
Q 91	24.76	0.94	3.19	0.12	42.32	1.61
Q 96	UNS	N/A	3.02	2.48	UNS	N/A
Q 99	UNS	N/A	6.30	0.39	UNS	N/A
Mean	**73.40**	**11.10**	**10.34**	**1.56**	**65.63**	**9.92**
Total	**N/A**	**N/A**	**299.02**	**1.28**	**N/A**	**N/A**

Error	**Meaning**
UNS	Unsupported or unrecognized feature

Table A.4: TPC-DS benchmark results (scale factor = 300 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	2129.25	1923.43	0.90	1210.81	0.57	1919.14	0.90	1192.09	0.56
Q 03	23.06	67.16	2.91	29.53	1.28	88.91	3.86	29.36	1.27
Q 07	**20.68**	271.76	13.14	38.45	1.86	206.00	9.96	40.18	1.94
Q 15	**5.03**	**89.23**	**17.73**	**40.44**	**8.04**	**83.32**	**16.56**	**55.77**	**11.08**
Q 19	**9.85**	**108.45**	**11.01**	**52.74**	**5.35**	**128.76**	**13.07**	**43.99**	**4.47**
Q 25	**33.64**	**366.95**	**10.91**	**226.30**	**6.73**	**482.26**	**14.34**	**304.26**	**9.04**
Q 26	**17.45**	**163.33**	**9.36**	**28.31**	**1.62**	**107.38**	**6.15**	**30.83**	**1.77**
Q 29	**48.29**	**357.32**	**7.40**	**192.49**	**3.99**	**432.39**	**8.95**	**275.69**	**5.71**
Q 41	0.29	4.78	16.44	3.78	12.99	3.07	10.56	1.77	6.10
Q 42	**2.85**	**61.24**	**21.52**	**20.09**	**7.06**	**64.74**	**22.75**	**23.43**	**8.23**
Q 43	48.52	88.53	1.82	16.58	0.34	87.63	1.81	47.75	0.98
Q 48	34.51	235.88	6.83	33.70	0.98	225.76	6.54	31.59	0.92
Q 50	**24.70**	**213.11**	**8.63**	**127.48**	**5.16**	**233.16**	**9.44**	**149.45**	**6.05**
Q 52	**2.80**	**61.27**	**21.90**	**30.00**	**10.72**	**120.00**	**42.89**	**27.49**	**9.82**
Q 55	**3.56**	**59.42**	**16.72**	**22.97**	**6.46**	**64.50**	**18.14**	**34.78**	**9.78**
Q 62	23.09	25.74	1.12	9.09	0.39	28.75	1.25	8.47	0.37
Q 91	28.49	17.74	0.62	9.07	0.32	16.10	0.57	13.61	0.48
Q 96	**3.57**	**65.61**	**18.37**	**14.97**	**4.19**	**65.01**	**18.20**	**18.25**	**5.11**
Q 99	47.39	49.15	1.04	15.49	0.33	68.82	1.45	12.79	0.27
Mean	**11.97**	**81.72**	**6.83**	**29.02**	**2.42**	**85.89**	**7.18**	**32.68**	**2.73**
Total	**377.76**	**2306.69**	**6.11**	**911.47**	**2.41**	**2506.55**	**6.64**	**1149.44**	**3.04**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	1193.71	0.56	1193.71	0.56	1265.48	0.59
Q 03	200.50	8.70	39.56	1.72	126.22	5.47
Q 07	**246.47**	**11.92**	**60.51**	**2.93**	**216.64**	**10.47**
Q 15	**138.91**	**27.61**	**53.97**	**10.73**	**114.41**	**22.74**
Q 19	**201.26**	**20.43**	**50.87**	**5.16**	**136.42**	**13.85**
Q 25	**401.85**	**11.94**	**231.39**	**6.88**	**350.56**	**10.42**
Q 26	**112.95**	**6.47**	**27.07**	**1.55**	**94.34**	**5.41**
Q 29	**359.02**	**7.43**	**232.68**	**4.82**	**318.48**	**6.59**
Q 41	UNS	N/A	5.88	20.19	UNS	N/A
Q 42	**149.94**	**52.69**	**26.31**	**9.24**	**108.27**	**38.04**
Q 43	130.40	2.69	13.74	0.28	106.27	2.19
Q 48	129.95	3.77	34.43	1.00	130.84	3.79
Q 50	**268.44**	**10.87**	**120.68**	**4.89**	**268.33**	**10.86**
Q 52	**167.08**	**59.72**	**28.14**	**10.06**	**102.29**	**36.56**
Q 55	**148.78**	**41.85**	**24.48**	**6.89**	**102.24**	**28.76**
Q 62	UNS	N/A	10.08	0.44	UNS	N/A
Q 91	28.78	1.01	6.04	0.21	76.31	2.68
Q 96	**UNS**	**N/A**	**14.77**	**4.13**	**UNS**	**N/A**
Q 99	UNS	N/A	20.95	0.44	UNS	N/A
Mean	**164.89**	**13.78**	**31.78**	**2.66**	**142.68**	**11.92**
Total	**N/A**	**N/A**	**1001.55**	**2.65**	**N/A**	**N/A**

Error	**Meaning**
UNS	Unsupported or unrecognized feature

Table A.5: TPC-H benchmark results (scale factor = 10 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	29.19	92.06	3.15	61.69	2.11	94.04	3.22	59.70	2.05
Q 01	1.47	6.41	4.38	4.76	3.25	6.27	4.28	3.79	2.59
Q 02	3.26	13.84	4.25	14.95	4.59	4.45	1.37	5.81	1.79
Q 03	4.99	9.56	1.91	9.91	1.99	7.29	1.46	6.25	1.25
Q 04	2.22	3.45	1.55	2.92	1.32	3.69	1.67	2.60	1.17
Q 05	11.53	6.42	0.56	5.72	0.50	15.25	1.32	11.61	1.01
Q 06	0.15	2.86	19.70	2.62	18.08	2.11	14.52	0.98	6.77
Q 07	20.10	7.54	0.38	9.93	0.49	7.60	0.38	4.49	0.22
Q 08	23.49	10.10	0.43	3.98	0.17	5.65	0.24	3.86	0.16
Q 09	19.32	31.16	1.61	7.21	0.37	12.26	0.63	5.58	0.29
Q 10	6.68	19.00	2.84	7.49	1.12	5.47	0.82	4.24	0.63
Q 11	6.11	1.87	0.31	1.65	0.27	1.49	0.24	1.52	0.25
Q 12	2.79	3.26	1.17	2.22	0.80	3.52	1.26	2.68	0.96
Q 13	7.01	5.07	0.72	3.57	0.51	3.88	0.55	3.78	0.54
Q 14	1.64	4.08	2.49	3.13	1.91	2.90	1.77	1.45	0.89
Q 15	2.42	6.86	2.84	5.98	2.48	5.81	2.41	3.03	1.25
Q 16	3.58	6.35	1.78	14.48	4.05	3.17	0.88	3.77	1.05
Q 17	7.54	5.07	0.67	3.43	0.46	8.27	1.10	3.41	0.45
Q 18	4.23	8.83	2.09	7.24	1.71	6.79	1.60	5.96	1.41
Q 19	2.73	5.62	2.06	1.90	0.70	5.53	2.03	2.30	0.84
Q 20	2.45	8.25	3.37	9.59	3.92	4.19	1.71	3.25	1.33
Q 21	11.87	11.79	0.99	10.42	0.88	9.67	0.81	7.65	0.64
Q 22	2.78	2.02	0.73	2.64	0.95	1.40	0.50	1.94	0.70
Mean	**4.24**	**6.45**	**1.52**	**5.04**	**1.19**	**4.87**	**1.15**	**3.51**	**0.83**
Total	**148.32**	**179.39**	**1.21**	**135.73**	**0.92**	**126.65**	**0.85**	**89.97**	**0.61**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	130.10	4.46	130.10	4.46	154.88	5.31
Q 01	44.37	30.27	7.90	5.39	17.88	12.19
Q 02	CART	N/A	14.02	4.31	CART	N/A
Q 03	29.99	6.01	16.33	3.27	46.54	9.32
Q 04	12.49	5.63	6.12	2.76	38.29	17.26
Q 05	28.12	2.44	9.36	0.81	54.53	4.73
Q 06	5.33	36.74	3.03	20.90	6.21	42.83
Q 07	38.08	1.89	15.64	0.78	186.48	9.28
Q 08	CART	N/A	13.99	0.60	CART	N/A
Q 09	CART	N/A	28.67	1.48	CART	N/A
Q 10	27.90	4.17	12.51	1.87	42.36	6.34
Q 11	UNS	N/A	2.62	0.43	UNS	N/A
Q 12	10.40	3.73	3.09	1.11	32.18	11.55
Q 13	UNS	N/A	8.23	1.18	UNS	N/A
Q 14	7.35	4.49	3.79	2.31	18.23	11.14
Q 15	UNS	N/A	6.53	2.70	UNS	N/A
Q 16	CART	N/A	10.98	3.07	CART	N/A
Q 17	57.69	7.66	6.29	0.84	70.28	9.33
Q 18	42.72	10.09	15.21	3.59	68.36	16.15
Q 19	9.41	3.45	3.21	1.18	12.23	4.48
Q 20	25.71	10.50	14.32	5.85	46.53	19.01
Q 21	UNS	N/A	24.92	2.10	UNS	N/A
Q 22	UNS	N/A	2.71	0.98	UNS	N/A
Mean	**20.60**	**4.86**	**8.22**	**1.94**	**35.39**	**8.34**
Total	**N/A**	**N/A**	**229.48**	**1.55**	**N/A**	**N/A**

Error	**Meaning**
CART	Query planner decided query needs cartesian product
UNS	Unsupported or unrecognized feature

Table A.6: TPC-H benchmark results (scale factor = 30 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	101.90	221.61	2.17	155.62	1.53	220.80	2.17	144.03	1.41
Q 01	3.65	15.64	4.28	8.62	2.36	17.54	4.80	6.32	1.73
Q 02	3.30	14.68	4.44	13.65	4.13	6.21	1.88	6.27	1.90
Q 03	9.22	18.63	2.02	16.02	1.74	15.33	1.66	28.79	3.12
Q 04	1.82	9.43	5.18	5.48	3.01	8.47	4.65	5.19	2.85
Q 05	12.55	27.32	2.18	35.17	2.80	16.68	1.33	12.37	0.99
Q 06	0.23	7.96	35.37	5.89	26.16	5.80	25.79	3.26	14.50
Q 07	33.85	22.13	0.65	16.74	0.49	14.92	0.44	11.11	0.33
Q 08	30.25	51.80	1.71	34.17	1.13	15.01	0.50	10.41	0.34
Q 09	23.90	21.57	0.90	17.93	0.75	19.66	0.82	12.15	0.51
Q 10	8.81	15.23	1.73	10.91	1.24	12.68	1.44	10.36	1.18
Q 11	9.55	7.81	0.82	2.78	0.29	3.10	0.32	2.38	0.25
Q 12	3.83	8.62	2.25	3.65	0.95	9.08	2.37	4.91	1.28
Q 13	19.14	6.86	0.36	5.98	0.31	6.84	0.36	5.44	0.28
Q 14	1.65	7.42	4.51	6.41	3.89	6.53	3.96	3.57	2.17
Q 15	2.90	14.36	4.96	12.94	4.47	12.12	4.18	6.83	2.36
Q 16	4.54	18.33	4.04	18.90	4.17	6.50	1.43	6.83	1.51
Q 17	19.60	20.45	1.04	12.40	0.63	15.71	0.80	15.50	0.79
Q 18	6.49	39.89	6.14	35.87	5.52	15.30	2.36	12.90	1.99
Q 19	4.37	11.00	2.52	3.50	0.80	11.80	2.70	4.85	1.11
Q 20	5.30	14.21	2.68	13.64	2.57	8.35	1.57	5.66	1.07
Q 21	20.46	27.51	1.34	17.23	0.84	21.56	1.05	18.17	0.89
Q 22	3.41	4.54	1.33	2.45	0.72	4.30	1.26	2.79	0.82
Mean	**6.30**	**14.69**	**2.33**	**10.33**	**1.64**	**10.23**	**1.62**	**7.31**	**1.16**
Total	**228.80**	**385.36**	**1.68**	**300.34**	**1.31**	**253.49**	**1.11**	**196.06**	**0.86**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	207.68	2.04	207.68	2.04	250.84	2.46
Q 01	56.01	15.33	11.62	3.18	21.91	6.00
Q 02	CART	N/A	17.13	5.19	CART	N/A
Q 03	35.95	3.90	27.78	3.01	60.65	6.58
Q 04	17.51	9.61	10.23	5.62	66.33	36.43
Q 05	61.26	4.88	24.13	1.92	84.58	6.74
Q 06	7.33	32.56	6.69	29.74	6.70	29.77
Q 07	76.18	2.25	20.01	0.59	258.52	7.64
Q 08	CART	N/A	24.55	0.81	CART	N/A
Q 09	CART	N/A	40.22	1.68	CART	N/A
Q 10	51.80	5.88	20.79	2.36	72.81	8.27
Q 11	UNS	N/A	4.04	0.42	UNS	N/A
Q 12	15.40	4.02	7.93	2.07	42.24	11.03
Q 13	UNS	N/A	19.71	1.03	UNS	N/A
Q 14	9.37	5.69	7.26	4.41	18.21	11.06
Q 15	UNS	N/A	13.30	4.59	UNS	N/A
Q 16	CART	N/A	12.76	2.81	CART	N/A
Q 17	91.81	4.69	21.18	1.08	108.25	5.52
Q 18	66.82	10.29	39.35	6.06	102.52	15.79
Q 19	9.35	2.14	6.28	1.44	18.24	4.18
Q 20	32.72	6.17	17.56	3.31	102.40	19.31
Q 21	UNS	N/A	24.32	1.19	UNS	N/A
Q 22	UNS	N/A	4.74	1.39	UNS	N/A
Mean	**30.13**	**4.78**	**14.41**	**2.29**	**50.79**	**8.06**
Total	**N/A**	**N/A**	**381.59**	**1.67**	**N/A**	**N/A**

Error	**Meaning**
CART	Query planner decided query needs cartesian product
UNS	Unsupported or unrecognized feature

Table A.7: TPC-H benchmark results (scale factor = 100 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	474.59	678.66	1.43	491.18	1.03	662.79	1.40	414.05	0.87
Q 01	16.99	62.06	3.65	25.17	1.48	49.24	2.90	21.79	1.28
Q 02	**3.57**	**19.77**	**5.55**	**20.43**	**5.73**	**24.91**	**6.99**	**13.15**	**3.69**
Q 03	30.35	131.70	4.34	105.42	3.47	51.43	1.69	33.67	1.11
Q 04	11.79	32.74	2.78	18.31	1.55	35.28	2.99	17.78	1.51
Q 05	23.93	75.61	3.16	44.41	1.86	78.65	3.29	98.53	4.12
Q 06	**0.59**	**27.06**	**45.56**	**19.69**	**33.14**	**23.40**	**39.39**	**8.41**	**14.15**
Q 07	53.02	62.22	1.17	70.74	1.33	94.05	1.77	35.14	0.66
Q 08	52.26	126.32	2.42	65.69	1.26	65.81	1.26	36.18	0.69
Q 09	61.80	200.45	3.24	184.43	2.98	131.64	2.13	104.80	1.70
Q 10	16.74	58.43	3.49	33.70	2.01	50.60	3.02	27.05	1.62
Q 11	18.69	23.98	1.28	13.13	0.70	13.98	0.75	29.35	1.57
Q 12	14.62	52.24	3.57	12.62	0.86	44.17	3.02	13.92	0.95
Q 13	58.99	38.84	0.66	24.72	0.42	24.51	0.42	14.79	0.25
Q 14	**4.56**	**39.38**	**8.64**	**23.21**	**5.09**	**35.71**	**7.83**	**18.47**	**4.05**
Q 15	**6.61**	**63.65**	**9.63**	**36.53**	**5.53**	**50.84**	**7.69**	**18.37**	**2.78**
Q 16	**9.17**	**51.57**	**5.62**	**61.81**	**6.74**	**41.26**	**4.50**	**38.31**	**4.18**
Q 17	61.71	89.26	1.45	55.55	0.90	75.82	1.23	81.48	1.32
Q 18	**17.28**	**137.23**	**7.94**	**63.31**	**3.66**	**71.07**	**4.11**	**40.89**	**2.37**
Q 19	30.63	67.51	2.20	17.91	0.58	59.04	1.93	16.44	0.54
Q 20	11.77	59.36	5.04	40.36	3.43	37.04	3.15	20.95	1.78
Q 21	61.61	180.76	2.93	94.24	1.53	167.50	2.72	126.77	2.06
Q 22	7.66	10.75	1.40	8.84	1.15	10.15	1.33	9.17	1.20
Mean	**16.62**	**57.89**	**3.48**	**35.32**	**2.13**	**46.01**	**2.77**	**27.69**	**1.67**
Total	**574.30**	**1610.92**	**2.81**	**1040.25**	**1.81**	**1236.10**	**2.15**	**825.42**	**1.44**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	494.49	1.04	494.49	1.04	531.62	1.12
Q 01	97.35	5.73	25.41	1.50	33.89	2.00
Q 02	**CART**	**N/A**	**32.81**	**9.20**	**CART**	**N/A**
Q 03	106.55	3.51	107.89	3.56	108.78	3.58
Q 04	61.73	5.24	19.94	1.69	92.29	7.83
Q 05	160.72	6.72	90.39	3.78	148.61	6.21
Q 06	**16.35**	**27.53**	**17.01**	**28.63**	**10.20**	**17.17**
Q 07	343.39	6.48	88.12	1.66	352.50	6.65
Q 08	CART	N/A	96.50	1.85	CART	N/A
Q 09	CART	N/A	134.19	2.17	CART	N/A
Q 10	114.00	6.81	46.80	2.80	152.40	9.10
Q 11	UNS	N/A	28.14	1.51	UNS	N/A
Q 12	44.50	3.04	25.88	1.77	62.21	4.26
Q 13	UNS	N/A	23.54	0.40	UNS	N/A
Q 14	**25.40**	**5.57**	**28.26**	**6.20**	**38.22**	**8.38**
Q 15	**UNS**	**N/A**	**36.85**	**5.58**	**UNS**	**N/A**
Q 16	**CART**	**N/A**	**53.41**	**5.82**	**CART**	**N/A**
Q 17	326.63	5.29	89.31	1.45	250.28	4.06
Q 18	**209.37**	**12.12**	**135.65**	**7.85**	**212.38**	**12.29**
Q 19	32.44	1.06	28.03	0.92	26.24	0.86
Q 20	109.04	9.26	41.16	3.50	134.55	11.43
Q 21	UNS	N/A	127.20	2.06	UNS	N/A
Q 22	UNS	N/A	19.00	2.48	UNS	N/A
Mean	**87.64**	**5.27**	**46.46**	**2.80**	**85.14**	**5.12**
Total	**N/A**	**N/A**	**1295.51**	**2.26**	**N/A**	**N/A**

Error	Meaning
CART	Query planner decided query needs cartesian product
UNS	Unsupported or unrecognized feature

Table A.8: TPC-H benchmark results (scale factor = 300 GB). Query times are approximated to the nearest $^1/_{100}{}^{\text{th}}$ of a second. Ingest time is not included in the mean or total.

	DDO	SparkSQL 2		SparkSQL 2 (compressed)		SparkSQL 3		SparkSQL 3 (compressed)	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	1874.49	1892.59	1.01	1396.92	0.75	1940.28	1.04	1261.71	0.67
Q 01	60.46	147.27	2.44	63.32	1.05	136.29	2.25	61.87	1.02
Q 02	**10.22**	**83.01**	**8.13**	**34.24**	**3.35**	**62.59**	**6.13**	**31.08**	**3.04**
Q 03	116.76	338.76	2.90	163.65	1.40	301.69	2.58	153.84	1.32
Q 04	29.11	135.02	4.64	69.96	2.40	118.70	4.08	70.07	2.41
Q 05	**54.07**	**519.39**	**9.61**	**344.13**	**6.36**	**434.27**	**8.03**	**344.36**	**6.37**
Q 06	**1.63**	**112.24**	**68.82**	**64.16**	**39.34**	**102.23**	**62.68**	**51.70**	**31.70**
Q 07	116.75	559.00	4.79	381.95	3.27	324.67	2.78	280.46	2.40
Q 08	187.61	630.55	3.36	526.88	2.81	326.02	1.74	243.18	1.30
Q 09	174.21	919.60	0.00	912.02	0.00	697.89	0.00	491.97	0.00
Q 10	60.02	333.39	5.55	239.22	3.99	265.41	4.42	135.03	2.25
Q 11	35.68	54.54	1.53	85.58	2.40	57.65	1.62	51.89	1.45
Q 12	34.94	241.25	6.91	81.53	2.33	240.17	6.87	75.75	2.17
Q 13	141.03	88.58	0.63	72.81	0.52	145.14	1.03	47.22	0.33
Q 14	**11.06**	**147.82**	**13.37**	**99.63**	**9.01**	**180.45**	**16.32**	**84.44**	**7.64**
Q 15	**15.81**	**286.18**	**18.10**	**159.46**	**10.09**	**296.90**	**18.78**	**154.80**	**9.79**
Q 16	**23.97**	**382.38**	**15.95**	**449.41**	**18.75**	**282.46**	**11.79**	**273.07**	**11.39**
Q 17	**132.33**	**667.30**	**5.04**	**477.32**	**3.61**	**890.39**	**6.73**	**421.50**	**3.19**
Q 18	**47.46**	**670.49**	**14.13**	**472.24**	**9.95**	**584.94**	**12.32**	**645.03**	**13.59**
Q 19	68.11	251.99	3.70	79.52	1.17	237.20	3.48	95.59	1.40
Q 20	**36.10**	**339.72**	**9.41**	**182.98**	**5.07**	**199.98**	**5.54**	**114.86**	**3.18**
Q 21	174.88	1138.86	6.51	662.14	3.79	866.68	4.96	497.21	2.84
Q 22	18.03	53.17	2.95	34.09	1.89	42.03	2.33	35.93	1.99
Mean	**44.12**	**261.59**	**5.93**	**165.44**	**3.75**	**227.70**	**5.16**	**133.65**	**3.03**
Total	**1550.23**	**8100.52**	**5.23**	**5656.24**	**3.65**	**6793.76**	**4.38**	**4360.86**	**2.81**

	Hive on Spark 2		SparkSQL 2 on Hive		Hive on Tez	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
Ingest	1329.94	0.71	1329.94	0.71	1356.44	0.72
Q 01	194.42	3.22	59.73	0.99	66.24	1.10
Q 02	**CART**	**N/A**	**50.64**	**4.96**	**CART**	**N/A**
Q 03	327.23	2.80	176.03	1.51	216.65	1.86
Q 04	236.34	8.12	76.62	2.63	178.33	6.13
Q 05	**459.15**	**8.49**	**412.00**	**7.62**	**396.61**	**7.33**
Q 06	**53.41**	**32.75**	**70.73**	**43.36**	**36.20**	**22.19**
Q 07	1048.51	8.98	392.50	3.36	746.53	6.39
Q 08	CART	N/A	459.93	2.45	CART	N/A
Q 09	CART	N/A	863.67	0.00	CART	N/A
Q 10	356.95	5.95	243.55	4.06	294.23	4.90
Q 11	UNS	N/A	53.24	1.49	UNS	N/A
Q 12	156.81	4.49	83.12	2.38	116.22	3.33
Q 13	UNS	N/A	78.25	0.55	UNS	N/A
Q 14	**101.60**	**9.19**	**96.42**	**8.72**	**80.20**	**7.25**
Q 15	**UNS**	**N/A**	**160.69**	**10.16**	**UNS**	**N/A**
Q 16	**CART**	**N/A**	**285.06**	**11.89**	**CART**	**N/A**
Q 17	**987.87**	**7.47**	**537.07**	**4.06**	**748.29**	**5.65**
Q 18	**655.04**	**13.80**	**826.15**	**17.41**	**514.32**	**10.84**
Q 19	97.60	1.43	167.82	2.46	74.30	1.09
Q 20	**467.78**	**12.96**	**159.85**	**4.43**	**258.41**	**7.16**
Q 21	UNS	N/A	669.09	3.83	UNS	N/A
Q 22	UNS	N/A	44.10	2.45	UNS	N/A
Mean	**279.98**	**6.35**	**175.69**	**3.98**	**194.12**	**4.40**
Total	**N/A**	**N/A**	**5966.24**	**3.85**	**N/A**	**N/A**

Error	**Meaning**
CART	Query planner decided query needs cartesian product
UNS	Unsupported or unrecognized feature

Appendix B

Relational Query Planning

This appendix shows a few examples complementary to Section 4.6. Each example features an SQL query from the benchmark suites, its corresponding query plan as a lineage DAG, and the actual hardcoded query plan. Irrelevant and/or minor details in the code were omitted for clarity.

B.1 Inner Queries

SQL queries may have inner queries which can be *correlated* or *uncorrelated* on a record level. Uncorrelated inner queries can be computed separately at the beginning of the query or computed after some result is available during execution. This type of inner query can be injected into the lineage DAG trivially. For correlated inner queries, however, this is not possible. Correlated inner queries reference dynamically changing values from the outer query. To support this, a special type of join involving expressions on computed values needs to be implemented. Consider the following SQL query:

```
1  -- TPC-H Potential Part Promotion Query (Q20)
2
3  SELECT
4      s_name,
5      s_address
6  FROM
7      supplier,
8      nation
9  WHERE
10     s_suppkey IN (
11         SELECT
12             ps_suppkey
13         FROM
14             partsupp
15         WHERE
16             ps_partkey IN (
17                 SELECT
18                     p_partkey
19                 FROM
20                     part
21                 WHERE
22                     p_name like 'forest%'
23             )
24             AND ps_availqty > (
25                 SELECT
26                     0.5 * SUM(l_quantity)
27                 FROM
28                     lineitem
29                 WHERE
30                     l_partkey = ps_partkey
31                     AND l_suppkey = ps_suppkey
32                     AND l_shipdate >= date '1994-01-01'
33                     AND l_shipdate < date '1994-01-01' + interval '1' year
34             )
35     )
36     AND s_nationkey = n_nationkey
37     AND n_name = 'CANADA'
38 ORDER BY
39     s_name;
```

The first inner query selecting supplier keys (*s_suppkey*) can be converted to a *LeftSemi* join. *LeftSemi* joins are a special type of join implemented in this system (inspired by SparkSQL) to filter out records from

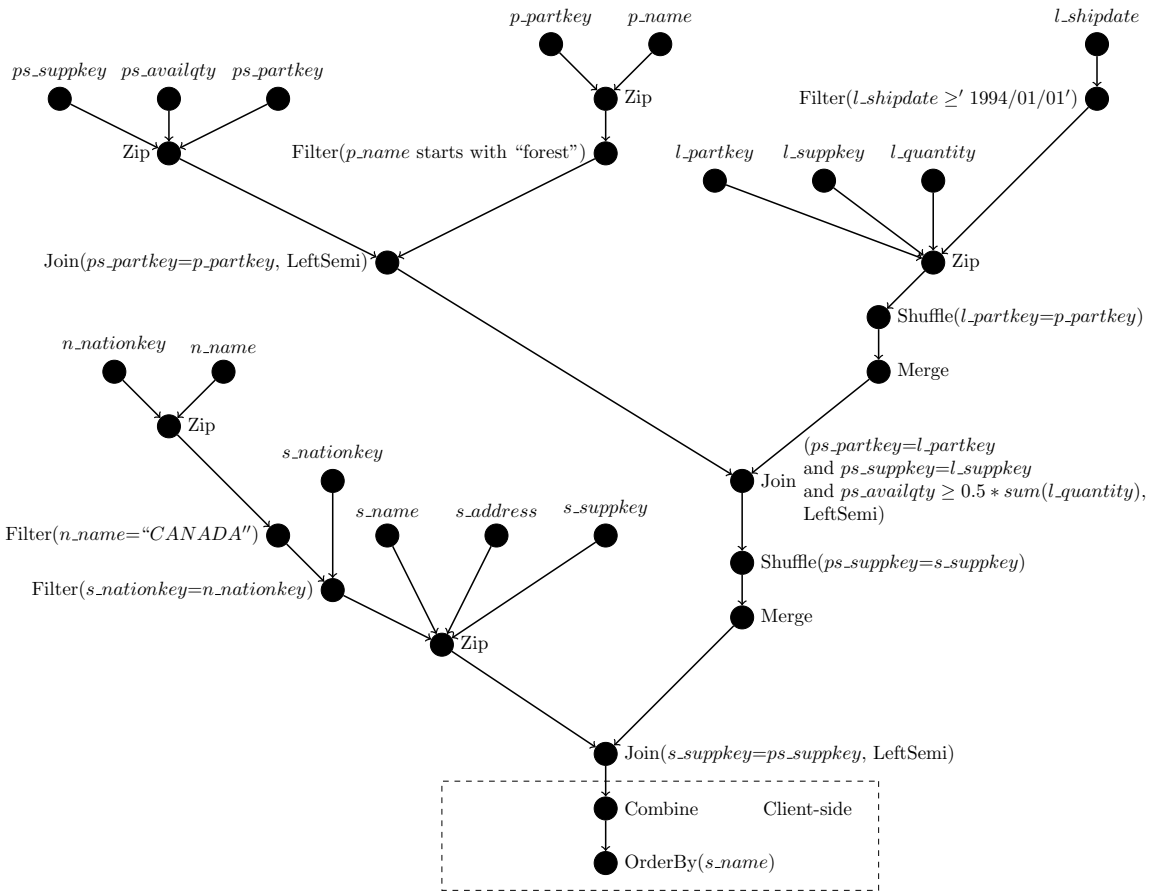


Figure B.1: Lineage DAG showing inner correlated and un-correlated queries. Both uncorrelated "IN" clauses are evaluated using *LeftSemi* joins. The correlated inner query is evaluated using a join expression referencing aggregated values. Note the *LeftSemi* join near the upper left corner. No shuffle was required due to record distribution and DDO part co-location.

the left side that do not match any records from the right side. Ambiguity between left and right sides in the lineage DAG is resolved using the expression; i.e. the join expression must be in the form $left = right$. *LeftSemi* joins are used to evaluate both "IN" clauses as shown in Figure B.1. The final inner query is correlated and filters suppliers based on the part-supplier relation. If the quantity of one part from one supplier comprises more than half the total quantity, the supplier is considered. To evaluate this, a join-and-filter operation needs to be implemented. Here, the filter contains a dynamically aggregated value from the right side. The following listing shows the corresponding query plan in code:

```
1  auto lineitem = Universe::instance()
2      ->getNamespace("tpch")->as<Schema>()
3      ->getTable("lineitem")
4      .replicas()
5      .indexedBy("l_shipdate");
6
7  auto part = Universe::instance()
8      ->getNamespace("tpch")->as<Schema>()
9      ->getTable("part");
10
11 auto partsupp = Universe::instance()
12     ->getNamespace("tpch")->as<Schema>()
13     ->getTable("partsupp");
14
```

```
auto supplier = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("supplier")
    .replicas()
    .indexedBy("s_nationkey");

auto nation = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("nation");

Lineage queryPlan = {
    Lineage(supplier["s_nationkey"])
    + Filter(c(supplier["s_nationkey"]) == computedLiteral<uint8>())(
        Zip()({
            nation["n_nationkey"],
            nation["n_name"]
        })
        + Filter(c(nation["n_name"]) == l<String>("CANADA"))
    )
    + Zip()({
        supplier["s_name"],
        supplier["s_address"],
        supplier["s_suppkey"]
    })
    + Join(
        c(supplier["s_suppkey"]) == c(partsupp["ps_suppkey"]),
        JoinType::LeftSemi
    )(
        Zip()({
            partsupp["ps_suppkey"],
            partsupp["ps_availqty"],
            partsupp["ps_partkey"]
        })
        + Join(
            c(partsupp["ps_partkey"]) == c(part["p_partkey"]),
            JoinType::LeftSemi
        )(
            Zip()({
                part["p_partkey"],
                part["p_name"]
            })
            + Filter(c(part["p_name"]).startsWith("forest"))
        )
        + Join(
            c(partsupp["ps_partkey"]) == c(lineitem["l_partkey"]),
            c(partsupp["ps_suppkey"]) == c(lineitem["l_suppkey"])
            && c(partsupp["ps_availqty"]) >= "0.5*sum(l_quantity)",
            JoinType::LeftSemi
        )(
            Lineage(lineitem["l_shipdate"])
            + Filter(
                c(lineitem["l_shipdate"]) >= date("1994-01-01")
                && c(lineitem["l_shipdate"]) < date("1995-01-01")
            )
            + Zip()({
                lineitem["l_partkey"],
                lineitem["l_suppkey"],
                lineitem["l_quantity"]
            })
            + Shuffle(c(lineitem["l_partkey"]) == c(part["p_partkey"]))
```

```
            + Merge()
        )
        + Shuffle(c(partsupp["ps_suppkey"] == c(supplier["s_suppkey"])))
        + Merge()
    )
};
```

B.2 Result Reuse

SQL queries may sometimes need to utilize some previously-computed result. Compared to the lineage DAG in Figure B.1 which has a tree structure with no result reuse, the query in the following listing performs multiple operations on the table *lineitem*:

```
-- TPC-H Suppliers Who Kept Orders Waiting Query (Q21)

SELECT
    s_name,
    COUNT(*) AS numwait
FROM
    supplier,
    lineitem l1,
    orders,
    nation
WHERE
    s_suppkey = l1.l_suppkey
    AND o_orderkey = l1.l_orderkey
    AND o_orderstatus = 'F'
    AND l1.l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT
            *
        FROM
            lineitem l2
        WHERE
            l2.l_orderkey = l1.l_orderkey
            AND l2.l_suppkey <> l1.l_suppkey
    )
    AND NOT EXISTS (
        SELECT
            *
        FROM
            lineitem l3
        WHERE
            l3.l_orderkey = l1.l_orderkey
            AND l3.l_suppkey <> l1.l_suppkey
            AND l3.l_receiptdate > l3.l_commitdate
    )
    AND s_nationkey = n_nationkey
    AND n_name = 'CANADA'
GROUP BY
    s_name
ORDER BY
    numwait DESC,
    s_name;
```

The "EXISTS" clause is evaluated using a *LeftSemi* join, while the "NOT EXISTS" clause is evaluated using a *LeftAnti* join. A *LeftAnti* join will produce records from the left side that do not match any records from the right side. The LeftAnti join (NOT EXISTS) uses the same *lineitem* table that was filtered on $l_receiptdate > l_commitdate$ as shown in Figure B.2. The following listing shows the corresponding query plan in code:

```
auto supplier = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
```

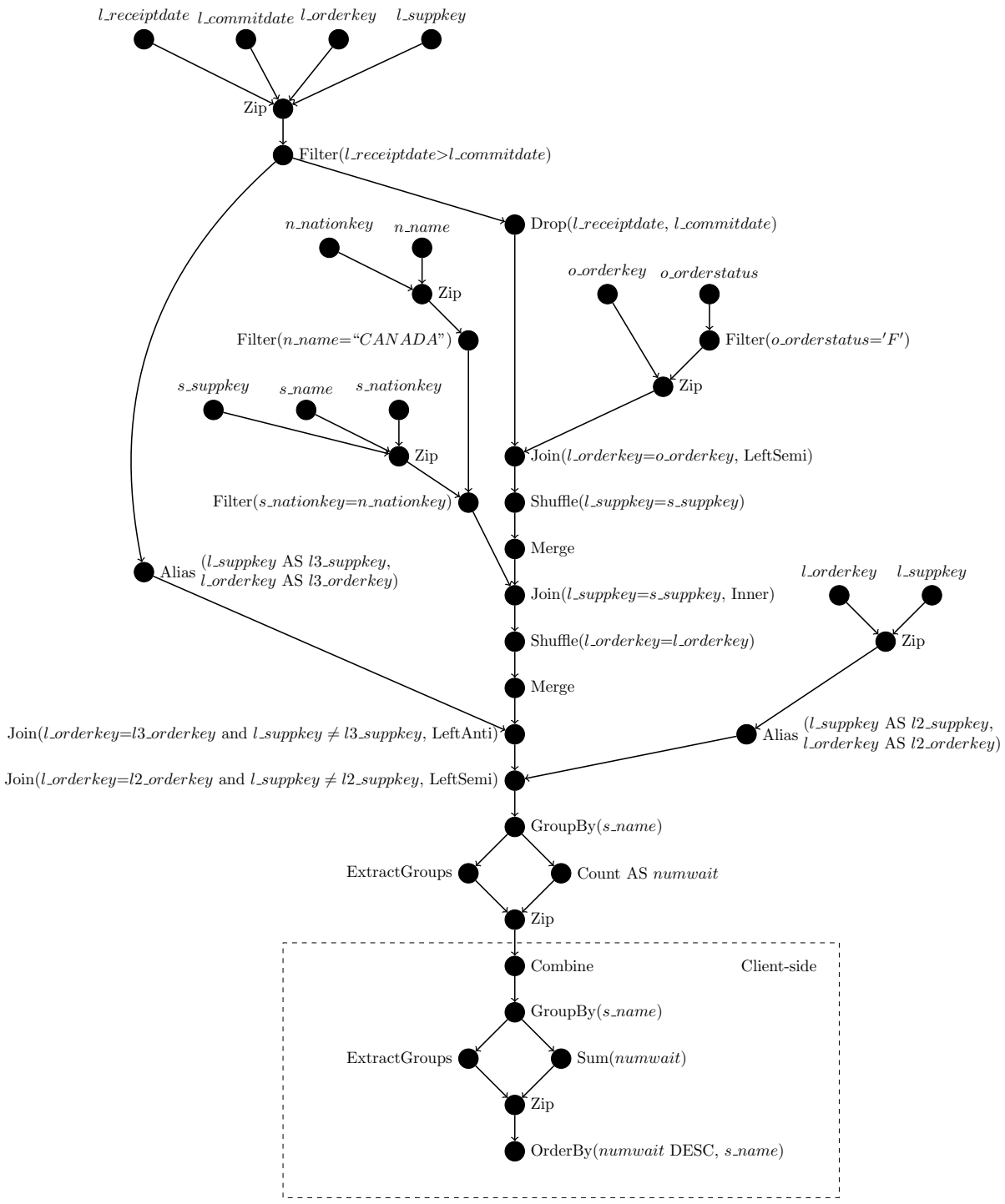


Figure B.2: lineage DAG showing aliasing and reuse of previously computed DDO parts. The "EXISTS" and "NOT EXISTS" clauses are evaluated using a *LeftSemi* join and a *LeftAnti* join, respectively. The filtered "lineitem" table is aliased and input to the *LeftAnti* join node. Aliases are used to prevent ambiguity in expressions.

```
    ->getTable("supplier");

auto nation = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("nation");

auto lineitem = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("lineitem");

auto orders = Universe::instance()
    ->getNamespace("tpch")->as<Schema>()
    ->getTable("orders")
    .replicas()
    .indexedBy("o_orderstatus");

Lineage l1 = {
    Zip()({
        lineitem["l_orderkey"],
        lineitem["l_suppkey"]
    })
};

Lineage l1_filter = {
    l1
    + Zip()({
        lineitem["l_receiptdate"],
        lineitem["l_commitdate"]
    })
    + Filter(c(lineitem["l_receiptdate"]) > c(lineitem["l_commitdate"]))
    + Drop({ "l_receiptdate", "l_commitdate" })
};

Lineage rel = {
    l1_filter
    + Join(c(lineitem["l_orderkey"]) == c(orders["o_orderkey"]), JoinType::LeftSemi)(
        Lineage(orders["o_orderstatus"])
        + Filter(c(orders["o_orderstatus"]) == l<char>('F'))
        + Zip()(
            orders["o_orderkey"]
        )
    )
    + Shuffle(c(lineitem["l_suppkey"]) == c(supplier["s_suppkey"]))
    + Merge()
    + Join(
        c(lineitem["l_suppkey"]) == c(supplier["s_suppkey"]),
        JoinType::Inner
    )(
        Zip()({
            supplier["s_suppkey"],
            supplier["s_name"],
            supplier["s_nationkey"]
        })
        + Filter(c(supplier["s_nationkey"]) == computedLiteral<uint8>())(
            Zip()({
                nation["n_nationkey"],
                nation["n_name"]
            })
            + Filter(c(nation["n_name"]) == l<String>("CANADA"))
        )
    )
```

```
    + Shuffle(c(lineitem["l_orderkey"]) == c(lineitem["l_orderkey"]))
    + Merge()
    + Join(
        c(lineitem["l_orderkey"]) == c("l3_orderkey")
        && c(lineitem["l_suppkey"]) != c("l3_suppkey"),
        JoinType::LeftAnti
    )(
        l1_filter
        + Alias(lineitem["l_suppkey"], "l3_suppkey")
        + Alias(lineitem["l_orderkey"], "l3_orderkey")
    )
    + Join(
        c(lineitem["l_orderkey"]) == c("l2_orderkey")
        && c(lineitem["l_suppkey"]) != c("l2_suppkey"),
        JoinType::LeftSemi
    )(
        l1
        + Alias(lineitem["l_suppkey"], "l2_suppkey")
        + Alias(lineitem["l_orderkey"], "l2_orderkey")
    )
    + GroupBy({ "s_name" })
};

Lineage queryPlan = {
    Zip()({
        rel + ExtractGroups(),
        rel + Count()
    })
};
```