UNIVERSITY OF TROMSØ UIT

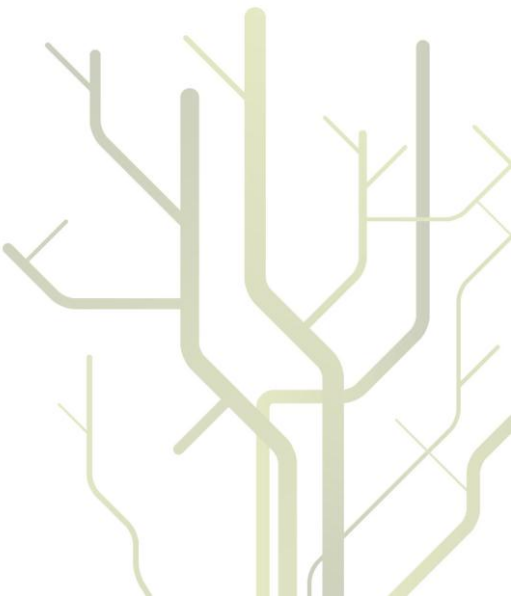
FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Cogset: A High-Performance MapReduce Engine

Steffen Viken Valvåg

A dissertation for the degree of
Philosophiae Doctor

September 2011



Abstract

MapReduce has become a widely employed programming model for large-scale data-intensive computations. Traditional MapReduce engines employ *dynamic routing* of data as a core mechanism for fault tolerance and load balancing. An alternative mechanism is *static routing*, which reduces the need to store temporary copies of intermediate data, but requires a tighter coupling between the components for storage and processing. The initial intuition motivating our work is that reading and writing less temporary data could improve performance, while the tight coupling of storage and processing could be leveraged to improve data locality.

We therefore conjecture that a high-performance MapReduce engine can be based on static routing, while preserving the non-functional properties associated with traditional engines. To investigate this thesis, we design, implement, and experiment with *Cogset*, a distributed MapReduce engine that deviates considerably from the traditional design.

We evaluate the performance of Cogset by comparing it to a widely used traditional MapReduce engine using a previously established benchmark. The results confirm our thesis that a high-performance MapReduce engine can be based on static routing, although analysis indicates that the reasons for Cogset's performance improvements are more subtle than expected. Through our work we develop a better understanding of static routing, its benefits and limitations, and its ramifications for a MapReduce engine.

A secondary goal of our work is to explore how higher-level abstractions that are commonly built on top of MapReduce will interact with an execution engine based on static routing. Cogset is therefore designed with a generic, low-level core interface, upon which MapReduce is implemented as a relatively thin layer, as one of several supported programming interfaces.

At its core, Cogset provides a few fundamental mechanisms for reliable and distributed *storage* of data, and parallel *processing* of statically partitioned data. While this dissertation mainly focuses on how these capabilities are leveraged to implement a distributed MapReduce engine, we also demonstrate how two other higher-level abstractions were built on top of Cogset. These may serve as alternative access points for data-intensive applications, and illustrate how some of the lessons learned from Cogset can be applicable in a broader context.

Contents

List of Figures

Acknowledgements

Back in 2004, I was one of the software developers working for Fast Search & Transfer (FAST), a leading actor and recognized innovator in the enterprise search business; presently a part of Microsoft. One morning, professor Dag Johansen, my future Ph.D. advisor, stopped by the office and brought a recently published paper to our attention: it described a new system called MapReduce, that greatly simplified distributed processing of large, unstructured data sets.

As programmers with first-hand experience with the practical challenges posed by distributed applications, we immediately recognized the significance of MapReduce, and looked for ways to apply similar principles in our own software. In one meeting, while we were reflecting upon the restricted precedence graph implicit in MapReduce, pondering ways to support a more expressive programming interface, Frode Vatvedt Fjeld remarked that a precedence graph could be specified using a regular Makefile. The next day I built a prototype version of a distributed "make" program that quickly evolved into a new foundation for our web analysis platform.

This kind of rapid innovation is a good example of my time in FAST, and I thank all of the people I worked with there for inspiring me and expanding my horizons. In particular I'd like to thank Tåle Skogan and Håkon Brugård for being very useful discussion partners not just back then, but throughout the course of my Ph.D. work. In general, many of the ideas I explore in this dissertation can be traced back to discussions I have had with people in FAST.

At the University of Tromsø, I have also enjoyed a welcoming, supportive, and creative workplace environment. Many of my colleagues here and in the iAD research project have aided me with useful suggestions, discussions, and insightful feedback along the way. Åge Kvalnes, Håvard Dagenborg Johansen, Audun Østrem Nordal, Dmitrii Zagorodnov, and professor Johannes Gehrke have all been particularly helpful at various stages of my work.

Professor Dag Johansen deserves a special mention, as my Ph.D. advisor and academic mentor for several years. His undying enthusiasm is infectious, and I thank him for all his good advice, for his persistent support, and for his unwavering faith in me and my work. I also thank my second advisor, professor Bjørn Olstad, for inspiration and encouragement.

Finally, I am grateful for my dear girlfriend Mona, and for our lovely children. They made it all worth my while.

Chapter 1

Introduction

Modern data centers commonly contain very large clusters of commodity computers, built from low-cost hardware components. Individually, these machines are generally no more powerful than the average home computer, and apart from a more compact physical design that allows them to be stacked in racks, they are quite unremarkable. However, by cooperating, having each participant work in parallel on a small part of a much bigger overall problem, clusters of low-end machines can perform highly demanding computations, such as weather forecasting, DNA sequencing, or sifting through the vast and rapidly growing collection of interlinked web pages that comprise the World Wide Web.

Ideally, if one machine takes two hours to solve a problem, one could expect two cooperating machines to do it in one hour. Such an ideal is called *linear speedup*. Additionally, if those two machines could solve a twice as large problem in the same time as one machine, there would be so-called *linear scaleup*. Three reasons why these ideals are hard to attain in practice are: *(1)* in order to divide a problem into suitable sub-problems and prepare all participants, there will be some start-up and communication overhead, which eventually grows dominant, *(2)* as the number of participants increases, it will be increasingly hard to devise evenly sized sub-problems for all of them, causing bottlenecks, and *(3)* there may be contention for shared resources, causing *interference* between participants and overall slowdowns.

To minimize the problem of interference, a much-adopted solution is to avoid centralized storage systems and shared memory. Instead, each participating machine is a regular low-cost commodity computer with its own private CPU, memory and local hard drive, and the machines are loosely interconnected in a generic local area network (LAN) of the kind found in many workplaces. Successfully deployed at ever-increasing scales, such *shared-nothing* architectures have proved to be a very cheap way of scaling [1]. Commodity hardware, when purchased in bulk, is sufficiently inexpensive that faulty components can simply be replaced rather than diagnosed and repaired. On the other hand, in an environment where individual components are generally unreliable and even considered disposable, it is by no means trivial to program robust distributed applications. A simple and inexpensive approach to scaling hardware can thus lead to a proliferation of highly complex distributed software.

To combat this complexity, there is great demand for modular software abstractions that factor out specific functional concerns, to which programmers must explicitly relate, while re-using established solutions for generic non-functional concerns such as distribution of data, inter-process communication and synchronization, and fault tolerance. Examples include distributed file systems, which focus on reliably storing data in an unreliable environment by

replicating it on multiple machines; naming services, which provide an abstract namespace through which distributed processes can establish communication; and distributed lock services, which allow processes that execute in parallel to synchronize their actions as required in order to ensure correct results. Such *software infrastructure* emerged as a natural consequence of distributed computing in general and shared-nothing hardware architectures in particular.

As new application areas and deployment scenarios appear, introducing new requirements, the software abstractions that support shared-nothing clusters continue to evolve. Two specific developments have been particularly influential in recent years: the increased focus on *data-intensive computing*, and the advent of *cloud computing*. The next sections describe these particular developments and how they have motivated new software infrastructure and novel programming models.

1.1 Data-Intensive Computing

For most traditional high-performance computing applications, such as weather forecasting, cryptography, or protein folding, performance is mainly limited by the available CPU time. Such *compute-intensive* applications may informally be characterized as number-crunching applications, since they operate on small and well-defined data sets, but perform extensive computations in order to produce the desired results (e.g, the future state of a simulated weather model, or the prime factors of a large integer).

In contrast, the performance of *data-intensive* applications is mainly limited by the amount of data to be processed. Such applications are particularly sensitive to scheduling algorithms and data placement, due to the high cost of accessing data and shuffling it between participating machines. This introduces a new set of requirements and trade-offs, and techniques such as batching, caching, and co-location of related data items may be instrumental for optimizing data-intensive computing.

The explosive growth of the world wide web brought a new set of data-intensive applications. As an unprecedented source of rapidly growing and largely unstructured data, the web introduced new technical challenges as well as business opportunities. Companies found ways to profit not just from serving and distributing web data, but also from collecting and analyzing it. Perhaps most notably, a 1998 startup company named Google Inc. was highly successful and prospered greatly from providing a conceptually simple service: *web search*.

The web has become the default medium for publishing electronic information, but it has no built-in facilities for locating information of interest. To provide such a service, a large portion of the web must be downloaded, processed and analyzed in various ways and finally indexed—and in order to provide up-to-date search results at all times, the process must be repeated as frequently as possible in a continuous cycle [2]. Google achieved this by establishing an environment for data-intensive computing based on large shared-nothing clusters, which were scaled up to unprecedented sizes in step with the ever-increasing amounts of data and users on the web.

As one of the first companies to attempt large-scale web indexing, Google found existing software abstractions lacking or unsuitable, and developed a new suite of software infrastructure, aimed specifically at supporting large-scale data-intensive computations. Many of the individual systems that comprise this infrastructure have been the subject of academic publications [3, 4, 5, 6, 7, 8, 9, 10] and received considerable interest, since they demonstrate practical approaches that have been deployed in live production environments on very large scales.

In particular, the MapReduce programming model [6] made a great impact by demonstrating a simple, flexible and generic way of processing and generating large distributed data sets. MapReduce programs are written in a particular functional style and may be executed within a framework that automatically enables distributed and highly parallel execution. MapReduce was quickly embraced as a new paradigm for data-intensive computing, and widely adopted by other companies working with web-scale data sets. For example, Hadoop—an open source implementation of MapReduce—is currently used by major companies such as Amazon, eBay, Facebook, Twitter, Yahoo!, and many others [11].

1.2 Cloud Computing

The adoption of MapReduce and similar high-level programming models was encouraged by another concurrent development: a new paradigm for software deployment known as *cloud computing* [1]. The aim of cloud computing is to turn computing power into a common utility that is always available on demand and in abundant supply, much like electrical power delivered through the electricity grid. This long-held dream has recently become economically viable, with the construction and operation of extremely large-scale, commodity-computer data centers at low-cost locations.

Cloud providers allow customers to purchase on-demand access to computing power in various forms, for example as a cluster of virtual machines, provisioned from a much larger underlying cluster of physical machines. Customers may log in remotely and execute their own software on the virtual machines, and purchase access to additional machines whenever the need arises. As a result, computing clusters are now more accessible than ever: there is no need for small companies to make large up-front investments in hardware to provision for peak loads or anticipated future growth—virtual computing clusters hosted in the cloud may simply be expanded on demand, minimizing financial risks [1].

To fully exploit the potential of cloud computing, applications should be able to scale up or down with minimal disruption of service, and without excessive tuning and reconfiguration. This property is referred to as *elasticity*, and extends the list of requirements placed on software infrastructure for cloud environments. Conversely, high-level software abstractions and programming models are particularly attractive for cloud applications, given the dynamic nature of the underlying hardware environment.

MapReduce programs do not explicitly relate to the division of work and distribution of data, nor to the many low-level complications that distributed execution entails. As long as the application obeys certain restrictions imposed by the programming model, the MapReduce engine ensures fault-tolerant and scalable execution. A MapReduce program may thus be developed and tested on a single machine or small cluster, and be deployed with a high degree of confidence on a much larger scale. This makes MapReduce a particularly good match for data-intensive applications executing in the cloud. Amazon's commercial cloud computing platform, Amazon Web Services, includes MapReduce support both as a separate high-level web service [12], and in the form of customized virtual machine images.

While hard to quantify, a final and frequently emphasized reason why MapReduce appeals to a broad audience of software developers is its *simplicity*. With the increased accessibility and availability of distributed computing clusters, enabled by cloud computing, development of distributed software is no longer a field exclusive to domain experts. Using MapReduce,

even novice programmers can learn to make effective use of large computing clusters [6]. This is due to a remarkably simple programming interface, which is described next.

1.3 MapReduce

In the MapReduce programming model, data sets are modeled as collections of key/value pairs. A MapReduce program processes one such data set and produces another. The programmer is relieved of non-functional concerns such as data distribution, scheduling, and fault tolerance, and is only required to provide two functions: a *map function* and a *reduce function*. The map function is applied to each input key/value pair, producing a set of intermediate key/value pairs. The MapReduce engine groups the set of intermediate pairs by key, and the reduce function is invoked once for each unique intermediate key. The reduce function may access all of the values associated with the given intermediate key using an iterator, and emits the key/value pairs that constitute the final output.

As an example of a computation that could be implemented as a MapReduce program, consider the task of constructing an *inverted index*. An inverted index is a fundamental index data structure used by search engines to associate index terms (e.g., alphanumeric strings) with indexed documents. In a web search engine, the input data set would typically be a set of *(url, html)* pairs, associating URLs with the HTML source of the corresponding web pages. The desired output would be a mapping from index terms to the URLs of the pages in which the terms occur. Using MapReduce, this could be accomplished by mapping all input *(url, html)* pairs to sets of intermediate *(term, url)* pairs, and reducing all *(term, url)* pairs for each unique term to a final *(term, url-list)* pair, where a *url-list* would be a set of URLs, typically encoded in a compact form. The map function in this example would essentially be a custom HTML parser that extracts and emits all index terms from a web page, while the reduce function would encode the list of URLs for a given index term.

The MapReduce programming model was introduced in 2004 [6]. In its abstract form, it is little more than an interface specification, and a functional MapReduce engine is trivial to implement. For example, Section 2.3.2 includes a minimal but functionally complete single-process MapReduce engine implemented in just 20 lines of Python code. The most significant contribution of the initial MapReduce paper was to describe how Google had designed and implemented a highly scalable and fault-tolerant *distributed* MapReduce engine and deployed it on clusters with thousands of machines. As such, the term MapReduce, unless otherwise qualified, usually refers to distributed MapReduce engines built along similar design principles as Google's original implementation, with similar non-functional properties. Throughout this dissertation, we will refer to such MapReduce engines as *traditional* MapReduce engines.

A traditional MapReduce engine is based on principles found in *dataflow engines*, a core component of parallel databases [13], and a natural building block for data-intensive computations. Despite this common ground, MapReduce differs from parallel databases in several ways:

Programming Interface MapReduce programs are written in a general-purpose programming language such as C++ or Java. The MapReduce engine is invoked using a library that allows a program to start executing as a single process before branching into a full-fledged distributed computation. The map and reduce functions are regular functions, implemented in the general-purpose programming language, and integration with existing code is therefore trivial.

This differs from the typical database approach, where computations are initiated by formulating queries in a separate, domain-specific query language—typically the Structured Query Language (SQL) or some restricted subset of SQL. Such query languages are generally unsuitable for expressing complex functions such as the HTML parsing from the preceding MapReduce example.

Fault Tolerance Traditional MapReduce engines target data-intensive applications and are highly resilient to machine failures. Long-running computations are supported and expected, and the traditional design tolerates machine failures with minimal disruption of the overall progress.

Parallel databases are optimized for high-throughput execution of numerous short-running queries and designed to handle failures by restarting all queries in progress.

Data Model MapReduce models all records as simple key/value pairs, and parses the input data at run-time into sequences of key/value pairs according to a user-specified input format that may be customized programmatically.

Parallel databases use schemas to model data, typically using the *relational model* [14]. Schemas enforce a well-known pre-defined structure for all records, and facilitate certain optimizations such as compression, indexing, and column-based storage, but schema-based approaches may be a poor fit for inherently unstructured data.

Decoupled Storage Traditional MapReduce engines are loosely coupled to the supporting storage system, which is typically a block-based distributed file system. The absence of schemas means there is no requirement to initially import and convert the data into a special database-specific storage format. Data may therefore be processed *in situ*, by implementing an input format that reads and parses records directly from the distributed file system.

A downside of this loose coupling is that data placement decisions are made independently from the processing logic. This may lead to poor data locality and be detrimental to performance for many computations. For example, relational joins benefit greatly from co-locating matching partitions of the input data sets, which is hard to ensure using a traditional MapReduce engine. Parallel databases closely integrate storage and processing to address these concerns.

1.3.1 Data Shuffling

A traditional MapReduce engine executes a program in two phases. In the first phase, a number of map tasks are executed; each map task reads a separate partition of the input data and invokes the map function for each key/value pair. In the second phase, a number of reduce tasks are executed; each reduce task processes a separate partition of the intermediate key space, invoking the reduce function for each intermediate key in that partition.

Before the second phase can begin, all intermediate data must be grouped and partitioned by key. An essential aspect of a MapReduce engine is the algorithm used for grouping intermediate data and transporting it to the machines that execute the relevant reduce tasks. This part of a MapReduce computation, known as *data shuffling*, is traditionally implemented using an algorithm we will refer to as *dynamic routing*. With dynamic routing, the output of each map

task is partitioned according to the total number of reduce tasks and temporarily stored on local disk. When a reduce task executes, it fetches a specific output partition from every map task, collecting and merging all intermediate key/value pairs for a given partition of the intermediate key space. Reduce tasks may therefore execute on any available machine, and intermediate data is copied on demand to the appropriate machines.

In contrast, parallel databases employ what we refer to as *static routing* of records, relying on a predetermined configuration or query plan to decide how data should be partitioned, and which machines are responsible for storing or processing each partition. Since the destination machine of each record can be determined immediately, communication can be *push-based*, streaming data directly to a subsequent consumer, and intermediate temporary storage can be avoided in many cases.

While efficient under many circumstances, two potential drawbacks of static routing are *(1)* poor fault tolerance, and *(2)* poor load balancing in the presence of *data skew*. Both result from the static assignment of partitions to machines: if a machine fails, all data routed to that machine is lost and the entire computation must be restarted, and if a machine is overloaded with data to process, it is difficult to off-load its work.

By using dynamic routing, traditional MapReduce engines effectively checkpoint the computation at frequent intervals. The flexible assignment of tasks to machines allows machine failures to be tolerated simply by re-executing failed tasks elsewhere. Partial output from failed tasks can trivially be discarded, since they only write to temporary storage while executing (as opposed to directly streaming data to subsequent tasks).

More generally, dynamic routing allows MapReduce engines to tolerate not just failed machines, but also slow or overloaded machines. Provided the computation is subdivided into a relatively large number of tasks (compared to the number of machines employed), load can easily be balanced across a heterogeneous cluster by allowing fast machines to execute more tasks than slow machines. Static routing is less flexible, because tasks may be tied to specific machines and unable to execute elsewhere. As a result, static routing is more vulnerable both to hardware heterogeneity and to data skew, where an uneven distribution of keys may require a minority of the available machines to process a majority of the records, causing performance bottlenecks.

In summary, dynamic routing facilitates both fault tolerance and load balancing, but has a potential overhead that stems from the temporary storage of intermediate output partitions on local disk. This incurs additional I/O compared to an approach that streams data directly to a subsequent consumer. In addition, when the intermediate output partitions are fetched on demand, the resulting I/O access pattern can cause excessive disk seeking, which is detrimental to performance [15].

1.4 Thesis Statement

Since dynamic routing is a core mechanism in the traditional design of a MapReduce engine, central to both fault tolerance and load balancing, it may be interesting to explore the properties of a MapReduce engine based on static routing. Such an engine could shuffle data by streaming it directly to the appropriate nodes, reducing the need for intermediate temporary storage. On the other hand, this would require alternative mechanisms for fault tolerance and load balancing, and a tighter coupling to the underlying storage layer. We conjecture that such

a design is feasible, and that it could result in improved performance. Specifically, the *thesis of this dissertation* is that:

It is possible to build a high-performance MapReduce engine based on static routing.

To evaluate this thesis, we must either adapt a traditional MapReduce engine such as Hadoop to use static routing, or design and implement a new engine based on static routing. The former approach might facilitate evaluation of the thesis, since our adapted MapReduce engine could be compared directly to the original version. On the other hand, when adapting an existing engine our options would be limited by existing design choices. We will therefore choose the latter approach and design and implement a new MapReduce engine from the ground up. Our central design choice of static routing will be allowed to shape the remainder of the design, subject only to the constraints that are imposed by other requirements that MapReduce engines are commonly expected to meet. Specifically, these additional requirements are as follows:

Fault Tolerance The MapReduce engine should let applications continue to make progress, and complete in a timely manner, in spite of individual machine failures.

Load Balancing The MapReduce engine should be able to compensate for slow machines by balancing load in an adaptive manner.

Reliable Storage The MapReduce engine should either be coupled to, or integrated with a component that offers reliable, replicated storage of data sets to be processed.

To facilitate performance evaluation, we introduce a fourth requirement, for compatibility with Hadoop, the most widely deployed open source implementation of a traditional MapReduce engine.

Compatibility The MapReduce engine should be compatible with Hadoop, allowing existing MapReduce applications written for Hadoop to be executed with minimal changes.

This fourth requirement allows existing applications and previously established benchmarks to be used for the evaluation of our thesis. Furthermore, innovations or existing higher-level abstractions layered on top of Hadoop, such as the Pig Latin query language [16], will remain applicable to our MapReduce engine.

The common approach of stacking additional functionality and abstractions in layers on top of MapReduce—as exemplified by Pig Latin—raises the question of how such higher-level abstractions might leverage an engine based on static routing. A natural extension of our work is to investigate the potential implications that a redesigned core engine might have for higher-level abstractions. This motivates a fifth requirement.

Extensibility The MapReduce engine should fully expose the fundamental mechanisms that enable reliable and distributed *storage* of data, as well as parallel *processing* of statically partitioned data—independently of the MapReduce interface.

In other words, while a MapReduce interface compatible with Hadoop is one requirement, this interface should be built as a layer on top of a more generic, low-level interface that exposes the engine's full set of capabilities. This final requirement allows us to freely experiment with higher-level abstractions that exploit static routing, independently of the semantics that would be enforced by going through the MapReduce interface.

1.5 Scope and Assumptions

Throughout this dissertation, we make certain assumptions about the hardware environment and problem domain, both to focus our attention and guide our design choices, and as the implicit backdrop for our discussion. We document these assumptions here, and define the scope of our research by specifying any limitations and deferred lines of inquiry.

- We target a distributed shared-nothing environment [17], where machines have locally attached storage, no shared memory, and only communicate by exchanging messages over an unreliable communication network.
- We restrict our problem domain to applications that process large data sets, which cannot fit in the combined main memory of the available machines. In other words, the computations must involve secondary storage. We also assume that network I/O can be a potential bottleneck (i.e. we do not assume infinite or extremely high network bandwidth).
- We assume the predominant form of available non-volatile secondary storage are magnetic hard disks, or devices with similar performance characteristics. Although non-volatile storage comes in many other forms, the cost-efficiency of magnetic hard disks still makes them the preferred alternative when working with large data sets.
- We adopt the *fail-stop* failure model [18]. In other words, we make the common assumptions that *(1)* processors will halt on failure, rather than make erroneous state transformations, *(2)* processors can detect when other processors have failed, and *(3)* there is some amount of *stable storage* available which will be unaffected by failures. Stable storage can be approximated through replication. We also assume synchronous communication, where there is an upper bound on message latency. In combination with the fail-stop model, this allows failures to be detected via *pinging*, i.e. by exchanging regular status messages to signify liveness.
- While scalability is an important concern, we limit our evaluation to small and medium-scale computing clusters. This is for practical reasons, to allow for rapid development and deployment, and flexible experimentation in a controlled environment. While large clusters of virtual machines could be provisioned from a commercial cloud provider, such a cluster would be harder to monitor and manage. Software and configuration changes would take longer to deploy, and experiments in such an environment would not be reproducible, diminishing their scientific value.

1.6 Methodology

Computer systems are man-made, which to some may disqualify them as worthy objects of study within the natural sciences. However, at its core, computer science is the study of information processes, and such processes do occur in nature. Computers need not rely on electronic hardware; they can also be implemented on alternative physical media such as biomolecules or trapped-ion quantum computing devices. The first programmable computer, designed by Charles Babbage in the early 19th century, was built of gears and mechanical components, and powered by cranking a handle [19]. Conversely, many processes occurring in nature can be

viewed as instances of information processing—though such naturally occurring computations may be running on what we would view as exotic hardware.

With this in mind, computer science does meet every criterion for being a science [20]. As with other sciences, computer science research relies on a body of techniques collectively known as the *scientific method*. A scientific method of inquiry must be based on gathering observable, empirical and measurable evidence subject to specific principles of reasoning. Using the *hypothetico-deductive method*, explanations for phenomena are sought by formulating hypotheses and comparing their predictions to experimental observations. Correct predictions may strengthen a hypothesis, while observations that conflict with the predictions may falsify it. A falsified hypothesis must be discarded or modified to account for the new evidence. This is an iterative process: based on the results of experiments, conclusions may be drawn that serve as a starting point for a new hypothesis, from which additional predictions are made, leading to the design of a new set of experiments.

The field of computer science is commonly divided into three disciplines, which correspond to different paradigms for research [21]:

Theory Rooted in mathematics, this discipline studies objects whose properties and relationships can be clearly defined and reasoned about using logical reasoning. A prime example is the study of algorithms; given sufficiently detailed descriptions, hypotheses about algorithms (such as the hypothesis that a given algorithm will eventually terminate) can be proved using logical reasoning.

Abstraction Rooted in experimental science, this discipline constructs models based on hypotheses or through inductive reasoning about observable objects or phenomena. The studied objects could be software or hardware components, or the holistic behavior of a complex computer system. The model is evaluated by comparing its predictions to experimentally collected data. Abstraction resembles the scientific disciplines within natural sciences like biology, physics and chemistry. Their common goal is to construct accurate models of the rules and laws that govern the behavior of observable objects. Accurate models can be used to predict the behavior in circumstances that have not been observed experimentally.

Design Rooted in engineering, this discipline uses a systematic approach to construct systems or devices that solve specific problems. A set of requirements describes the functional and non-functional characteristics of the construct. Next, the system or device is specified, designed and implemented. Finally, the construct is tested to verify that it meets the stated requirements. If not, the process is repeated, refining and improving the end product with each new iteration.

In practice, these disciplines are intertwined, and research typically draws upon all three paradigms to varying degrees. This dissertation is not of a theoretical nature, but we draw upon much established theory, for example regarding the inherent properties and limitations of distributed systems. We use abstraction to reason about system behavior at a high level and form hypotheses about how that behavior will be affected by architectural changes. Through experiments we check if our high-level model correctly predicted system behavior.

Our central thesis is evaluated by allowing it to dictate a central design choice for an experimental MapReduce engine. When complemented by a set of additional requirements,

this forms a specification that we translate into a complete design and working implementation using the methodology of the design discipline. In this dissertation, we present the end product—a working MapReduce engine—but also draw conclusions based on experience from earlier iterations of the process, where we encountered unexpected complications or synergies that affected the final design.

This iterative process is a core aspect of *systems research*: after designing and implementing a prototype, we test and experiment with it, and use the resulting experience to design new versions. In the context of engineering, this process ideally culminates in a finished product that meets the stated requirements. In systems research, the process is explorative, speculative, and open-ended.

Experiments are the backbone of our research. This applies both in the general sense of building experimental systems that challenge fundamental assumptions and central design choices, and in the specific sense of conducting unbiased and reproducible experiments in a controlled environment. Based on experience, intuition, and creativity, we devise and experiment with untraditional approaches and explore new territory. Through empirical measurements we observe how our experimental systems behave, and gain new experience.

1.7 iAD Context

This work is part of the Information Access Disruptions (iAD) project. Partially funded by the Research Council of Norway as a Centre for Research-based Innovation (SFI), iAD is an international project directed by Microsoft Development Center Norway (originally by Fast Search & Transfer) and includes multiple other commercial and academic partners: Accenture, Cornell University, Dublin City University, BI Norwegian School of Management and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO). The iAD Centre targets core research for the information access domain. This domain includes but is not limited to search and search-derived applications.

The iAD research group in Tromsø focuses on fundamental structures and concepts for runtime systems supporting large-scale information access applications. Such applications commonly require a framework for performing data-intensive *analytics*, for example to analyse web graphs or compute recommendations for end users. This role would traditionally be filled by a MapReduce engine; the work presented in this dissertation is part of an on-going effort to investigate alternative approaches to analytics.

1.8 Summary of Contributions

This dissertation makes the following contributions.

- We have designed and implemented *Cogset*—a new MapReduce engine that diverges considerably from the traditional design. In accordance with our thesis, Cogset is designed to use static routing of data. In turn, this required us to design and implement a new set of mechanisms to meet the additional requirements of fault tolerance and load balancing. Cogset demonstrates how static routing can be applied by a MapReduce engine, incorporating architectural elements from parallel databases, while preserving the non-functional properties commonly associated with MapReduce. This is achieved through a design that combines new and previously established techniques into a novel

composition. The design and implementation of Cogset are presented in chapters 3 and 4.

- We have evaluated the performance of Cogset by comparing it to the widely deployed Hadoop engine, in order to determine whether or not Cogset qualifies as a high-performance MapReduce engine. To ensure an unbiased comparison, we employed previously established benchmarks developed for Hadoop. Our results show that Cogset performs better than Hadoop for a range of benchmark tasks, with speedups of up to 100%. The details and results of this evaluation are presented in Chapter 5.

- As a by-product of our experimental evaluation, we have uncovered specific performance issues with the Hadoop engine. Chapter 5 also describes the analysis and investigation that caused us to suspect and allowed us to identify these issues. In response, we have developed a custom plug-in for Hadoop to partly address one issue, relating to multi-core CPU performance, without modifying the core Hadoop code. We also show how to patch Hadoop's task scheduling algorithm to address another issue that causes excessive idle time. Our evaluation confirms that these changes are effective at improving Hadoop's performance, closing some of the performance gap between Hadoop and Cogset.

- We have investigated how higher-level abstractions that offer different entry points for data-intensive applications can be built using Cogset as a core engine, potentially bypassing the MapReduce API. As part of this work, we have developed two new abstractions. *Oivos* allows workflows involving multiple related data sets to be expressed in a declarative manner, by automatically compiling suitable dataflow graphs that are executed using Cogset. *Update Maps* leverage Cogset's support for efficient relational joins into a new abstraction for batch processing that mimics a key/value database, but replaces its traditional synchronous interface with a mechanism for making asynchronous updates. These new abstractions are discussed in Chapter 6.

1.9 Outline of the Dissertation

In this chapter, we have presented the background and motivation for our research. We have stated our thesis, described our methods, and summarized our main contributions. The remainder of the dissertation is structured as follows.

- Chapter 2 surveys previous work of particular relevance, including early work on dataflow graphs, and the more recent generation of systems inspired by MapReduce.

- Chapter 3 presents the overall architecture and design of Cogset.

- Chapter 4 details the implementation of Cogset.

- Chapter 5 evaluates the performance of Cogset by benchmarking it against Hadoop, discusses the results, and details our related optimizations of Hadoop.

- Chapter 6 discusses how to build higher-level abstractions on top of Cogset, and demonstrates such extensibility by presenting the new Oivos and update map abstractions.

- Chapter 7 concludes and outlines possible future work.

Chapter 2

Related Work

In this chapter we discuss previous work of particular relevance to Cogset. We begin by reviewing the concepts of dataflow graphs and dataflow engines, which are central components in parallel databases and a foundation for MapReduce. Next, we describe the MapReduce programming model in detail through a series of example applications, and outline the traditional architecture of a distributed MapReduce engine. We also describe various specializations and refinements of MapReduce, a number of higher-level abstractions implemented by using MapReduce as a building block, and hybrid systems that combine architectural elements of MapReduce with conventional database technology.

2.1 Dataflow Graphs

One way to relate MapReduce to previous work is by viewing it as a way of specifying and executing *dataflow graphs*. In a dataflow graph, a collection of operators is composed into a directed acyclic communication graph. Whenever the output from one operator serves as the input to another operator, there is a corresponding edge in the dataflow graph. While each operator may be restricted to processing its input sequentially, parallelism can still be achieved either by pipelining operators (allowing consumers and producers to execute concurrently), or by partitioning data and processing all partitions in parallel using multiple replicated operators. These two patterns are referred to respectively as *pipelined parallelism* and *partitioned parallelism*.

Dataflow graphs may be constructed explicitly by a programmer, or generated automatically by a compiler. Once constructed, a dataflow graph can be analyzed using well-known graph algorithms, and any potential for parallelism will be evident from its structure. An orthogonal line of research concerns how to automatically discover and exploit parallelism in sequential programs. One approach to automatic parallelization is to rewrite sequential programs into dataflow graphs according to deduced data dependencies—this approach hinges on accurate algorithms for determining data dependencies. A pragmatic middle ground is to employ programming models that explicitly expose data dependencies, facilitating automatic compilation into dataflow graphs.

Regardless of their actual method of construction, dataflow graphs remain important abstractions for parallel processing. Once a computation is expressed as a dataflow graph, many aspects of its execution can be automated by a generic execution engine. For example, operators in the graph can be scheduled to run automatically once the required input data is available

(in a so-called *data-driven* computation), or the engine can deduce which operators to execute in order to produce some particular output (in a *demand-driven* computation).

Dataflow graphs are based on fundamental ideas and insights that date back several decades. The notion of structuring programs as simple building blocks composed into more elaborate directed graphs according to their data dependencies dates back at least to the late 1950s [22, 23, 24]. During the 1960s, this topic was explored extensively, typically in the context of job scheduling constrained by a precedence graph [25, 26, 27]. The first graphical user interface for specifying programs modeled as precedence graphs appeared in 1966 [28]. The modern notion of dataflow programming, as graph-structured parallel computations, was first clearly articulated in 1970 [29], although most subsequent work in the 1970s was based on a more restrictive model [30]. Hardware architectures based on the dataflow model also emerged[31]. Although these had limited commercial success, out-of-order execution in modern processors resembles a dataflow architecture in that instructions within a given window are allowed to execute concurrently while completing in data dependency order.

During the 1980s, state of the art high-performance database systems moved from running on mainframes with special-purpose hardware to running on shared-nothing clusters [32, 33, 34]. At the core of this new generation of parallel database systems were software engines for executing dataflow graphs [13]. The next section describes these software engines and how they inspired or evolved into more generic, stand-alone dataflow engines.

2.2 Dataflow Engines

Parallel database systems employ internal dataflow engines to evaluate queries. The data base tables are partitioned, distributing relational tuples among the available machines. Queries, usually expressed in the Structured Query Language (SQL), are compiled into dataflow graphs containing relational operators, which are in general well-suited for partitioned parallelism [13]. Operators exchange tuples, filtering, sorting, aggregating and correlating them in order to produce the final query result. For example, an operator might filter a sequence of tuples such that each tuple only retains a certain subset of attributes, implementing *projection*, a fundamental operation in relational algebra.

Since database queries are expressed in a high-level declarative language, the actual means of query evaluation is traditionally viewed as an internal implementation detail of which users should be oblivious. In practice, some insight into the actual query evaluation algorithms and their performance characteristics may be required in order to formulate optimal queries, but the underlying dataflow engine is not directly exposed as a programming abstraction.

In contrast, generic dataflow engines have a lower-level interface, but support a broader range of applications, by allowing programs to be explicitly structured as dataflow graphs. An early example is River [35, 36]—a programming environment that offers abstractions for connecting application-specific *modules* to each other via *queues*. Queues may have multiple producers and consumers, and may connect modules running locally on the same machine, or modules distributed on different machines. River focuses on supporting I/O-bound applications in heterogeneous environments, and is tightly coupled with an underlying I/O substrate. Dynamic load balancing is provided through the *distributed queue* abstraction, which allows multiple consumers to consume data at different rates from a single multi-consumer queue, and through the *graduated declustering* algorithm, in which the effective disk read bandwidth offered from

individual disks to specific clients is adjusted to compensate for perturbations in disk performance.

More recently, Microsoft Research developed Dryad [37], a general-purpose distributed execution engine for applications structured as dataflow graphs. Dryad succeeded—and was directly inspired by—Google's implementation of MapReduce, but we describe it here in the more general context of dataflow engines. Similar to River modules, the vertices of a Dryad application may reside in the same local process or be distributed on multiple machines. Vertices communicate through channels based on TCP pipes, shared-memory FIFO queues, or temporary files. In the latter case, the channel is effectively a persistent data buffer, and its two endpoint vertices (the producer and the consumer) do not have to execute concurrently. Accordingly, the channel may serve as a checkpoint from which execution can be resumed in case of failures.

Dryad applications compose dataflow graphs using a generic graph composition language that is embedded into C++ through a combination of operator overloading and regular method calls. The core of the language is a graph object, which encapsulates a directed acyclic graph in which certain vertices are tagged as *input vertices* and others are tagged as *output vertices*. Edges represent communication channels and may not enter input vertices or leave output vertices. Large graphs may be composed from simpler subgraphs by applying operators to existing graph objects, starting initially with singleton graphs constructed from individual vertices. For example, the unary ^ operator may be used to create multiple replicas of a single graph, and the binary >= and >> operators combine two disjoint graphs into a new graph by connecting the output vertices from one graph to the input vertices of the other (using pointwise or complete bipartite composition, respectively). The binary || operator merges two arbitrary subgraphs that may have vertices in common, allowing the expression of other, potentially asymmetric graph topologies.

Dryad only provides point-to-point communication channels; although multi-producer, multi-consumer channels such as the distributed queues in River can be implemented by inserting an intermediate vertex to which all of the producers and consumers are attached, any associated load balancing would have to be implemented by user code inside the vertex. The main strengths of Dryad, which distinguish it from River, are its abilities to automatically *deploy* a dataflow graph, mapping vertices to a set of available machines, and to provide fault-tolerant execution of a deployed dataflow graph.

For efficiency, Dryad allows the mapping of vertices to machines to be explicitly guided by constraints or preferences that are manually assigned to each vertex. For example, input vertices should be co-located with the data they intend to read, and vertices that exchange high data volumes using TCP pipes may prefer certain subsets of machines to take advantage of network topology. Dryad has a central *job manager* component that performs the initial deployment of vertices in accordance with the given constraints and goes on to schedule additional vertices for execution as their input channels become available. Vertices are required to be deterministic, so if a vertex fails, the scheduler simply arranges for it to execute again using the same input.

The above mechanisms have certain limitations related to the use of *transient channels* such as TCP pipes and shared-memory FIFOs. Vertices connected by transient channels have to execute concurrently, and upon failure, errors will propagate along the channels, making large connected components of vertices fail as a unit. As such, Dryad programs must take care not to make excessive use of transient channels, to ensure that enough machines can be allocated at all times, and to maintain the desired level of fault tolerance by inserting occasional persistent file-based channels to serve as checkpoints.

An interesting aspect of Dryad is the ability to dynamically restructure a dataflow graph while it is being executed, by adding new vertices or edges. Each vertex is assigned to a particular *stage*, and for each stage there is an associated *stage manager* which receives notifications about state changes in its vertices and may react accordingly. For example, the stage manager could dynamically adjust the degree of parallelism to use in a particular stage of the computation based on the observed size of the input data, dynamically repartition the data based on an observed sample, or implement an aggregation tree that adapts dynamically to the network topology. One feature implemented by the default stage manager attempts to prevent a single slow machine from delaying the completion of an entire stage, by monitoring the rate of progress of all vertices and scheduling duplicate executions of vertices that appear to be progressing slower than their peers. The mandatory deterministic nature of vertices ensures that duplicate executions will produce the same output, and downstream vertices are free to use the output from the first duplicate execution to finish. This particular feature was directly inspired by the redundant scheduling of *backup tasks* in Google's implementation of MapReduce [6].

In general, Dryad provides some powerful mechanisms such as deployment constraints, transient channels, stage managers and run-time restructuring of the dataflow graph, but it is largely up to applications to make intelligent use of the mechanisms through customized code. A natural tendency will be to refactor such application code into reusable library code, for example in the form of generic vertices for common tasks, custom stage managers that encapsulate specific communication patterns, or custom routines for constructing entire communication graphs according to certain general parameters. In effect, Dryad may thus serve primarily as a platform for implementing higher-level programming abstractions that are easier to adopt, while the low-level generic interfaces for graph construction remain available to expert users. For example, one specific abstraction that can be built as a layer on top of Dryad is MapReduce—the subject of the next section.

2.3 MapReduce

When the Internet search company Google Inc. started digesting and analyzing large portions of the world wide web on a regular basis, they concluded that no existing system for distributed data processing could meet their requirements. Their computing environment consisted of massive shared-nothing clusters built from inexpensive but unreliable commodity hardware, and scaled up to handle unprecedented volumes of unstructured web data. To make effective and productive use of this environment, programmers required tools that allowed them to focus on the actual application logic, rather than the numerous non-functional concerns such as partitioning and distribution of data, division and scheduling of work, inter-process communication and synchronization, fault tolerance and load balancing.

A dataflow engine excels at parallel data processing, and is able to mask certain non-functional concerns—specifically those of scheduling and synchronization—but in order to explicitly specify a distributed dataflow graph, the details of data placement and partitioning must be exposed to the programmer. Furthermore, some algorithmic insight is required in order to manually construct dataflow graphs that are suitable for efficient parallel execution, and the exact topology of such a graph should be tailored to the number of available machines. The graph topology may also need to be adjusted in order to achieve dynamic load balancing, which is another desirable feature in large clusters. Explicit dataflow programming for a distributed environment therefore requires a certain expertise.

A final important requirement for large-scale distributed computations is fault tolerance. The dataflow engines used in parallel databases are tuned for high-throughput execution of numerous short-running queries and handle failures simply by restarting affected queries. This is inappropriate for long-running distributed computations, which are required to keep making progress even if individual machines fail. Hence, while a low-level generic dataflow engine may be a useful building block for distributed computations, it has to meet new requirements.

To meet these challenges, Google developed MapReduce [6], which couples a simple and generic way of specifying distributed computations with an execution engine that is highly resilient to failures. Under the hood, MapReduce programs are structured as dataflow graphs with a particular topology, to be executed by a custom dataflow engine tailored for the purpose. Rather than explicitly constructing the dataflow graph, programmers merely customize the behavior of certain operators in the graph, as described next.

2.3.1 Programming Interface

MapReduce derives its name from a remarkably simple programming interface, in which two user-supplied functions play a prominent role: a *map function* and a *reduce function*. The map function typically filters or transforms the input data to produce an intermediate data set, while the reduce function aggregates the intermediate data to produce the final output.

The data sets involved are modeled as collections of key/value pairs; a MapReduce program essentially reads one such data set and produces another, as follows. All input pairs are initially passed to the map function, which emits a sequence of intermediate key/value pairs. The MapReduce engine groups the intermediate pairs by key, and then invokes the reduce function once for each unique intermediate key. Using an iterator, the reduce function can access all of the values associated with the given intermediate key. The key/value pairs that are emitted by the reduce function constitute the final output.

2.3.2 Example Applications

MapReduce programs are written in a general-purpose programming language such as C++, Java or Python, invoking the MapReduce engine through a library. The exact appearance of a MapReduce program therefore depends on the language employed and the details of the library interface. We provide some examples here that are written in Python; these can be executed by the example execution engine that we present in the next section. In our examples, key/value pairs are emitted using Python's *yield* statement. In other words, the map and reduce functions are implemented as *generators*: suspendable functions that generate sequences on demand, upon iteration. In other languages that lack this feature, pairs are typically emitted by invoking a provided callback function.

Our three examples are idiomatic MapReduce applications selected to represent three partially overlapping classes of computations: *grouping*, *aggregation*, and *sorting*. These reflect different aspects of the exact MapReduce semantics; we detail these semantics and their implications as we present the examples.

Birthday Grouping Example

This example illustrates how MapReduce can be used to group records according to custom criteria. To apply MapReduce for grouping of records, associate a unique intermediate key with

```
def mapper(name, birthday):
    yield (birthday.strftime('%a'), name)

def reducer(weekday, names):
    yield (weekday, '/'.join(names))

# Sample input:
input = [('Alice',   date(1980,  5, 21)), ('Bob',   date(1977, 8, 11)),
         ('Charlie', date(1962, 11,  3)), ('David', date(1972, 4, 19))]

print(MapReduce(mapper, reducer, input))

# Output: [('Sat', 'Charlie'), ('Thu', 'Bob'), ('Wed', 'Alice/David')]
```

Figure 2.1. Birthday grouping example implemented in Python.

```
def mapper(key, text):
    for word in text.split():
        yield (word, 1)

def reducer(word, values):
    total = 0
    for count in values:
        total += count
    yield (word, total)

# Sample input:
input = [(1, 'to be or not to be'), (2, 'that is the question')]

print(MapReduce(mapper, reducer, input))

# Output: [('be', 2), ('is', 1), ('not', 1), ('or', 1),
#          ('question', 1), ('that', 1), ('the', 1), ('to', 2)]
```

Figure 2.2. Word count aggregation example implemented in Python.

```
def mapper(country, stats):
    density = stats['Population'] / stats['Area']
    yield (density, country)

def reducer(density, countries):
    for country in countries:
        yield (country, '%.2f' % density)

# Sample input:
input = [('China',  {'Area': 9640821, 'Population': 1336718015}),
         ('Norway', {'Area':  385252, 'Population':    4943600}),
         ('USA',    {'Area': 9826675, 'Population':  308745538})]

print(MapReduce(mapper, reducer, input))

# Output:
# [('Norway', '12.83'), ('USA', '31.42'), ('China', '138.65')]
```

Figure 2.3. Population density sorting example implemented in Python.

each group and use a map function that emits records as values associated with the appropriate keys. The MapReduce engine guarantees exactly one invocation of the reduce function for each unique intermediate key; the reduce function will thus be invoked once for each group.

The example groups a set of names according to the weekday of their associated birth dates. There are 7 groups, corresponding to the days of the week; their keys are regular strings: *'Mon', 'Tue', ..., 'Sun'*. The output is a single concatenated string of names for each day of the week. This application also illustrates the seamless integration between the MapReduce engine and the general-purpose programming language in which it is embedded; in this case, the functionality for manipulating dates and determining their weekdays is provided by Python's standard library. Figure 2.1 shows the Python code for this application.

Word Count Aggregation Example

This classical example shows how MapReduce can be used to compute custom aggregate functions such as sums and averages over groups of records. As in the previous example, the MapReduce engine performs the actual grouping, according to the intermediate keys emitted by the map function. The aggregation of a group of records is implemented by the reduce function, by iterating over the values associated with a given intermediate key, using the provided iterator. The MapReduce engine guarantees that the full set of values associated with the key will be accessible through the iterator, without omissions or repetitions, ensuring correct aggregation.

The example aggregates the number of occurrences of each unique word in a collection of text data. Each input value is a text string; the map function splits the string into distinct words and emits an intermediate *(word, 1)* pair for each word, reflecting the fact that the given word occurred once in the input text. The reduce function aggregates the occurrence counts for a given word, summing up the total number of occurrences in all of the input text. In this example, the input keys, which presumably provide a context for the text strings, are irrelevant and therefore ignored; they could for example be URLs, file names, page numbers or line numbers. Figure 2.2 shows the Python code for this application. Although Python has a built-in *sum* function to compute the sum of a sequence, the reduce function uses explicit iteration to emphasize the general approach.

Population Density Sorting Example

This example illustrates the ordering guarantees provided by MapReduce. As noted, the reduce function is guaranteed to be invoked exactly once for each unique intermediate key, providing iterator access to a complete set of associated values. Additionally, the MapReduce engine guarantees that the reduce function processes intermediate key/value pairs in increasing key order.[1] This ordering guarantee can be used to sort records according to custom criteria as follows. Let the map function emit records as values associated with intermediate keys that are chosen or constructed to reflect the desired sort order. According to the ordering guarantees, the records emitted by the reduce function will then be output in sorted order.

The example processes a set of countries with associated demographic statistics. It uses the map function to compute the population density of each country, and orders the output in as-

[1] These are the semantics described in the original MapReduce paper. In the absence of side effects, an equivalent guarantee would be to reorder emitted output pairs to correspond to the same *apparent* order of processing.

cending order of population density by using the computed population densities as intermediate keys. The reduce function simply collects and emits all records unchanged. Figure 2.3 shows the Python code for this application. Note how the input and output pairs remain keyed by country, while the chosen intermediate key determines the sort order.

```
def GroupKeys(pairs):
    group_key, group_values = None, []
    for key, value in sorted(pairs):
        if group_key == key:
            group_values.append(value)
        else:
            if group_key is not None:
                yield (group_key, group_values)
            group_key, group_values = key, [value]
    if group_key is not None:
        yield (group_key, group_values)

def MapReduce(mapper, reducer, input):
    # Map all input pairs to produce intermediate pairs
    intermediate = []
    for key, value in input:
        intermediate.extend(mapper(key, value))
    # Group the intermediate pairs by key, and reduce each
    # unique intermediate key, with associated values
    output = []
    for key, values in GroupKeys(intermediate):
        output.extend(reducer(key, values))
    return output
```

Figure 2.4. Minimal MapReduce engine implemented in Python.

2.3.3 Execution Engine

Google's MapReduce engine was designed to facilitate distributed, fault-tolerant processing of large data sets that cannot fit in main memory. Before considering the details of Google's distributed engine, it may be useful to consider how an in-memory single-process MapReduce engine could be implemented. This is quite straightforward, as illustrated by the example execution engine we have implemented in Figure 2.4. Our example engine implements the *MapReduce* entry point that was invoked in the preceding examples. While non-distributed and based on in-memory data structures, the engine is functionally complete. Its execution can be broken down into three steps.

1. *Mapping.* The engine iterates over all input key/value pairs and invokes the map function for each pair, collecting all emitted intermediate pairs.

2. *Grouping.* The intermediate key/value pairs are grouped by key, using sorting. Once the pairs are sorted by key, pairs with equal keys appear as contiguous runs and can be identified in linear time. Hashing could be used as an alternative approach to grouping, but a sorting step would still be required in order to provide the correct ordering guarantees.

3. *Reduction.* The reduce function is invoked for each unique intermediate key. For simplicity, our example engine extracts each group of equal-keyed values into a new list that is passed to the reduce function. It is possible to avoid this memory overhead by implementing a custom iterator that directly accesses the appropriate range of indexes in the originally accumulated list.

We now contrast our single-process example engine to the actual distributed MapReduce engine that was developed by Google. The distributed engine follows the same high-level algorithm, but must divide work between all participating machines in order to benefit from distribution. This is accomplished by *partitioning* the data to be processed, thereby implicitly partitioning the workload. Two parameters, M and R, determine the granularity of this partitioning. To elaborate, Google's distributed MapReduce engine executes a similar sequence of steps as our example engine, but differs in the following respects:

1. *Parallel Mapping*. The distributed engine automatically splits the input data into a number of partitions that may be processed independently and in parallel. Specifically, the input is split into M partitions, and a separate *map task* is scheduled to process each individual partition. A map task sequentially reads its input partition, parsing the input into a sequence of key/value pairs according to a user-specified input format, and applies the map function to each pair.

2. *Parallel Grouping*. The intermediate key/value pairs emitted by the map function are buffered in memory by the map tasks, and periodically written to local disk, partitioned according to their keys into R separate intermediate files. The intermediate key space is thus partitioned (using hashing by default), grouping the intermediate records into R groups. Since the intermediate pairs are partitioned by key, all equal-keyed pairs are assigned to the same group, so the groups can be reduced independently and in parallel.

3. *Parallel Reduction*. To perform the reduction, the distributed engine schedules a second set of tasks. There are R such *reduce tasks*, each of which is responsible for a separate partition of the intermediate key space. Each reduce task must read M intermediate files: one specific output file from each of the M map tasks. The required files are fetched using remote procedure calls, and subsequently sorted into a single sequence of intermediate key/value pairs (held in memory if possible). The sorting is necessary because multiple intermediate keys may map to the same reduce task, and groups the key/value pairs such that pairs with equal keys appear as contiguous sequences. The reduce task can therefore read the sorted pairs sequentially and invoke the reduce function for each unique key that is encountered, passing a special iterator to the reduce function that may be used by the user code to read subsequent values associated with the same key.

4. *Partitioned Output*. As a consequence of the parallel reduction, the final output emitted by the reduce function is partitioned. Specifically, the output is split into R partitions: one per partition of the intermediate key space. A user-specified output format determines how to persist the output partitions; typically, output is written to a distributed file system such as the Google File System [7]. If an application cannot operate directly on a partitioned output data set, it must manually implement a final merging step to produce non-partitioned output, or use $R = 1$, which limits the potential parallelism. The

distributed MapReduce engine does accept partitioned input data, so if the output of one MapReduce program is to be passed as input to another, no extra step is required.

The architecture of Google's MapReduce engine has been re-used by other implementations. In particular, Hadoop—the most widely deployed open-source MapReduce engine—has a very similar architecture. It employs the exact same strategy of partitioning work into map tasks and reduce tasks, and fetching intermediate map output files on demand. Hadoop also includes a distributed file system called HDFS, which closely mirrors the design of the Google File System, and plays the same role as the common place to store the data sets involved in MapReduce computations. For convenience, we collectively refer to MapReduce engines that follow Google's original architecture as *traditional* MapReduce engines.

2.3.4 Semantics and Additional Hooks

The original Google implementation of MapReduce passes strings to and from all user-defined functions and leaves it to the user code to convert between strings and appropriate types. However, the map and reduce functions conceptually have associated type signatures:

map	$(k1, v1)$	$\rightarrow list(k2, v2)$
reduce	$(k2, list(v2))$	$\rightarrow list(k3, v3)$

In other words, input keys and values are drawn from a different domain than the intermediate keys and values, and output keys and values are drawn from a third separate domain. In practice, it may be useful to also use the intermediate keys as output keys, to preserve the grouping done by the MapReduce engine.

For deterministic results, the map and reduce functions are required to have certain additional properties that allow MapReduce programs to be automatically parallelized. Specifically, they cannot maintain hidden state across invocations, have semantically observable side effects, or rely on external input. In other words they must be *pure functions*. Imperative languages such as C++ and Java do not in general provide mechanisms to enforce the purity of a function, so in MapReduce programs implemented in those languages it is up to the programmer to abide by these restrictions. Failure to do so will lead to unpredictable and potentially incorrect results.

As a pure function, the map function may conceptually be evaluated in parallel for all input key/value pairs. Similarly, the reduce function may be evaluated in parallel for all unique intermediate keys. As such, a MapReduce program can in principle be executed with arbitrary degrees of parallelism, without modifying its code, subject only to the limitations posed by the number of input pairs and the number of unique intermediate keys. In practice, the overhead of this extremely fine-grained approach is likely to be excessive under most circumstances, and a more coarse-grained division of work is desirable. The parameters M and R determine this granularity in traditional engines. Hadoop's default configuration adjusts M according to the size of the input data, such that each map task processes 64 MB of the input.

In addition to the map and reduce functions, MapReduce engines typically provide several other programming hooks where user-defined code may be plugged in to customize a computation.

Input format The MapReduce programming model does not make particular assumptions about the on-disk storage format. In fact, the input is not required to reside on disk at all;

it could for example be retrieved as query results from a relational database, or simply be generated by a deterministic algorithm. Independence from the underlying storage system is cited by the MapReduce designers as a key advantage over parallel databases. [15] This flexibility is achieved by allowing custom *input formats* to be plugged in as a piece of user-defined code. The exact interface to this code may vary between engines, but the common goal of the input format is to provide key/value pairs to the map function, and (in a traditional engine) to provide automatic partitioning of the input, according to the desired number of map tasks.

In Hadoop, the default input format reads sequentially from a set of HDFS files, assigning separate files and byte ranges to each map task. The details of which files to read (typically a single input directory in HDFS), how to parse the data into key/value pairs, and the expected types of the keys and values, are all specified in the job's configuration. When passed to the map function, keys and values are instances of regular Java classes, as specified in the configuration.

Google's engine is implemented in C++, and delegates the responsibility for type safety to the user-defined functions. Keys and values are simple byte strings; the map function must interpret these as appropriate, for example by parsing them into integers or decoding them as unicode strings. The logic for retrieving input data and parsing it into key/value pairs may still be customized, as in Hadoop.

Output format There is typically a similar way to customize how the output emitted from the reduce function is collected, by plugging in a custom *output format*. Usually, the output should be persisted in the companion distributed file system. (HDFS, in the case of Hadoop). Traditional engines assume that the output from reduce tasks is persisted (or consumed) in a reliable manner, so once a reduce task has completed, it will never be re-executed. This differs from how map tasks are handled; even if a map task has completed, it may later need to be re-executed in order to regenerate intermediate data that has been lost due to a failed machine.

Partitioning function For many applications, the details of how the intermediate key space is partitioned are irrelevant; the only concern is to avoid excessive data skew, which would adversely impact load balancing. Engines therefore default to hash partitioning; the intermediate data is partitioned by hashing the intermediate keys into R separate buckets. Since each reduce task produces one partition of the final output, the partitioning of the intermediate key space also determines how the final output is partitioned. This can commonly be customized by plugging in a *partitioning function*, for example to use range partitioning, where each partition contains a given contiguous range of intermediate keys. Range partitioning is more prone to data skew, but can be useful in many scenarios, for example for sorting. If the aim is to produce a single sorted output file, this can be done by range partitioning the intermediate data by the sort key, and finally concatenating all output partitions.

Comparison function As noted, traditional MapReduce engines rely on sorting to correctly implement reduction and to provide a guarantee for the order in which keys are passed to the reduce function. For correct reduction, an arbitrary ordering of keys is sufficient, since the sorting is only used to group equal-keyed records. When the ordering guarantee is used to produce sorted output, applications may wish to employ a particular

ordering of keys. This is accommodated by a hook to specify a *comparison function*, which compares a pair of keys to determine if they are equal, or if not, which one is less than the other. Accordingly, the sorting algorithm must be comparison-based (excluding radix sort algorithms, for example) and all comparisons between keys are implemented by invoking the comparison function.

Combiner function If a distributed computation is commutative and associative, and performs a data reduction, it can benefit from an aggregation tree [37]. For example, the sum of a set of values can be aggregated from multiple partial sums; this is beneficial because the partial sums can be computed locally, close to where the data is stored. This reduces the amount of data that must be transferred over the communication network, as an instance of *upstream evaluation* [38].

Due to the general restrictions on the topology of the underlying dataflow graph, as described in the next section, traditional MapReduce engines do not support arbitrarily shaped multi-level aggregation trees. However, a mechanism for symmetric, two-level aggregation is provided through an optionally specified *combiner function*. The combiner function has the same signature as the reduce function, and partially aggregates intermediate key/value pairs from a single map task, before the partially aggregated data is transferred to the reduce tasks.

Commonly, the same function is used both as the combiner and as the reduce function. The word count aggregation example given above could use its reduce function as a combiner too, in order to compute partial word counts. To see why this would be beneficial, consider a very common word such as "the", which would occur very frequently in the input data. With no combiner, every map task would emit a high number of intermediate pairs keyed by the word "the"—all of which would have to be stored temporarily on disk and subsequently transferred over the network to a reduce task for aggregation. With a combiner, all of these intermediate pairs would be aggregated into a single pair, reflecting the total number of occurrences in that particular input partition, *before* the intermediate data was stored and transferred to a reduce task.

2.3.5 The MapReduce Dataflow Graph

To summarize the architecture of traditional MapReduce engines, they decompose computations into a number of separate tasks, each of which has certain input requirements and produces some output. When considering these data dependencies, the tasks form a dataflow graph with a specific shape. Figure 2.5 shows an example with three map tasks and two reduce tasks ($M = 3$, $R = 2$). In total there are $M \times R$ intermediate files, stored on local disks throughout the cluster, and labeled in the figure as *t-1-1*, *t-1-2*, etc. If sorting is viewed as an integral part of the reduce tasks, the map and reduce tasks form a complete bipartite graph in which all reduce tasks depend on data from all map tasks. This reflects the two-phase nature of MapReduce, which dictates that all map tasks must finish before any reduce task may start. Intermediate files may be prefetched and partially sorted, but the reduce tasks may not start invoking the reduce function before the output from all map tasks is available.

Viewed as dataflow engines, traditional MapReduce engines are essentially specialized implementations in which the dataflow graphs are restricted to the specific bipartite shape described above. The exact number of map and reduce tasks may be varied, and the behavior of

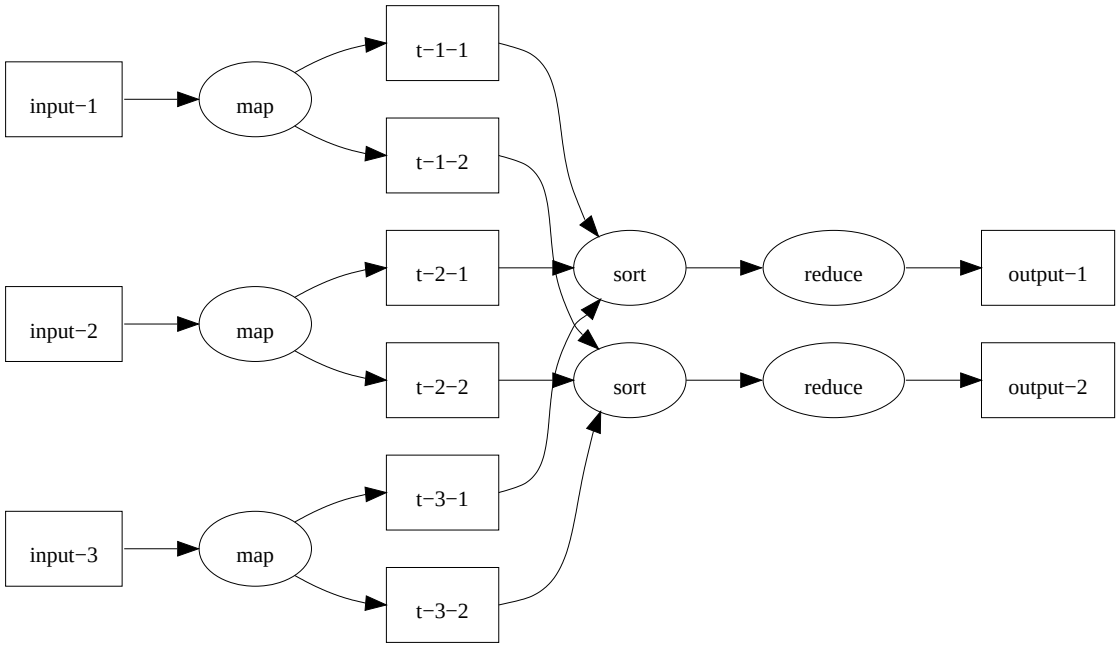


Figure 2.5. A MapReduce dataflow graph with M=3 and R=2.

these tasks is customizable through the map and reduce functions, but the overall communication pattern remains fixed. While sacrificing some generality, this restriction does simplify various parts of the system, such as the data structures maintained by the task scheduler, since there is no need to explicitly represent the edges of the graph.

2.4 Map-Reduce-Merge

The relational data model [14] has underpinned database technology for several decades, and remains a popular way to model and think about data, even when other tools are employed for data processing. In relational terms, a MapReduce program implements a selection and a projection operation in the map phase, followed by a group-by aggregation in the reduce phase. The selection and projection is implemented by the user-supplied map function, the execution engine implements the grouping, and the user-supplied reduce function implements the aggregation. However, computations involving more complex relational algebra are cumbersome to express using MapReduce. Due to the fixed topology of the underlying dataflow graphs, certain queries can only be executed as a sequence of collaborating MapReduce programs. In addition, relational joins are common when processing multiple related heterogeneous data sets, but joining is not efficiently supported by the MapReduce model.

Within the constraints of the MapReduce model, a simple sort-merge equi-join algorithm can be implemented using a map function that emits relational tuples keyed by the join attribute, and a reduce function that merges all equal-keyed tuples, but this approach has some potential problems. Conceptually, there are two input relations, and tuples from one input relation should be paired with tuples from the other. However, the reduce function can only access a single collection of tuples: they all have the same join attribute (since it was used as the intermediate key), but they may originate from either of the input relations. Consequently, the reduce function must read all of the tuples into memory in order to do the pairing correctly.

For example, an inner join should discard tuples whose join attribute only occurs in one of the input relations, but the reduce function cannot safely do so until all of its input tuples have been examined.

Map-Reduce-Merge [39] recognized the shortcomings of MapReduce with regard to relational joins, and proposed an extension of the programming model. The extended model and implementation allows two input data sets, and adds a third *merge phase* after the map and reduce phases. The two input data sets are first processed by two separate sets of map and reduce tasks, and in the third phase the output from the two sets of reduce tasks is passed to a final set of merge tasks, which apply a user-defined *merger* function. The merger defines how to combine two intermediate key/value pairs (one derived from each input data set) and produce the final output. Despite its name, the Map-Reduce-Merge model does not necessarily rely on merging in the third phase—the iteration logic that determines the order in which intermediate pairs are passed to the merger can be customized. This allows the implementation of different join algorithms such as nested-loop joins, hash joins and sort-merge joins, as well as other binary matching algorithms such as set difference. The flow of data between reduce and merge tasks can also be customized through a separate *partition selector* function.

The downside of the extensions provided by Map-Reduce-Merge is that they complicate both the programming interface and the implementation of the MapReduce engine in order to support the specific use case of efficient relational joins. At the same time, the topology of the dataflow graph is restricted in similar ways as by MapReduce, and complex relational queries must still be executed as a sequence of Map-Reduce-Merge programs. As such, Map-Reduce-Merge sacrifices some of the simplicity for which MapReduce is popular, without fully addressing its weaknesses when compared to a generic dataflow engine.

2.5 Workflow Composition and High-level Languages

Even if joins are not a critical requirement, a MapReduce computation is fundamentally restricted in that the data to be processed can be repartitioned only once, as it is output by the map phase and persisted as an intermediate data set. As a consequence, certain computations can not easily be implemented as a single MapReduce computation. To compensate for this, more complex computations may be composed as a collection of related MapReduce programs, arranged in a pipeline, where the output of the first program serves as the input to the next, or in a more complicated manner. For example, Google rewrote the indexing system that produces the index structures for their web search service from using ad hoc distributed passes over the input data to using a sequence of 5 to 10 MapReduce programs [6].

Many higher-level abstractions are also designed specifically to ease the composition of multi-pass MapReduce computations, and may diverge from the traditional MapReduce programming model to provide additional operations beyond mapping and reducing, such as merging of multiple data sets. For example, Sawzall [10] is an interpreted language for data analysis that is specifically designed to be integrated with MapReduce as an underlying execution engine. A Sawzall program conceptually executes in parallel for every record in a data set, and may produce output by emitting records to any number of declared *aggregators*. When executing an analysis using Sawzall, a generic MapReduce program interprets the Sawzall program once for each record during the map phase, and then aggregates the various records that were emitted during the reduce phase. In practice, Sawzall collapses and consolidates what would

usually be expressed as a number of related MapReduce programs executing on the same data set into one entwined computation that just makes a single pass over the input data set.

Other systems provide libraries or embedded languages to compose workflows involving multiple related data sets, including operations such as merging, to enable a more natural way to express relational joins. Cascading [40], FlumeJava [4], Hive [41], and Pig [16] are examples of systems that directly compile workflows into collections of MapReduce programs. Cascading and FlumeJava offer a programmatic interface to workflow composition, similar in nature to DryadLINQ [42], which implements a set of programming abstractions for compiling Dryad graphs. This approach may be viewed as embedding a new language into a general-purpose language, in the form of a library. Hive and Pig define new domain-specific languages—HiveQL and Pig Latin, respectively—and provide compilers and interpreters to execute programs expressed in those languages as a series of MapReduce computations on a Hadoop deployment. Another similar language is SCOPE [43]; these languages all share a similar set of relational data operators, and emphasize extensibility by providing language features for seamless integration of user-defined code implemented in a general-purpose language. In a somewhat similar vein, BOOM [44, 45] is a data-centric, declarative language specifically intended to develop distributed systems running in the cloud, and has been used to reimplement the Hadoop runtime.

The trend of creating new, declarative languages extends to other high-level abstractions that blur the distinction between rigid, domain-specific query languages, exemplified by SQL as the established standard for relational query processing, and MapReduce-style computations, structured around opaque user-defined functions. For example, some of the recent abstractions developed by Google target similar application areas as MapReduce, while offering quite different programming interfaces. Dremel [9] supports a SQL variant for large-scale query processing or aggregation of *nested* data, while Pregel [8] implements a vertex-centric graph processing language.

2.6 Alternative and Hybrid Architectures

Despite the above refinements and specializations, MapReduce still inhabits a key position in the software stack that Google developed to harness their gigantic computing clusters. While the Google File System [7] provides the abstraction of a reliable and shared *storage*, MapReduce offers a simple, functional interface for distributed *processing*. Once applications are cast into the MapReduce framework, they all enjoy the scalability and fault tolerance inherent in its execution engine.

This separation of concerns may be a sound engineering practice, but by separating storage and processing into separate layers, an apparent risk is that performance may suffer from the extra layer of indirection. An alternative approach is to fuse the distributed file system and processing engine into a single, tightly coupled component. This philosophy is characteristic of parallel databases, and is also embraced by others, for example in the twin systems Sector and Sphere [46]. These closely integrate the mechanisms for data processing with the storage layer, by offering the capability of evaluating user-defined functions locally on storage nodes. Sector and Sphere may leverage this capability to execute MapReduce computations, but have no built-in features for automatic partitioning of data, and are designed for geographically distributed data centers connected through high-speed networks.

Traditional MapReduce engines are designed for data-intensive computations and target shared-nothing environments with locally attached hard disks. As such, data should ideally be read in bulk and using a sequential access pattern, to accommodate the performance characteristics of hard disks, where seeking between different locations on disk is a time-consuming, mechanical process and adversely affects the rate of data transfer. In alternative hardware environments, such as a shared-memory multi-core architecture, the underlying storage may support constant-time access. There are MapReduce implementations that target such environments, such as Phoenix [47, 48], which is a shared-memory implementation of MapReduce for multi-core environments, and MARS [49, 50], which accelerates MapReduce using graphics processors for coprocessing. In a similar vein, the Nornir run-time [51] offers an efficient multi-core implementation of Kahn process networks [52], allowing the structuring of CPU-intensive computations as arbitrary communication graphs, which may contain cycles.

In our work, we focus on the traditional application areas of MapReduce, i.e. large-scale data-intensive analysis. We therefore target the traditional shared-nothing hardware environment. These alternative implementations are still relevant, since they underline the generality of MapReduce as a programming model for parallel computations. Our experimental evaluation in Chapter 5 also reveals that multi-core performance may be an important concern even for traditional workloads.

Twister [53] is a MapReduce engine that extends the classic MapReduce programming model with support for iterative jobs. Like Cogset, Twister explores alternative algorithms for data shuffling; in the case of Twister, all communication and data transfer is factored out as a separate and substitutable publish-subscribe messaging infrastructure. This may be an interesting platform for experimentation, and effectively breaks the MapReduce engine into three loosely coupled components: storage, communication, and processing. Such a loosely coupled architecture is flexible, but most systems opt for a closer integration of these three concerns, in the interests of performance.

As noted in Section 2.3.4, traditional MapReduce engines are loosely coupled to the underlying storage system, and may retrieve records from a variety of sources using customized input formats. One interesting line of research is to integrate a distributed MapReduce engine with existing single-node relational databases, using a number of database instances as the storage layer and the MapReduce engine as a centralized coordinator that distributes queries and merges results. Such hybrid approaches essentially *federate* multiple single-node databases into a single distributed database, using MapReduce as a framework for fault tolerance and load balancing. Two concrete examples are HadoopDB [54] and Osprey [55], which both integrate a number of single-node instances of PostgreSQL, a powerful open source database system [56]. HadoopDB is based on Hadoop and extends Hive [41], to draw on its support for compiling SQL queries into Hadoop execution plans. HadoopDB modifies the execution plans so that certain sub-queries are executed in parallel by the PostgreSQL databases; the results are then merged by Hadoop.

Osprey has a similar strategy for query execution, but implements the central coordinator from scratch, rather than building on Hadoop. The coordinator compiles SQL queries into a number of independent sub-queries to be executed in parallel on different nodes. Failed sub-queries are re-executed on other nodes with replicas of the same data, mimicking the general approach to fault tolerance and load balancing taken by traditional MapReduce engines, while drawing on the capabilities of PostgreSQL for efficient query execution on individual nodes.

2.7 Summary

This chapter surveyed related work, focusing on the MapReduce programming model and possible ways to implement a MapReduce engine. As a programming model, MapReduce is distinguished by delegating the majority of functionality to user-defined functions, whose internals are opaque to the execution engine. A single-process MapReduce engine is simple to implement, as evidenced by our example implementation in Python. The original *distributed* MapReduce engine developed by Google is founded on principles from dataflow engines, but restricts the dataflow graph to a certain topology. In return, traditional MapReduce engines are more resilient to failures than the dataflow engines under the hood of parallel databases. Subsequent work has layered additional abstractions on top of MapReduce in various ways, either to facilitate computations spanning multiple MapReduce passes, or to enable expression of more complex queries. Many efforts aim to combine the robustness of traditional MapReduce engines with SQL-based query interfaces and/or index-based query execution, blurring the distinction between MapReduce and parallel databases.

Chapter 3

Design and Architecture of Cogset

The thesis of this dissertation posits that a high-performance MapReduce engine can be constructed with static routing as a core mechanism. In this and the following chapter we present *Cogset*, a distributed system that we designed and implemented to investigate this conjecture. Here, we describe the overall architecture and design of Cogset, and in Chapter 4 we dive into the details of its implementation.

Since Cogset is intended to be a core engine on which multiple higher-level abstractions may be constructed (see our requirement for extensibility in Section 1.4), our overall focus is on maintaining simplicity, generality, and performance. Additional complexity may be added in layers on top of Cogset, and should not be imposed in cases where a less restrictive interface is sufficient. We refer to all programs that communicate with Cogset through its public interfaces as *clients*, whether they be actual end-user applications or other services offering higher-level abstractions. Our ambition to let Cogset serve as the foundation for a stack of software abstractions is reflected in many aspects of our design, where we seek to identify the common denominators for potential higher-level abstractions.

The design of Cogset required some fundamental initial choices that have far-reaching (and in some cases subtle) consequences for the whole system, affecting both its interfaces, semantics and non-functional aspects. We start by describing these key design choices and their motivation, establishing a broad overview of the Cogset architecture and its properties. Subsequently, we proceed to discuss the central programming abstractions in Cogset, and the core mechanisms that enable reliable storage and fault-tolerant, load-balanced processing of data.

3.1 Key Design Choices

Functionally, Cogset is a system for storage and processing of data sets, which are comprised of records. Cogset stores a collection of named data sets and provides interfaces for parallel data processing. For efficiency, and as a consequence of static routing, the mechanisms for storage and processing are more tightly coupled than in a traditional MapReduce engine. Even though Cogset can run on a single node, our design is aimed at efficient distributed deployments.

In the following, we discuss five important and major choices in the Cogset design. The first two design choices are directly motivated by our decision to employ static routing. As such, they may be viewed as a more elaborate statement of our one overarching, guiding design choice. They detail our concrete understanding and interpretation of the concept of static rout-

ing: data placement should be predetermined by a static configuration, and new records should be transferred directly to the appropriate storage nodes.

Predictable data placement Commonly, storage systems effectively assign individual data partitions to arbitrary storage nodes, either because the system is unaware of the partitioning (i.e. it is done externally, outside of the storage system), or because other concerns dictate the data placement policy. For example, in the case of a traditional MapReduce computation generating data to be stored in the Google File System (GFS) [7], each reduce task writes one output partition to GFS. The actual storage nodes for each partition are selected by GFS based on concerns such as network topology, favoring nodes close to the one hosting the reduce task. The contents and context of the partition (e.g., partition number 3 out of 7) do not factor into the decision. As such, it is difficult to ensure co-location of related records that originate from different MapReduce computations, and applications such as relational joins may suffer from poor data locality.

Cogset takes a different approach and allows clients to explicitly influence data locality. Specifically, clients may ensure that records are co-located by assigning them to the same partition number. All data sets are partitioned into the same number of partitions, and the corresponding partition numbers (of all data sets) are always co-located. This restricts Cogset with regard to data placement policies, but paves the way for highly efficient relational joins, and benefits all computations that are able to capitalize on co-located input data.

Direct routing of data As noted in Section 1.3, traditional MapReduce engines temporarily store the intermediate output data produced by map tasks on local disks, to be retrieved on demand once the corresponding reduce tasks are scheduled to execute. This is both a central part of the traditional fault tolerance mechanism, and an approach born from necessity, given the overall design: since reduce tasks can be scheduled to execute on any node, the map output data cannot be routed directly to the appropriate storage nodes.

Given our choice to predetermine data placement via a static configuration, a natural follow-up is to route data directly to its destination, rather than performing additional I/O to store an extra temporary copy. We view this as an integral part of static routing, so Cogset does adopt direct routing of data, avoiding temporary storage. New records are transferred over the network directly to the appropriate storage nodes, where they are immediately persisted. As a consequence, we cannot employ the traditional MapReduce mechanisms for fault tolerance, where partial output from failed tasks can trivially be discarded. Instead, we designed a new mechanism, *distribution sessions*, which are a form of simplified transactions, allowing undesired side effects resulting from failures to be ignored.

Beyond these two design choices, which embody our central choice of using static routing, we made some additional and largely orthogonal choices. These design choices are motivated by our supplementary requirements for efficiency, generality and extensibility.

Explicit partitioning When data sets are too big to fit on the local storage of a single node, they must be partitioned in order to enable parallel processing on multiple nodes. Given this requirement, we chose to expose partitions as a fundamental concept in Cogset,

rather than provide abstractions that hide the inherently partitioned nature of our data. When generating a new data set, or processing an existing one, all new records are assigned to a particular partition number. As noted, clients are aware of and may explicitly influence this partitioning, for example to co-locate records.

Higher-level abstractions built on top of Cogset still have the option of implicitly partitioning data, if this is desirable and feasible for the abstraction in question. An abstraction using implicit partitioning would by nature also be tasked with any data locality concerns. Explicit partitioning is therefore appropriate at this abstraction level, as the most general common denominator.

Visitor-based data access In traditional storage systems, data is streamed to clients on demand. Whenever a client wishes to process a data set, the entire data set must be transferred over the network. To see how this can be inefficient, consider a simple example where a client needs to scan through a large set of data items looking for a particular item, to determine whether or not it exists. Streaming all of the data over the network before scanning wastes bandwidth compared to the alternative approach of simply scanning the data locally on the storage node, and transferring a simple yes/no answer. Furthermore, a parallel scan of the data would have to be orchestrated entirely by the client; a non-trivial task that would likely involve multiple cooperating nodes retrieving and scanning separate subsets of the data.

In contrast, Cogset encourages all data access to be made locally on the storage nodes. Rather than streaming large data sets to the clients, the situation is reversed, and clients transfer small pieces of code to the storage nodes, as an instance of *remote evaluation* [57]. Specifically, a *visitor* function is transferred and evaluated in parallel on each storage node. When evaluated, the visitor may thus access the data stored locally on each storage node, without consuming any network bandwidth. Parallel computations are also easy to express, since the visitor will be evaluated in parallel for all partitions of the data.

Replication In a general sense, replication serves two purposes. It provides *redundancy*, which enables fault tolerance, and it increases *availability*, which facilitates load balancing. Since both fault tolerance and load balancing are important requirements, we decided to make the replication mechanism an integral part of Cogset. Replication of data may appear to be a self-evident requirement in a system for reliable storage, but there was still the option of making Cogset an *unreliable* storage system, and implementing replication in higher-level abstractions only. However, we rejected this option, because replication is so tightly coupled to fault tolerance and data placement. Instead, we exploit our awareness of and control over data replication to provide a dynamic load balancing mechanism, as detailed further in the next two sections.

3.2 Data Partitioning

Traditionally, MapReduce engines store data in a block-based distributed file system, breaking each data set into a variable number of fixed-size partitions, or blocks. In contrast, we designed Cogset to break each data set into a fixed number of variable-sized partitions. When the number of partitions is fixed, a static configuration can assign partitions to nodes. In block-based file systems such as GFS and HDFS, the mapping that assigns blocks to nodes must be

dynamic, and is maintained by a centralized master. As a consequence, the centralized master must be consulted before reading or writing data blocks. In Cogset, a globally known static configuration determines data placement. In practice, this is implemented through a simple configuration file that is read upon start-up, and assumed to be identical for all nodes. In our deployments we ensure this in a convenient way by keeping the configuration file in a common, NFS-mounted location accessible to all nodes.

Cogset's configuration only specifies where each partition of a data set should be stored. It does not specify how individual records map to partitions; this may be influenced by user-defined hooks, which we describe in more detail in Section 4.3. Applications should preferably be oblivious of the concrete details concerning data placement. On the other hand, they may require co-location of related records in different data sets, for example to perform a relational join. Cogset accommodates this requirement by enforcing a particular constraint: corresponding partitions of different data sets are *always* co-located. As noted, all data sets are split into a fixed number of partitions; the partitions are enumerated, and equal partition numbers are always assigned to the same storage node. Applications thus ensure co-location of relevant data by assigning related records to the same partition number. Given the co-location constraint, data placement is configured simply by mapping partition numbers to nodes; there is no specific configuration for individual data sets.

Given the static configuration of data placement, Cogset nodes or clients can distribute data directly to the appropriate storage nodes without involving any centralized master, avoiding a potential bottleneck. Direct routing of data also allows multiple concurrent data streams to be merged by a storage node before they are written to disk, interleaving records produced by different nodes. With the traditional algorithm for dynamic routing of data, each map task must temporarily store all produced records locally; they are later fetched and transferred to other nodes, subject to the unpredictable scheduling of reduce tasks. Overall, this leads to a more scattered I/O access pattern, which may be a cause for inefficiency by causing excessive disk seek times [15].

3.3 Distribution Sessions

One concern that must be addressed when static routing is employed for distribution of data is how to handle failures gracefully, without restarting the whole computation. In the traditional MapReduce architecture, all map tasks write their output to separate intermediate files. In the event of task failures, there is no need for a complicated rollback mechanism to undo side effects; the respective intermediate files are simply discarded. In contrast, Cogset by design transfers data directly to the appropriate nodes and streams it continuously to disk, potentially merging data streams from multiple producers. If a process fails while producing data, and is subsequently re-executed, the produced data could be duplicated on disk.

To maintain a consistent view of data, even in the event of failures, we therefore designed a new abstraction for Cogset. This abstraction—a *distribution session*—may be viewed as a simple form of transaction, in which the only allowed operation is to append data to a file. To add records to a data set, a client process must initiate a new distribution session, creating a unique 64-bit identifier for the session. This *distribution session identifier* (DSI) is associated with all records produced as part of the session, and stored along with the records when they are persisted on disk. Clients may commit a distribution session using a two-phase commit protocol coordinated by the client. When committing a distribution session, each Cogset node

syncs the affected local files, flushing the data to disk, before updating a log of committed DSIs. Optimizing for the common case where failures are rare, there is no rollback mechanism in the event of failures or aborted sessions. Instead, the DSI of each record is inspected when reading data from disk, and any uncommitted records are skipped. If desired, long-lived data sets can be cleansed of uncommitted records by copying the data set, reclaiming any wasted disk space. As an implementation detail, the storage overhead associated with DSIs is reduced by batching records into *pages*, as described in section Section 4.5.

3.4 Replica Placement and Neighborship Relation

While data must be *partitioned* for distributed processing, it must also be *replicated* for reliable storage. In Cogset, a partition is also the unit of replication. The static configuration that maps partitions to nodes may therefore specify the replication strategy simply by assigning partitions to more than one node. When a record is distributed, it is sent to *all* storage nodes hosting the record's partition. Cogset does allow replica placement to be configured arbitrarily, but is designed with a particular default strategy for replica placement in mind, described next.

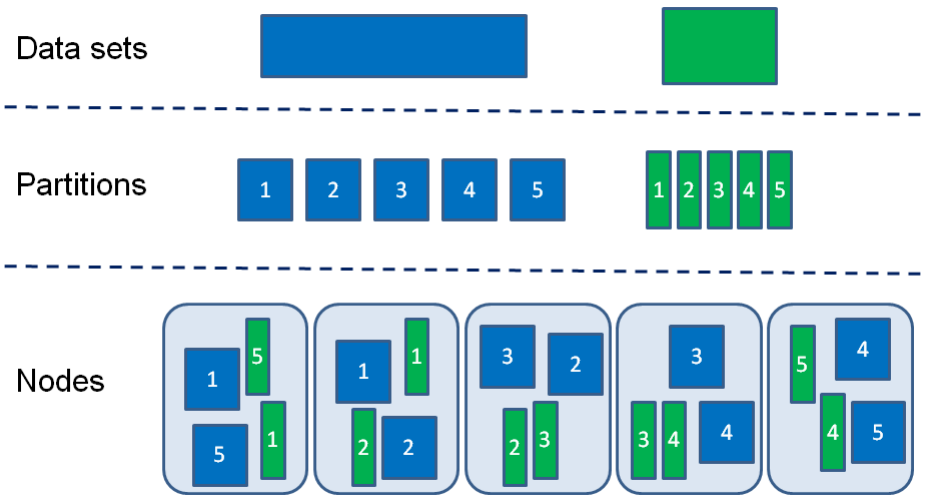


Figure 3.1. Data set partitioning, using chained declustering for replica placement.

Figure 3.1 shows a concrete example to illustrate how data sets are partitioned in Cogset, using its default configuration. The example features two data sets, color-coded as blue and green, respectively. For ease of exposition, the example breaks each data set into 5 partitions; a typical configuration would specify a higher number of partitions relative to the number of nodes, for more fine-grained load balancing. Using *chained declustering* [58], partitions are assigned to nodes in round-robin fashion, storing a replica of partition number i on node $i \bmod N, i+1 \bmod N, ..., i+R-1 \bmod N$ where N is the number of nodes and R is the replication degree. In this example, partitions are replicated with replication degree 2, which is the default. The salient aspect of chained declustering is that it imposes a logical structure where nodes form a chain, with each pair of neighbors in the chain having some replicas in common. We formalize this notion through the following definition:

Definition 1. *Two nodes are **neighbors** if and only if they have replicated partitions in common.*

Note that the above *neighborship relation* is defined independently of the actual replica placement strategy. Using the default configuration, each partition is replicated on two neighboring nodes, and the nodes form a circular chain of neighbors.

3.5 Traversals

So far, we have described the parts of Cogset's design that govern how data is partitioned, distributed, stored consistently, and replicated. This sets the stage for the most central aspect of Cogset's design—the fundamental and generic mechanism for processing data in a parallel, fault-tolerant and load-balanced manner: *traversals*.

In a traditional MapReduce architecture where the distributed file system is decoupled from the processing engine, data is generally streamed from a storage node to a processing node. In contrast, Cogset integrates a core mechanism for parallel processing into the storage nodes, encouraging all data access to be made locally. Instead of streaming large data sets to the clients, clients transfer small pieces of code to the storage nodes.

Specifically, a *visitor* function is transferred and evaluated in parallel on each Cogset node; we refer to this parallel evaluation as a *traversal*. A visitor may access the data stored locally on each node, without consuming any network bandwidth. Over the course of a traversal, a visitor may read multiple data sets, and output records to any number of new or existing data sets. This flexibility allows complex algorithms such as multi-way joins to be implemented as a single traversal. Unlike traditional MapReduce jobs, traversals are not tied to the processing of one specific data set; nor is there any pre-declared list of data sets that will be accessed during a traversal. It is entirely possible for a traversal to read some records from one data set, and depending on the results of that inspection, decide which other data sets to access, if any.

The aim of a traversal is to invoke the user-defined visitor function exactly once for each partition number. When invoking the visitor function, Cogset provides an object that offers access to *all* data sets of a particular partition number. Due to the data placement constraints, these partitions are all co-located locally on the node where the function is invoked. The visitor function is opaque to Cogset, so its side effects are unknown. However, some of them can be observed by Cogset; notably, to read or add any records to new or existing data sets, the visitor must go through Cogset's APIs. Given the black box nature of visitor functions, we refer generically to the invocation of a visitor function simply as "processing a partition". The specific programming interfaces involved in traversals are detailed in Chapter 4; here we focus on the high-level traversal algorithm.

Cogset's traversal algorithm diverges from the traditional MapReduce architecture, which is based on the master-worker paradigm, where a central master monitors and coordinates the full set of workers and makes all scheduling decisions. With Cogset, the client process fills a similar role as the central coordinator, sending the control messages for initiating and finalizing a traversal, and receiving progress messages from the Cogset nodes as the traversal proceeds. However, the scheduling algorithm is fully distributed and symmetric. Specifically, every Cogset node executes the same algorithm during a traversal, whose steps are as follows.

1. An initiation message is received from the client process. The message contains a unique identifier for the traversal, to distinguish between separate and potentially concurrent traversals, a (serialized) visitor function to be employed for the traversal, and a set of previously committed DSIs that defines the traversal's view of its input data. It may also specify a subset of partitions to process, defaulting to the full set of partitions.

2. The Cogset node spawns a dedicated child process for the traversal. This is a practical step to isolate errors caused by the user-defined visitor functions, preventing such errors from propagating and affecting concurrent traversals. The visitor function is sent to the child process.

3. A set P of partitions is initialized to contain all partitions that are hosted by the node.

4. A set N of *neighbors* is initialized to contain all other nodes that also host replicas of partitions in P.

5. One partition in P is selected for processing, and communicated to the child process, which invokes the user-defined visitor function to initiate processing of the selected partition.

6. When the child process reports that the processing of a partition is complete, the Cogset node removes the partition from P, notifies its neighbors N, and reports progress to the client. The progress message also includes any new DSIs that have been created, in the common case of visitors that generate new output data.

7. Unless P is empty, steps 5-7 are repeated.

This high-level description omits some important details concerning how nodes monitor and coordinate with their neighbors. Concurrently with the above steps, each node regularly exchanges status messages with its neighbors N, both to detect failed neighbors and in order to accurately update the set of remaining partitions P. To avoid the race condition where two neighbors simultaneously decide to process the same partition, the implementation of step 5 includes one round of status updates, flagged to generate immediate responses, minimizing latency. If a race is detected, node identifiers are used as tie breakers and the losing nodes make a new selection.

3.6 Load Balancing

Given the static policy for data placement, and general strategy of accessing data locally, it may appear that Cogset's capabilities for load balancing would be limited. In reality, the fact that all data is replicated can be exploited to effectively transfer load between neighboring nodes. During a traversal, all partitions must be processed, but every partition is replicated on multiple nodes, and can be processed by either of those nodes. The crucial step in the traversal algorithm is step 5, where a Cogset node selects which partition to process next, effectively implementing a distributed scheduling algorithm.

Using simulations, we have evaluated multiple such scheduling policies, and arrived at the following default. To select a partition, a node estimates the total amount of work remaining for itself, and for each of its neighbors. It then selects the node with most work remaining, and determines which of the remaining partitions hosted by that node has the highest estimated processing cost; this partition is selected as the next one to be processed. To obtain the estimated processing costs, a separate user-defined hook is invoked for each partition, as described in Section 4.10. Our simulations using this scheduling policy indicate that Cogset's traversal algorithm is sufficiently flexible to balance load effectively both in heterogeneous environments and in the presence of data skew.

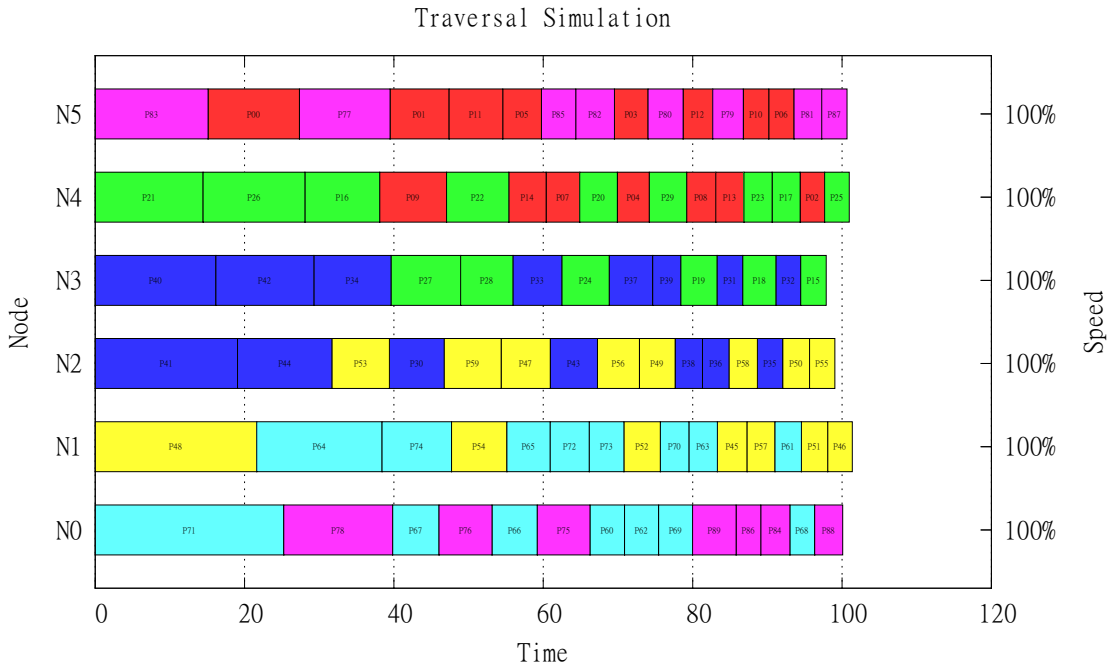


Figure 3.2. Example traversal. Large partitions are processed first, to balance the load evenly between all nodes.

For an intuitive illustration of how this works, consider Figure 3.2, which visualizes a simulated traversal using a Gantt chart. The figure shows a traversal involving 6 nodes, hosting a total of 60 partitions, replicated for a total of 20 partition replicas per node. For each node, the timeline shows the order in which partitions are processed, and the time taken to process each partition. Partitions are color-coded to illustrate their locations; N0 hosts all pink and cyan partitions, N1 hosts all cyan and yellow partitions, etc. The timeline is normalized such that all nodes would ideally finish at time 100, if the load were perfectly balanced. In the example, all nodes do finish the traversal very close to that time, even though there is significant data skew. By processing large partitions first, the total load is balanced effectively, while respecting the restrictions imposed by the replica placement. For example, both N2 and N3 start off by processing different blue partitions, which dominate in size; the neighboring nodes N1 and N4 compensate by processing yellow and green partitions.

A second example, in Figure 3.3, shows a scenario where one node (N3) fails mid-way through the traversal. The failed node hosts blue and green partitions, and after its failure is detected, the neighboring nodes N2 and N4 switch to almost exclusively processing blue and green partitions, respectively, picking up the extra load. In turn, their neighbors compensate by processing more yellow and red partitions, in a ripple effect that effectively balances out the extra load among all remaining nodes. Once again, all nodes complete the traversal in close succession, this time at around time 110 due to the extra load.

In summary, the key insight behind Cogset's load balancing algorithm is that *neighbors may off-load each other*. This motivates our choice of chained declustering as the default strategy for replica placement, as described in Section 3.4. The neighborship relation is determined by replica placement, and when using chained declustering, the relation connects *all nodes* into a

circular chain. This allows very effective load balancing by propagating excess load along the chain of neighboring nodes.

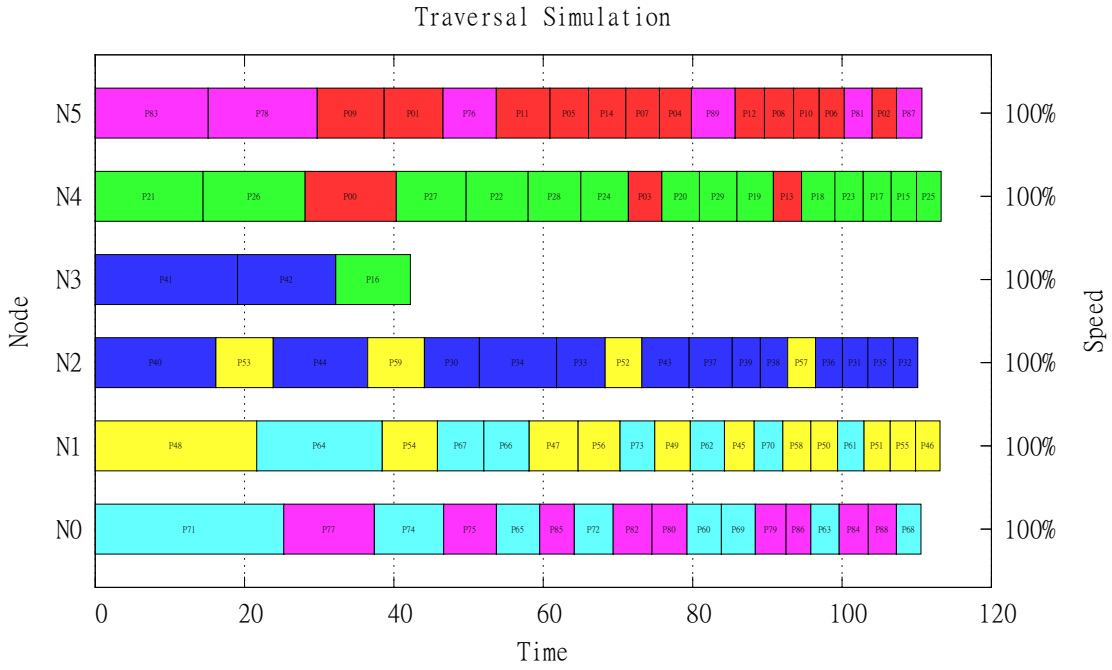


Figure 3.3. Traversal where one node (N3) fails mid-way through the traversal. Neighboring nodes pick up the extra load (blue and green partitions).

The use of chained declustering as a foundation for load balancing was also explored in the recent Osprey system [55]. Osprey is a federated parallel database that decomposes queries into sub-queries to be executed in parallel on a collection of regular single-node PostgreSQL instances. Failed sub-queries are re-executed on another instance with access to a replica of the same data. The core algorithm for balancing load between instances is similar to what Cogset incorporates into its traversal algorithm, although our algorithm is fully distributed in a symmetric manner, while Osprey uses a central coordinator for scheduling. Our algorithm was independently designed but appears to be based on the same intuitions; the concurrent work on Osprey confirms that this is a sound approach to load balancing.

3.7 Fault Tolerance

To elaborate on fault tolerance, recall that the goal of a traversal is to process every partition; in other words, to invoke the visitor function exactly once for each partition number. In theory, this implies that a visitor function should be invoked as an atomic operation. In practice, those semantics must be approximated, due to the potential for failures. The invocation of a visitor function may take a long time (more than a minute for some partitions in some of our benchmarks), and a node may well fail while invoking the function. For example, there may be software bugs in the visitor function, which is defined by the user code.

Cogset handles such failures by re-execution: the partition in question is simply rescheduled for processing, potentially on a neighboring node. Since a separate child process is used to

invoke the visitor function, software bugs and transient errors are contained and prevented from crashing the main process of the Cogset node. If the main process fails while a partition is being processed, the child process will also terminate, as if failing.

Re-execution is a straightforward strategy for fault tolerance, but to make it work, Cogset must be able to ignore the side effects of partially processing a partition. This is enabled by the *distribution session* mechanism. When a visitor function is invoked, it may distribute new records to other nodes, which are immediately persisted on disk, even though the visitor function may subsequently fail and be re-executed. Distribution sessions ensure that such partial output can be discounted, since the records in question would be uncommitted and ignored by future reads.

Traversals may be read-only (or more generally, idempotent), but typically do produce some output records, which are added to one or more data sets. If and when to commit these records is a matter of client policy; as the traversal progresses, all new DSIs are reported back to the client. If the client maintains a log of completed partitions, it may commit DSIs on a per-partition basis, effectively check-pointing the computation after each completed partition. Such a client can recover from a failure by starting a new traversal using the same visitor function, restricted to the set of partitions whose output remains to be committed. Alternatively, all DSIs may be committed together when the traversal is complete, in which case the client can be stateless, in exchange for making a client failure (analogous to a master failure in traditional MapReduce engines) more costly.

A more common failure scenario is the case where a Cogset node fails, as in Figure 3.3. While our load-balancing algorithm is able to compensate for failed nodes, a failed node will reduce the replication degree of a number of partitions. This must be remedied by re-integrating a new node to host the affected partitions, or by creating new replicas of the partitions on the remaining nodes. To reintegrate a failed node, the partition replicas stored on that node must be synchronized with the replicas on its neighboring nodes, to incorporate any changes that were committed during its downtime. Reconfiguring the placement of partition replicas is simple, but can entail copying of significant data volumes between nodes. Currently, these operations can only be performed off-line, while no traversals are in progress. Potential future work includes devising algorithms for re-integrating nodes into a running system with traversals in progress, and similarly for on-line reconfiguration of replica placement.

3.8 In-situ Data Processing

In Section 1.3, we listed the potential for in-situ data processing as one of the advantages of MapReduce, setting it apart from parallel databases. By implementing a custom input format, as described in Section 2.3.4, pre-existing external files can be used as input to a MapReduce computation, without any initial reformatting. All that is required to integrate an external file is the customized logic for parsing it into a sequence of key/value pairs.

Internally, Cogset batches records into *pages* before they are transferred over the network or stored on disk, as described in more detail in Section 4.5. Pages are self-contained and self-extracting data structures; their internal representation may vary and is opaque to Cogset. A Cogset node stores data as a set of regular files in the local file system, using one file per partition per data set. Files created by Cogset invariably contain a sequence of pages. Despite this apparent rigidity, Cogset still retains the capability for in-situ processing of records.

This is accomplished by providing a very cheap mechanism for importing, or more accurately integrating, existing data files into the storage managed by a Cogset node.

Specifically, a set of records stored in a local file can be added—as a constant-time bulk operation—to a given partition of a data set using a simple command-line tool. The arguments to the tool are the local file to integrate, a specification of the format of that file, and the data set and partition number to which the external records should be added. The tool works by creating an *indirection page* which is appended to the appropriate partition file in the regular manner. When unpacking itself, the indirection page will not decode its own payload (which is empty); instead, it will read the contents of the external file, parsing records as originally specified on the command-line, for example by treating each line of plain text as one record.

A similar approach can be used to integrate other external data sources into Cogset, such as relational database queries, although we have not created specific command-line tools for that. Just like a traditional input format, an indirection page can contain arbitrary logic for providing records. For example, it could generate a very large but deterministic sequence of records based on a pseudo-random number generator initialized with a specific seed—a useful feature in many scenarios, notably for testing purposes.

3.9 Summary

This chapter presented the design of Cogset, which is intended to be a core engine for distributed data processing, on which multiple higher-level abstractions may be constructed. MapReduce support is a central requirement, but ultimately viewed as just one specialized application of Cogset. The defining aspects of Cogset's design are its basic mechanisms for replicating and distributing data using *static routing*, and a generic *traversal* mechanism that allows parallel evaluation of an arbitrary user-defined function for all data partitions. The traversal algorithm encapsulates the details required to ensure fault tolerance and dynamic load balancing, while imposing fewer restrictions than MapReduce on the supplied user-defined functions, allowing direct implementation of algorithms such as hash joins.

Chapter 4

Implementation of Cogset

Chapter 3 describes the design of Cogset, in terms of the overall architecture and the core abstractions. In this chapter, we describe how our design was instantiated by a concrete Java implementation. Java is a widely used general-purpose object-oriented programming language that is portable and has native support for dynamically importing user-defined code at run-time. Using reflection and introspection, classes can be dynamically instantiated, and objects resulting from invoking user-defined code can be examined and type checked at run-time. This makes Java well suited for our system, but it should be noted that our overall design could be instantiated in any general purpose programming language without major modifications. The principal reason for choosing Java was for compatibility with Hadoop, which is implemented in Java. To facilitate experimental evaluation, Hadoop compatibility was one of our initial requirements for Cogset, as detailed in Section 1.4.

When listing Java code, we will in the interest of readability omit certain non-essential details such as exception types, access modifiers such as *public*/*protected*/*private*, and idiomatic code such as "getters" and "setters" (i.e., public access methods for private fields). If our design was instantiated in another programming language, such details would be prone to change, but the essential interfaces that we list could be transcribed into other object-oriented languages such as C++ or C# in a straightforward manner.

When describing our implementation, we adopt a bottom-up approach, working our way from the initial low-level details that must be addressed internally by Cogset to the higher-level interfaces to which client code must explicitly relate. Finally, we describe how the actual MapReduce interfaces that are compatible with Hadoop are implemented as a thin layer on top of the more generic interfaces to Cogset's core abstractions.

This bottom-up approach also offers a different perspective on the system's design than the top-down view given in Chapter 3. In some respects, this may provide a more instructive motivation for the design. When developing software, implementation and design are commonly overlapping phases in a cycle of gradual refinement. As such, the most accurate way to describe our implementation is not just as an instantiation of the final design, but also as a process that was instrumental in shaping it. We seek to reflect this duality by motivating our implementation choices along the way, pointing out alternative approaches that were identified but discarded.

4.1 Records

The key to handling large data sets is to partition them, so that each partition can be processed independently and in parallel. A prerequisite for partitioning is to define which contiguous byte sequences must be processed as one, or equivalently, the boundary points at which partitioning is allowed. The common term for this indivisible sequence of bytes is a *record*, and in a system for data storage and processing, such as Cogset, it constitutes both the basic data unit, and by extension, the unit of processing.

Cogset's public interfaces relate to individual records in various ways; records may for instance be added to existing data sets, inspected in order to assign a partition number, or read by user code during a traversal. In these contexts, when records are exposed to user code, they are represented as regular, in-memory Java objects. This is convenient, not just for type safety, but also because established object-oriented design patterns can be employed even where records are involved. For example, derived properties of a record (e.g., converting between units of measurement) may be implemented as regular methods to be invoked, and records may be compared in the usual way via the standard *Comparable* interface, or used as keys in an associative map such as the standard *HashMap* class.

The next section describes an important aspect of Cogset's implementation, which may frequently be overlooked in higher-level discussions: how in-memory records are converted to and from bytes stored on disk.

4.2 Record Formatters

Cogset supports records of both fixed and variable length. Fixed length records are less flexible, but have the advantage that the boundary points in a stream of bytes can easily be computed, since they are all multiples of the record length. With variable length records, it may be less trivial to determine valid boundaries. One approach is to add delimiters that can be identified while scanning the data sequentially, for example by using newline characters as delimiters and considering each line of a text file to be a (variable-length) record. Another is to add padding at certain intervals to ensure that a record boundary can always be found at certain specific offsets. Delimiters pose a storage overhead both by their necessary inclusion in between each record, and also by the need to add escape sequences for delimiters occurring within a record. Padding imposes a maximum record size, and also entails a storage overhead that depends on the average record size relative to the padding interval. For example, if records are on average 1024 bytes long, and padding is applied to align a record boundary at every 64 kilobytes, one would expect an average of 512 padding bytes in each 64 kilobyte interval, i.e. a space overhead of roughly 0.8%. In general, there are a number of ways to encode records into byte sequences, and the appropriate record encoding depends on the data types involved.

To accommodate a broad range of data types, Cogset does not enforce or require a specific record encoding. Rather, Cogset treats records as opaque objects, and requires clients to supply a *record formatter* in the form of a pluggable piece of code. The record formatter implements the necessary logic for encoding and decoding records, converting them between raw byte sequences and in-memory objects (i.e., regular class instances). This may be done using padding, delimiters, or any other scheme appropriate for the data types involved. Whenever a record is exposed to user-defined code, it is in the form of an in-memory object; the record formatter is the only piece of code that must relate to the actual record encoding.

```
interface RecordFormat<A> extends Serializable
{
    RecordDecoder<A> getDecoder(Class<A> recordClass, InputStream in);
    RecordEncoder<A> getEncoder(Class<A> recordClass, OutputStream out);
}

interface RecordDecoder<A> extends Closeable
{
    A decodeRecord();
}

interface RecordEncoder<A> extends Closeable
{
    void encodeRecord(A record);
    void encodeRecords(Iterator<A> records);
    void flush();
}

abstract class AbstractRecordEncoder<A> implements RecordEncoder<A>
{
    void encodeRecord(A record)
    {
        encodeRecords(Collections.singleton(record).iterator());
    }

    void encodeRecords(Iterator<A> records)
    {
        while (records.hasNext()) {
            encodeRecord(records.next());
        }
    }

    void flush() {}
}
```

Figure 4.1. Interfaces for record formatting.

Figure 4.1 shows the specific *RecordFormat* interface to be implemented by record formatters; given a standard Java *InputStream* or *OutputStream* instance, a record formatter wraps the instance with an adaptor that allows records of a specific type to be decoded or encoded. We refer to these adaptors respectively as *record decoders* and *record encoders*. Record formatters are parametrized by the class used to represent the records; a record formatter might only work for a specific record class, or it might work for any class implementing some specific interface. The record class could be anything ranging from the standard *String* class, to a *Map* associating string keys with string values, or a user-defined class containing fields of various types, all depending on the application needs. Either way, Cogset treats each record as an opaque object—whenever records need to be encoded as or decoded from byte sequences, Cogset simply delegates the task to a record formatter by acquiring a suitable record encoder or record decoder and remains oblivious of any internal details of the record class.

As can be noted in Figure 4.1, the *RecordEncoder* interface has separate methods for encoding a single record and for encoding a sequence of multiple records, accessed using an iterator.

Record encoders can choose to inherit default and mutually recursive implementations of these two methods from the *AbstractRecordEncoder* class, so only one of the two methods must be implemented. For simple record formatters that only consider one record at a time (relying, for example, on delimiters), it is sufficient to implement the "singular" *encodeRecord* method. Record formatters that employ encoding optimizations spanning multiple records, will want to implement the "plural" *encodeRecords* method to avoid batching records internally. For example, a formatter that employs compression typically benefits from compressing data in large chunks. Whenever a whole sequence of records is to be encoded, Cogset will invoke the *encodeRecords* method, allowing the formatter to perform the compression by accessing the original sequence of records directly, rather than batching the records internally. Since the sequence of records to be encoded is accessed via an iterator interface, it does not have to reside in-memory. It could be anything from a cursor retrieving records from a database query to a function that generates records on the fly. Despite these facilities to avoid internal batching of records, there may still be record encoders that choose to buffer data internally for convenience, for example to insert periodic checksums. To accommodate this option, Cogset will invoke the *flush* method whenever it requires all buffered data to be flushed to the underlying output stream.

Record formatters are commonly reusable for a range of record classes, and Cogset provides a library with several ready-made formatters. These include the *SerializableRecordFormat*, which relies on the standard Java serialization mechanism, and works for any class that implements the standard *Serializable* interface. There are also special record classes and formatters to facilitate interaction with Hadoop code. Hadoop internally employs a special-purpose light-weight encoding protocol as an alternative to standard Java serialization, and requires all keys and values to be instances of classes that implement a particular *Writable* interface. Cogset therefore includes a standard *WritableRecordFormat* formatter that works with any record class that implements the *Writable* interface. Furthermore, Cogset provides a generic *WritableKeyValue* record class which encapsulates an arbitrary key/value pair, and an associated record formatter. This enables seamless interfacing with Hadoop code that operates on key/value pairs.

4.3 Partitioning

In a distributed shared-nothing environment, large data sets must be partitioned for parallel processing. Ideally, the data set should be partitioned such that computational resources can be exploited to the extent possible. For example, if the computational cost of processing a record is uniform, and the environment is homogeneous, each node should process the same number of records. For some applications, a simple round-robin partitioning scheme that assigns record number i to partition $i \, mod \, P$, where P is the number of partitions, will suffice. Other operations require partitioning according to certain constraints, i.e. co-location of certain records; for example a relational equi-join requires all equal-keyed records to reside in the same partition. This particular constraint can be enforced using hash partitioning or range partitioning. The choice of partitioning algorithm is thus important, and the cost of repartitioning a data set can be high, since it requires reading, redistributing and writing all data.

Cogset provides a customizable and flexible way to specify partitioning algorithms, or implement new ones. Whenever new records are added to a data set, clients may provide a custom *partitioner* object that implements the *Partitioner* interface shown in Figure 4.2. As outlined in

```
interface Partitioner<A>
{
    int getPartition(A record, int numPartitions);
}
```

Figure 4.2. Partitioner interface.

Chapter 3, all data sets are split into a number of partitions; for each new record, the partitioner object is invoked to determine which partition should hold the record. Specifically, the record and the total number of partitions employed are passed to the *getPartition* method, and it returns a partition number for the record. Round-robin partitioning can thus be implemented simply by incrementing a counter held internally in the partitioner object, and returning the counter modulo the number of partitions.

4.4 Keys and Key Functions

As with record formatters, a number of ready-made partitioners are provided by library code. A very common approach is to partition data sets by hashing some key field contained in each record. This is the partitioning scheme hard-coded into Google's original MapReduce engine, and also the default algorithm used by Hadoop. Sorting is another fundamental operation that requires a defined ordering of records, commonly by comparing a certain "key" property of each record.

```
interface KeyFunction<A, K> extends Serializable
{
    K getKey(A record);
}
```

Figure 4.3. Key function interface.

Cogset supports both hash-based partitioning, range partitioning and sorting, but for all of these, the concept of a record key must be defined, along with methods for hashing and comparing record keys. Recall that Cogset represents a record as an opaque instance of a record class, not as an explicit key/value pair or any other pre-defined structure. To accommodate the case where records have a logical key that should be exposed, Cogset employs *key functions*, which are objects implementing the *KeyFunction* interface in Figure 4.3. A key function is invoked through the *getKey* method; it accepts a record and returns its key. Key functions serve to decouple the concept of a record key from the record class representing the record. The object returned by a key function must consistently implement the standard Java *hashCode*, *equals* and *compareTo* methods (the latter specified by the *Comparable* interface). Many standard Java classes meet this requirement, such as the *String* class and all the basic data types, so a typical key function implementation would be to simply return one of the fields defined in the record class.

The generic *HashPartitioner* and *RangePartitioner* classes provided by Cogset can be reused with any record class, despite having no view of its internals—clients just have to provide a suitable key function for extracting keys from records. As a concrete example, consider the *HashPartitioner* class, shown in Figure 4.4. Its constructor accepts a key function, which is

```
class HashPartitioner<A> implements Partitioner<A>
{
    KeyFunction<A, ?> keyFunction;

    HashPartitioner(KeyFunction<A, ?> keyFunction)
    {
        this.keyFunction = keyFunction;
    }

    int getPartition(A record, int numPartitions)
    {
        Object key = keyFunction.getKey(record);
        return Math.abs(key.hashCode() % numPartitions);
    }
}
```

Figure 4.4. A generic hash partitioner.

stored internally and used to extract keys from records as needed. This level of indirection allows the *HashPartitioner* to be reused with any record class.

Another reason to decouple key functions from record classes is that a given record may have several possible key properties, depending on context. To give a concrete use case to illustrate this, a commercial web site might employ an advertisement click tracking feature to log every event where a user clicks on an advertisement displayed on a web page. In the resulting data set, each record would have a source URL property, denoting the web page in question, and an advertisement ID property, identifying the particular advertisement. Such a data set could be mined to determine the effectiveness of the particular advertisements displayed on each page. To examine the advertisements most frequently viewed from each page, the data set should be partitioned based on the source URL property, ensuring that all records pertaining to a specific web page are co-located in the same partition. In contrast, to analyze the particular set of web pages from which each advertisement was viewed, the data set should be partitioned by the advertisement ID property, co-locating all records pertaining to a specific advertisement.

In Cogset, since key functions are decoupled from record classes, the above use case could be modeled with a single record class, and separate key functions for extracting either of the two possible key properties from a record. Partitioning the data set one way or the other would be achieved simply by passing the appropriate key function to the partitioner.

4.5 Pages

A Cogset deployment consists of multiple nodes, each hosting a particular subset of partitions. When a node is partitioning a data set, records must be transferred to the appropriate nodes, as dictated by the partitioner. Regardless of the network protocols employed, which we return to in Section 4.11, a record must be encoded as a sequence of bytes before being transmitted over the network. A certain amount of meta-data must also be communicated to the node receiving a record; most crucially, the record's partition number and the record formatter to be used for decoding the record. To reduce space overhead and amortize the cost of transferring meta-data, Cogset batches records into *pages*. The records of a page all have some meta-data in common, which is only transferred once per page. Specifically, all records in a page belong

to the same partition, and are encoded using the same record formatter. By always transferring data in units of a page, as opposed to transferring individual records, the number of I/O-related system calls is reduced, lowering the communication overhead.

Pages may have variable size, up to a configured maximum (64 kilobytes by default). Each page has a fixed size header which includes the length of the remaining data. The record formatter is encoded using standard Java serialization and follows immediately after the page header; the remaining data is a number of records encoded using the record formatter. To access the records of a page, the record formatter is first decoded using standard Java deserialization, and then used to decode the remaining data. As such, each page is a self-contained data structure with the ability to unpack itself, and the pages in a given partition may well be encoded using a variety of record formatters.

To construct a page incrementally, a separate *page builder* abstraction is provided. Given a specific record formatter, it writes records into a byte buffer that is grown on demand (a *ByteArrayOutputStream*). At any point, the page builder may be asked to construct a "finalized" page with the current contents. This constructs the actual page data structure, serializing the record formatter and appending the encoded record data, and resets the byte buffer. The page builder may thus be used as a record buffer to avoid pre-allocating large amounts of memory for pages, thereby bounding memory overhead.

4.6 On-disk Data Layout

When a Cogset node receives a new page, there is no need to unpack it before writing it to disk. The salient information about a page is readily accessible in its header: the name of the data set that the page belongs to, and its partition number. Each Cogset node maintains a separate directory in the local file system for each of the partitions that it hosts. The directory contains one data file for each data set, named after the data set. In addition, there is one log file per partition directory, recording on-going and committed distribution sessions.

Given this data layout, a Cogset node can trivially locate the appropriate local file for a page. The partition number and data set name are both accessible in the page header, and used to directly construct the corresponding path in the local file system. The entire page is then appended to the local file in question, without further inspection. This minimizes the overhead of decoding and encoding records; they are only encoded once, before transmitting them over the network, and only decoded once, when they are read from disk. This is particularly beneficial in the presence of replication; by making producers encode records *before* the data is replicated, the total overhead is further reduced.

The modular data layout, where each partition is stored in a separate directory, also aims to facilitate maintenance and redistribution of partitions. As an off-line operation, the data in a Cogset deployment can be redistributed by changing the configuration (specifically, how partitions are mapped to nodes) and directly copying the relevant directories between nodes. Similarly to pages, partition directories are self-contained data structures, which simplifies maintenance.

4.7 Record Distributors

To summarize, Cogset stores a collection of named data sets, which are partitioned, and each node hosts a certain subset of the partitions. Adding new records to a data set involves multiple

steps, building on the previously described components. First, the partition of each record must be determined by invoking a partitioner. Next, the records for each partition are batched into pages using a page builder and an associated record formatter. Finally, as pages fill up, reaching a configurable threshold size, they are transferred to the appropriate nodes, to be appended to the corresponding files on disk.

```
interface Cogset
{
    <A> RecordDistributor<A> openDistributor(Class<A> recordClass,
          Partitioner<A> partitioner, RecordFormat<A> format, String name);

    Traversal visitAllPartitions(PartitionVisitor visitor);

    void commitSessions(Collection<SessionHandle> sessions);
}
```

Figure 4.5. Main Cogset interface.

```
interface RecordDistributor<A> extends Closeable
{
    void addRecord(A record);

    void addRecords(Iterator<A> records);

    SessionHandle close();
}

interface SessionHandle
{
    void commit();
}
```

Figure 4.6. Record distributor and session handle interfaces.

All of these steps are orchestrated by a *record distributor*, which implements the interface shown in Figure 4.6. Clients may create record distributors using the *openDistributor* method of the main *Cogset* interface, shown in Figure 4.5, specifying the name of a data set to which records should be added, the record class used to represent records, a partitioner and a record formatter.

These arguments are stored internally in the record distributor. It also maintains one page builder per partition, as an array indexed by partition number. When the *addRecord* method is invoked, the record is passed to the partitioner, which returns a partition number for the record. The record is then passed to the corresponding page builder, which will buffer records until the threshold page size is reached. At this point, the distributor finalizes the page, resetting the page builder. To reduce memory consumption, pages reaching the threshold size are transmitted immediately to their destination nodes. An upper bound for the memory consumed by a record distributor is thus given by the threshold page size multiplied by the number of partitions. For example, with a threshold page size of 64 kilobytes and 1024 partitions, a record distributor will consume no more than 64 megabytes for main memory buffering. Note that records are

buffered not as instances of record classes, but as byte strings encoded by the record formatter. As such, any space saving technique—such as compression—that is applied by the formatter will also increase the number of records that can be buffered in main memory.

As noted, full pages are transmitted directly to the appropriate nodes. In turn, pages received by a node are written directly to disk. Each partition of a data set is stored as a single local file, to which pages are appended. For each file, Cogset's internal threads synchronize to ensure that pages are appended sequentially and in arrival order, preserving record and page boundaries on disk. The records originating from a particular record distributor will therefore appear in the order added. Records originating from different, concurrent distributors may be interleaved in any order.

Since pages are encoded by a record formatter prior to transmission, no further formatting is required on the receiving node and pages can be appended directly to the appropriate local files. This append-only I/O pattern is very efficient, since general purpose file systems and hard disks are generally heavily optimized for sequential bulk writes. Restricting file mutations to append-only also enables a straightforward mechanism for atomically committing changes, as described next.

4.8 Distribution Sessions

When a data set is generated or imported from an external source using a record distributor, the resulting pages are distributed and written continuously to disk on the various nodes in the Cogset deployment. In the event of a failure, for example on the client node where the record distributor executes, some pages may have been persisted while others have not. For determinism, a stricter consistency model is required, where either all or none of the records are added. Clients might rely on higher-level protocols implemented outside of Cogset to determine whether or not a data set is consistent, but Cogset also provides a built-in and flexible mechanism to ensure consistency, called *distribution sessions*.

A distribution session represents a set of records that should be added as a single atomic unit; either all of the records are persisted or none of them. Upon creation, each record distributor initiates a new distribution session, and all records added using the distributor are associated with that session. After the record distributor is closed, the session may be committed using a two-phase commit protocol coordinated by the client.

Each record distributor generates a unique *distribution session identifier* (DSI), which is a 64 bit number that includes bits from the Cogset node's IP address and process identifier to ensure uniqueness. The DSI is included in every page created by the record distributor, and follows the page as it is transmitted to other nodes. The record distributor maintains statistics of how many pages have been transmitted to each partition. Conversely, each node keeps a log of committed DSIs in each partition, as well as the number of pages written to each of its partitions in currently uncommitted sessions.

A client initiates a commit by sending a *prepare to commit* message to all nodes. The message is tailored for each node, specifying the DSI and the number of pages added to each partition hosted by the receiving node. Upon receipt, a node checks that the correct number of pages has been persisted for each of its partitions. If this is verified, the node responds with a *commit* message, attesting that all records have been persisted. Otherwise, the node responds with an *abort* message. The client tallies the responses; if all nodes respond with *commit* mes-

sages, the session is committed and the client broadcasts a final *committed* message to notify all of the nodes.

There is no explicit abort operation for a session, and no need to roll back state in the event of failures. Instead, pages whose sessions are uncommitted are simply skipped whenever a data set is read. On disk, each page has a small header that specifies the length of the page and its associated DSI. Skipping a page is therefore a simple matter of moving the file pointer (seeking) past it, and there is no need to actually decode its contents in order to skip it.

Sessions that go uncommitted (for whatever reason) will leave a certain amount of garbage data that will have to be skipped by subsequent readers. Under the common assumptions that failures are relatively rare, and most sessions will be committed, or in the case of short-lived data sets, this should pose little space overhead. If it *is* deemed to be a problem, the procedure for garbage collection is straightforward: simply read a data set, skipping over all uncommitted data, write a new copy of it, and remove the old one.

When a record distributor is closed using its *close* method, the return value is a *session handle*—an object that encapsulates the DSI of a session as well as other relevant statistics from the session. The session may subsequently be committed by invoking the *commit* method of the session handle, or the client may choose to commit multiple sessions at the same time, by passing a collection of session handles to the *commitSessions* method of the main *Cogset* interface. In other words, a client may choose to add records to multiple data sets (using multiple record distributors) and subsequently commit all of those changes as a single atomic operation.

In general, the main drawback of using a two-phase commit protocol is that participants may deadlock if the central coordinator fails while collecting commit/abort votes [59]. In Cogset, the client acts as the coordinator, and the potential for a failing client to deadlock the Cogset nodes should be avoided. However, the traditional concern for deadlocks stems from database applications, where the participating nodes typically hold a set of locks while waiting for the coordinator to confirm the final outcome of the protocol (abort or commit). In Cogset, the only operations to be committed are simple file appends; there is no need to hold any locks or otherwise impede progress while waiting for the client to make a decision, and hence no risk of deadlocking.

4.9 Traversals

Given the implementation details presented so far, a notable omission may be the lack of any interface for reading records. In traditional systems involving a shared file system and a separate abstraction for distributed processing, data is commonly streamed over the network to the processes reading the data. One of the observations motivating Cogset is that it is more efficient to schedule the processing on a node where data can be read locally, thus reducing network load. Ideally, all reading of data should happen locally, and data should only be transferred to another node for the purpose of storing it there. At the same time, clients should not have to be concerned with the exact locations of data, since that would breach the abstraction layers.

To meet both of the above requirements, Cogset offers an untraditional, visitor-based interface for data access. When clients wish to process a data set, they do not fetch records from the Cogset nodes. Instead they specify a function to execute locally (and in parallel) on the actual Cogset nodes. Specifically, clients specify a *visitor* function that should be invoked

```
interface PartitionVisitor extends Serializable
{
    void visitPartition(Partition p);

    double estimatedCost(Partition p);
}
```

Figure 4.7. Partition visitor interface.

```
interface Partition
{
    <A> RecordReader<A> readDataSet(Class<A> recordClass, String name);

    <A> RecordReader<A> readSortedDataSet(Class<A> recordClass,
        KeyFunction<A, ?> keyFunction, RecordFormat<A> format, String name);

    void scanDataSet(RecordVisitor<A> visitor, String name);

    long dataSetSize(String name);

    int getPartitionNumber();
}
```

Figure 4.8. Partition interface.

once for each partition number. The visitor is a serializable Java object that implements the *PartitionVisitor* interface shown in Figure 4.7. By invoking the *visitAllPartitions* method of the main *Cogset* interface (see Figure 4.5), a client initiates a *traversal*, in which copies of the specified visitor are distributed to all nodes, and invoked in parallel, once per partition. In Section 3.5, we described the high-level traversal algorithm; here, we focus on the implementation details and the specific interfaces involved.

During a traversal, each Cogset node repeatedly selects one of its locally hosted partitions, invoking the visitor for each partition through the *visitPartition* method. The supplied argument implements the *Partition* interface shown in Figure 4.8, through which the visitor can read records or add new ones. In particular, the *readDataSet* method can be used to read the records of a named data set; the returned *RecordReader* is a custom iterator that provides access to all the records of the specific partition and only those. This amounts to sequentially reading a local file (while skipping over uncommitted pages), so only local I/O is performed. The records of the given partition are returned in the order in which they were originally added. To read all records in a data set, a visitor may simply invoke the *readDataSet* for every partition, i.e. once per invocation of *visitPartition*.

An alternative way of reading records is provided by the *scanDataSet* method. This provides yet another visitor-based access method, allowing the visitor to process all records of a partition in parallel (in no particular order). The supplied *RecordVisitor* callback is invoked once for each record. Under the hood, Cogset employs multiple threads to scan a data set; while pages must be read sequentially, they are decoded and scanned in parallel. Record visitors must therefore be re-entrant, i.e. tolerant of concurrent invocations from separate threads. The two approaches to reading records complement each other; the conventional iterator-based interface

```
interface RecordVisitor<A>
{
    void visitRecord(A record);
}

interface RecordReader<A> extends Iterator<A>, Closeable
{
    A readRecord();
}
```

Figure 4.9. RecordVisitor and RecordReader interfaces.

is useful when the order in which records are read matters, while the visitor-based interface promises better multi-core performance for CPU-intensive tasks, due to its underlying multi-threaded implementation. The *RecordVisitor* and *RecordReader* interfaces, instrumental to the two approaches for reading records, are both detailed in Figure 4.9.

Visitors may also add records to new data sets using the *openDistributor* method. Its signature is identical to the equally named method in the main *Cogset* interface, but the returned record distributor has a slightly different behavior. When a given invocation of *visitPartition* completes, all record distributors that were created during the call are implicitly closed, without committing the distribution sessions. Instead, the session handles are communicated back to the client that initiated the traversal. If or when to commit the sessions is a matter of client policy, as discussed in Section 3.7.

Commonly, a visitor performs a symmetric computation and does the same processing for every partition, such as filtering or aggregating the records of a particular data set. In this case, its *visitPartition* method will have no need to determine which specific partition is being processed. If a visitor performs an asymmetric computation it may wish to process certain partitions differently; to this end, the *getPartitionNumber* method can be used to determine which particular partition is being processed in any given invocation of *visitPartition*.

In Section 3.6 we described how Cogset's traversal algorithm allows dynamic load balancing by carefully selecting an order of processing for partitions. The semantics of a traversal only require each partition to be processed once, but it may happen in any order, and using any of the partition replicas. To make intelligent scheduling decisions, it helps to have an estimate of how costly it will be to process a given partition. By preferring to process more costly partitions first, there will be less variance towards the end of a traversal, when some nodes must inevitably go idle.

The *estimatedCost* method of the *PartitionVisitor* interface provides a hook where visitors may indicate an estimate for how costly it will be to process a given partition. These estimates are obtained by each node at the start of a traversal, by invoking the method once for each partition hosted by the node. The returned estimates have no predefined domain, but are used as relative weights. A typical implementation is to return the amount of data to processed in a given partition, obtained using the *dataSetSize* method of the *Partition* interface. Bear in mind that visitor functions are user-defined black boxes to Cogset, and have no accompanying declaration naming the data sets that will be accessed (if any). As such, there is no way for Cogset to derive this estimate automatically. On the other hand, accurate estimates are not crucial for load balancing; they mainly serve to minimize idle time towards the end of a traversal, by ensuring

that costly partitions are processed early. For heterogeneous or hard to estimate computations, visitor functions may simply return a constant cost estimate of 1 for every partition.

A crucial property of the *Partition* interface is that it provides access to a particular partition of *all* data sets. Applications may ensure that records from different data sets are co-located by assigning them to the same partition number. To see how this is useful, consider the example of a relational equi-join implemented using the hash-merge join algorithm. The data sets would first be populated by adding the records with a record distributor, using a partitioner that hashes the desired join attributes. Records with equal join attributes would thus be co-located in the same partition. The join could subsequently be performed using a single traversal with a fully symmetric visitor: for each partition, it would read the data sets in sorted order using the *readSortedDataSet* method, merging them to keep only the records of interest, and add all resulting records to a new data set. Depending on the size of the partition, *readSortedDataSet* either sorts records in-memory or falls back to an external merge sort. In the latter case the supplied *RecordFormat* argument is required, in order to encode records as they are stored in temporary files.

4.10 MapReduce Support

A classical MapReduce computation can be performed using an extension of the above hash-merge join algorithm. Recall that in a MapReduce context, records are key/value pairs, and there are user-defined map and reduce functions to be applied. A MapReduce computation can be implemented manually as follows, using the generic interfaces to Cogset's core abstractions. When populating the data set, all records are passed to the map function, and the key/value pairs emitted by the map function are added to an intermediate data set, partitioned by hashing the keys. The reduce phase is performed by a traversal whose visitor function reads each partition of the intermediate data set, sorted by key, and applies the reduce function once for each contiguous sequence of equal-keyed records.

Cogset implements a library to generalize this, providing a traditional MapReduce interface on top of Cogset's traversal mechanism. In accordance with the requirements set forth in Section 1.4, this library implements Hadoop's public API, for compatibility with existing Hadoop applications. A Hadoop job is executed by Cogset as two consecutive traversals. The first traversal implements the map phase by reading one or more input data sets and applying the map function to each input record; the map output is automatically partitioned by passing it to a record distributor. The second traversal implements the reduce phase by sorting each map output partition and applying the reduce function.

Figure 4.10 illustrates how the map phase traversal is implemented. In this instance, the visitor class implements both the *PartitionVisitor* and *RecordVisitor* interfaces from Cogset, in addition to Hadoop's *OutputCollector* and *Reporter* interfaces. The *estimateCost* method, which is used by Cogset for scheduling purposes, estimates the relative cost of processing a partition by returning the total amount of data to be mapped. The *visitPartition* method performs the mapping by scanning all input data sets, and populating a new data set with records emitted from the map function. In response to the *scanDataSet* invocations, the *visitRecord* callback is invoked once for each input record. *visitRecord* in turn invokes the user-defined map function, which implements Hadoop's *Mapper* interface. All records emitted from the map function are collected by the *collect* method, and persisted using a record distributor.

```
class MapVisitor implements PartitionVisitor,
    RecordVisitor<WritableKeyValue>, OutputCollector, Reporter
{
    Mapper mapper;
    RecordDistributor<WritableKeyValue> out;

    double estimateCost(Partition p) {
        double cost = 0;
        for (String dataSet: getInputDataSets()) {
            cost += p.dataSetSize(dataSet);
        }
        return cost;
    }

    void visitPartition(Partition p) {
        mapper = getMapper();
        Partitioner par = new HashPartitioner();
        String dataSet = getOutputDataSet();
        out = p.openDistributor(par, dataSet);
        for (String dataSet: getInputDataSets()) {
            p.scanDataSet(this, dataSet);
        }
    }

    void visitRecord(WritableKeyValue record) {
        mapper.map(record.key, record.value, this, this);
    }

    void collect(Writable key, Writable value) {
        out.addRecord(new WritableKeyValue(key, value));
    }
}
```

Figure 4.10. Partition visitor for the map phase of a MapReduce job.

Figure 4.11 shows the visitor used for the second traversal, implementing the reduce phase. It is similar to the map phase visitor, but this time, an iterator-based interface is used for reading the input data set. By using the *readSortedDataSet* method, the visitor is requesting that Cogset should sort the data set partition on the fly. The sort order is determined in the usual way for Java objects, i.e. through the *Comparable* interface, which must be implemented by the record class. Depending on available memory and the size of the data set partition, an external sort may be required, temporarily flushing data to disk. This is an additional reason—beyond finer-grained load balancing—to configure Cogset to use a relatively high number of partitions: with a higher number of partitions, each data set partition is on average smaller, and the likelihood that it can be sorted in-memory increases. Note that a Cogset node only visits one partition at a time during a traversal, so all of the machine's physical memory is generally available for sorting each partition.

To execute a MapReduce job using Cogset, users may construct a Hadoop *JobConf* object in the regular way and replace the call to *JobClient.runJob()* with a similar call to the Cogset MapReduce library. Cogset ignores many settings in the *JobConf* object that are particular to Hadoop, but existing map, reduce and combine functions all work as expected. The reporter

```
class ReduceIterator implements Iterator<Writable>
{
    // Details omitted; custom iterator that wraps a record reader
    // with the capability for traversing contiguous sequences of
    // equal-keyed records, assuming sorted input
}

class ReduceVisitor implements PartitionVisitor, OutputCollector, Reporter
{
    RecordDistributor<WritableKeyValue> out;

    double estimateCost(Partition p) {
        return p.dataSetSize(getInputDataSet());
    }

    void visitPartition(Partition p)
    {
        Reducer reducer = getReducer();
        Partitioner par = new HashPartitioner();
        String dataSet = getOutputDataSet();
        out = p.openDistributor(par, dataSet);
        RecordReader<WritableKeyValue> in;
        in = p.readSortedDataSet(WritableKeyValue.class,
                                 WritableKeyValue.DEFAULT_KEY,
                                 WritableKeyValue.DEFAULT_FORMAT,
                                 getInputDataSet());
        ReduceIterator iter = new ReduceIterator(in);
        // Invoke the reducer once for each unique key in the input:
        while (iter.hasNextKey()) {
            WritableComparable key = iter.nextKey();
            reducer.reduce(key, iter, this, this);
        }
    }

    void collect(Writable key, Writable value) {
        out.addRecord(new WritableKeyValue(key, value));
    }
}
```

Figure 4.11. Partition visitor for the reduce phase of a MapReduce job.

object can be used to set status messages and increment counters; this information is relayed to the Cogset client. Also supported are the new *org.apache.hadoop.mapreduce* interfaces that were introduced with version 0.20 of Hadoop.

4.11 Communication Layer

Cogset nodes must communicate with one another and with clients both to distribute data and to coordinate their activities during traversals. To this end, we considered various implementation mechanisms and network protocols. Crucially, we sought to implement a communication layer that offered a stable interface, with the appropriate semantics required by Cogset, while allowing the underlying implementation and network protocols to be varied for experimental purposes. The choice of network protocols is important in a shared-nothing environment, where all inter-node communication must be done over an unreliable network, so we considered it important to facilitate future experimentation along that axis.

Both the original MapReduce implementation and the Hadoop implementation rely on remote procedure calls (RPCs) for communication. RPCs are a convenient abstraction for distributed applications, allowing inter-process communication to be structured like regular method calls, masking the fact that the invoked method executes in the address space of another process, typically on another machine. However, RPCs are a poor fit for many of the communication patterns employed by Cogset. When distributing data, a single page should be sent to multiple nodes for replication, whereas a conventional RPC targets a single node. Similarly, the two phase commit protocol must broadcast an initial message before waiting to collect replies from all participating nodes, tallying their votes. Synchronous RPCs are a poor fit for such a communication pattern.

We therefore decided to adopt a lower-level, but more general communication primitive as our common denominator: non-blocking, reliable message passing. Asynchronous communication is a better fit for Cogset's communication patterns, and synchronous abstractions can always be implemented on top of asynchronous abstractions. In contrast, the opposite may be harder to achieve in practice: each synchronous call would have to be wrapped in a separate thread in order to provide asynchronous semantics. The overhead of such an approach could be excessive with a high number of concurrent calls.

Reliable delivery of messages over an unreliable network can be achieved through retransmission of lost messages. This must be triggered by timeouts, whose duration must be tuned, and known complications are the possibility of duplicate or out-of-order messages. These issues have already been addressed by the ubiquitous and thoroughly tested TCP protocol, so we chose to employ TCP for all communication. TCP is connection-oriented, but to provide a pure message-passing interface, we implemented a communication layer with a connection-less interface, explicitly specifying the destination node for each individual message. Internally, the communication layer pools and re-uses TCP connections to avoid the overhead and latency of redundantly establishing duplicate connections between nodes. An added benefit of this approach is that the flow control mechanisms inherent in a TCP connection thus serve to throttle the rate of message delivery. This avoids overloading a recipient node with data messages from a sending node that produces data at a higher rate than it can be persisted by the recipient. An alternative implementation of the communication layer, for example based on UDP, can be substituted without affecting other parts of the Cogset implementation.

Beyond reliable delivery and flow control, TCP also provides *ordered delivery* of messages. As a consequence, pages (and hence, records) are persisted in the order in which they are created. Our protocols for committing distribution sessions or coordinating traversals do not rely on the ordering guarantee, so it is possible to modify our message passing implementation to employ a network protocol without ordering guarantees. The consequence would be slightly different semantics when adding records to data sets. In theory, the records would have no well-defined ordering. This may actually be acceptable for many applications, that either sort records before accessing them (as in the reduce phase of a MapReduce computation), or process each record independently of others (as in the corresponding map phase).

Considering the fact that Cogset replicates each page and distributes it to a number of nodes, there is an apparent potential to save some network bandwidth by employing a multicast protocol. However, this would complicate the implementation of the communication layer, both because the level of multicast support by the network infrastructure may vary, and also because it would preclude TCP as an underlying network protocol. As such, an implementation based on multicast would have to address the additional concern of reliable delivery. For this reason, we have refrained from using multicast in our current implementation of Cogset. It remains an interesting area to explore in potential future work.

4.12 Summary

This chapter presented the Java implementation of Cogset, describing its central programming interfaces and how Hadoop-compatible MapReduce support is implemented in a separate layer on top of the core Cogset interfaces. Cogset represents records using regular in-memory class instances, and provides hooks where the user can influence how records are encoded to and decoded from byte strings. Clients may also influence data placement by defining custom partitioning functions, and may determine sort order by implementing the *Comparable* interface in the record classes. The MapReduce support is accessed using a variant of the Hadoop API, by constructing a *JobConf* object in the usual way and passing it to Cogset. Clients may also bypass the MapReduce layer and execute traversals directly, specifying their own partition visitors. Such partition visitors have free reins to access and modify any and all data sets, and are therefore less restricted than the map and reduce functions in a MapReduce computation.

Chapter 5

Experimental Evaluation

To investigate our thesis that static routing can be used as a foundation for a high-performance MapReduce engine, we first designed and implemented Cogset, as described in the previous two chapters. This chapter describes the next step, where we experiment with Cogset, measuring its performance, and compare it to other systems in order to draw conclusions.

Our performance evaluation compares Cogset to Hadoop, an established and widely deployed open-source MapReduce engine that represents the state of the art for traditional engines. These experiments support our main thesis, in that Cogset generally outperforms Hadoop, but they also prompted a closer investigation of Hadoop's internals, to fully understand the underlying reasons. This chapter also relates our experience from that investigation, where we uncovered certain performance issues specific to Hadoop. We show how these issues can be addressed, and include results on how Hadoop performs after those optimizations have been applied. This constitutes valuable experience that both affirms some of our own design choices, and highlights potential pitfalls that should be avoided by others.

Compared to previous work, Cogset may be viewed as a hybrid between a traditional MapReduce engine and a parallel database: its architecture deviates from the traditional MapReduce approach of dynamic routing, drawing inspiration from parallel databases by adopting static routing. Yet Cogset embraces the MapReduce philosophy of structuring computations around user-defined functions, and creating a generic, reusable framework for evaluating such functions in a highly parallel, fault-tolerant and load-balanced way.

Since MapReduce was first introduced, its relative benefits and drawbacks as compared to parallel databases have been the subject of considerable debate. Certain aspects of that debate are inherently subjective, such as the various views on which programming interfaces should be provided, and on how data should be modeled. Such individual preferences may come naturally, as a matter of intuition, but while the merits of each approach can be propounded anecdotically, they are hard to quantify objectively.

Consequently, the most concrete points of contention revolve around perceived performance issues. In 2009, Pavlo et al. published a comprehensive work comparing MapReduce to parallel databases, viewing them as opposing paradigms for large-scale data analysis, and concluding that parallel databases have strikingly better performance [60]. The Hadoop engine was adopted as a representative for the MapReduce paradigm, and compared to three commercial parallel database systems. For this work, the authors developed a new benchmark that we refer to here as *the MapReduce/Database (MR/DB) benchmark*.

Given Cogset's position as an intermediate point in the design space between traditional MapReduce engines and parallel databases, the MR/DB benchmark was a natural benchmark to adopt for our performance evaluation. Cogset is intended to serve as a potential replacement not just for MapReduce engines, but also for computations that might otherwise be delegated to parallel databases. By adopting a benchmark designed to compare the two, we ensure that the workload is relevant for our intended application areas, and therefore suitable for testing our hypotheses. Adopting a previously established benchmark also has the obvious advantage that it allows indirect comparisons to other systems for which results using the same benchmark have been reported.

In the remainder of this chapter, we first present the MR/DB benchmark and detail its constituent data sets and tasks. We then give a brief overview of the systems that have previously been evaluated using the same benchmark. Next, we describe our experimental setup, present our results and discuss their significance. Finally, we give an in-depth account of how we used our experiments to investigate Hadoop's behavior and devised a new set of optimizations specifically for Hadoop.

5.1 The MR/DB Benchmark

The MR/DB benchmark is explicitly designed to compare the performance of MapReduce engines and parallel databases. It consists of five tasks that can be expressed either as SQL queries or as MapReduce computations: *Grep*, *Select*, *Aggregate*, *Join*, and *UDF*. The benchmark includes Java source code for the MapReduce implementations of each task, using the Hadoop API. It also defines relational schemas for the data sets employed, and includes tools to generate them. Before executing the benchmark tasks, the data sets are imported by each of the benchmarked systems in a system-specific way. Parallel databases may take this opportunity to build index structures for the data. The MapReduce implementations of the benchmark tasks access records sequentially from regular files, without relying on index structures.

In the following, we detail each of the benchmark tasks in turn, describing the data sets involved, and outline how the tasks are typically executed by a parallel database and by a MapReduce engine.

Grep The *Grep* task was first described in the original MapReduce paper [6]. It performs a complete scan of a data set looking for lines of text that match a specific pattern and emits matching lines. *Grep* executes on a randomly generated data set of 10 billion 100-character lines, comprising about 1 TB of data. The search pattern occurs infrequently, so little output is generated. The SQL schema and query for *Grep* is shown below:

```
CREATE TABLE Data (
   key          VARCHAR(10) PRIMARY KEY,
   field        VARCHAR(90)
);

SELECT * FROM Data WHERE field LIKE '%XYZ%';
```

There is no index on the field being searched, so a parallel database must scan through every record sequentially in order to execute the query. Depending on the on-disk data layout, the database might be able to access only the relevant field of each record. The

MapReduce implementation of this task scans through the entire data set, using the map function to search for and emit matching records. For this simple task, there is no reduce function, so in this case the output from the map phase constitutes the final output of the MapReduce computation. (Both Hadoop and Cogset support map-only computations; this option is not explicitly mentioned in the original MapReduce paper, but it is a simple extension.)

The remaining four tasks use three related data sets. Two data sets model log files of HTTP server traffic: *Rankings* contains URLs and their associated *pageRank* values, and records in *UserVisits* denote users visiting web sites. UserVisit records include multiple fields, such as the URL that was accessed and the originating source IP address (*sourceIP*). The third data set is a randomly generated collection of HTML documents with embedded hyperlinks. The SQL schemas for these data sets are as follows:

```
CREATE TABLE Rankings (
  pageURL      VARCHAR(100) PRIMARY KEY,
  pageRank     INT,
  avgDuration  INT
);

CREATE TABLE UserVisits (
  sourceIP     VARCHAR(16),
  destURL      VARCHAR(100),
  visitDate    DATE,
  adRevenue    FLOAT,
  userAgent    VARCHAR(64),
  countryCode  VARCHAR(3),
  languageCode VARCHAR(6),
  searchWord   VARCHAR(32),
  duration     INT
);

CREATE TABLE Documents (
  url          VARCHAR(100) PRIMARY KEY,
  contents     TEXT
);
```

Select The *Select* task selects all Rankings records with a *pageRank* larger than some threshold. Parallel databases typically index this table by sorting it by *pageRank*, and can trivially find matching records—the MapReduce version uses a map-only job to perform a sequential scan over the entire data set. It corresponds to the following SQL query:

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

Aggregate The *Aggregate* task groups UserVisits records by *sourceIP* and calculates the total advertisement revenue (*adRevenue*) generated by visits from each IP. Using MapReduce, this task is implemented by a map function that outputs the *sourceIP* and *adRevenue* for each input record (projecting just the relevant attributes), and a reduce function that calculates the total *adRevenue* for each IP. A combiner function is employed for pre-aggregation in the map phase. In SQL, this is a classic group-by aggregation query, as shown below:

```
SELECT sourceIP, SUM(adRevenue)
  FROM UserVisits GROUP BY sourceIP;
```

A second variant query groups records by a substring of the IP address, producing a smaller result set, with fewer groups:

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

Join The *Join* task performs a join between the Rankings and UserVisits data sets, to produce intermediate records that associate *pageRank*s with the URLs of each user visit from a specified date range. The intermediate records are grouped by *sourceIP*, and for each group of user visits originating from the same IP, the total *adRevenue* and average *pageRank* is computed. Finally, the single source IP that generated the highest total *adRevenue* is emitted, along with its average *pageRank*. For this task, parallel databases can use an index to retrieve user visits from the specified date range, whereas MapReduce performs a complete scan of the input. Furthermore, grouping by source IP is done when data is loaded into the parallel databases, so they can perform the join locally on each node. In contrast, MapReduce must repartition the data to perform this grouping, by using the join attribute (URL) as intermediate keys, and completing the join in the reduce phase. A second MapReduce pass computes the total *adRevenue* and average *pageRank* for each source IP, and a third pass finds the single record with the highest total *adRevenue*. In SQL, this task is implemented by one query to populate a temporary table with aggregated values, and a second query to select the single output record:

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

SELECT sourceIP, totalRevenue, avgPageRank
  FROM Temp
 ORDER BY totalRevenue DESC LIMIT 1;
```

UDF The final benchmark task, *UDF*, exercises the capabilities of parallel databases for integrating user-defined functions (UDFs) into query plans. The task inverts a web graph by extracting all hyperlinks from each HTML document and aggregating the total number of in-links for each unique URL. MapReduce uses a map function that parses HTML code to extract hyperlinks from the document contents. Since such parsing cannot easily be expressed in SQL, parallel databases must integrate a user-defined function *F*, similar to the map function, which extracts hyperlinks into a temporary table:

```
SELECT INTO Temp F(contents) FROM Documents;

SELECT url, SUM(value) FROM Temp GROUP BY url;
```

The second SQL query aggregates the output from the user-defined function and corresponds to the reduce phase of the MapReduce computation.

5.2 Previously Benchmarked Systems

Previous work has employed the MR/DB benchmark to compare several other systems to Hadoop. Here, we briefly describe these systems and their previous benchmark performance.

Vertica [61] is a commercial parallel database system based on the C-Store research project [62]. Vertica stores table columns (record attributes) separately, which is advantageous if only a subset of the columns are accessed frequently. Also, efficient compression techniques can be employed to improve performance for read-mostly workloads. In the original benchmark paper, Vertica outperformed Hadoop by significant factors for most tasks—this can be attributed both to its compression techniques and queries that allow use of indexes for data access. The corresponding MapReduce programs are implemented as brute force scans over the entire data sets. Subsequent criticism notes that realistic map functions for these queries would take advantage of similar indexes, either occurring naturally or otherwise supported by the input data source [15].

DB-X is an unidentified parallel edition of a commercial database system and was included in the original MR/DB benchmark. Like Vertica, DB-X employs hash partitioning and compression, but the storage is row-based, so individual columns can not be accessed separately. DB-X also outperformed Hadoop, but given the difference in storage formats, it did not match the performance of Vertica for the queries that accessed small subsets of the columns in the input tables.

HadoopDB [54] is a hybrid system that employs Hadoop as a coordinator for a collection of single-node PostgreSQL databases. It extends Hive [41], to draw on its support for compiling SQL queries into Hadoop execution plans. HadoopDB modifies the execution plans so that certain sub-queries are executed in parallel by the PostgreSQL databases; the results are then merged by Hadoop. HadoopDB was evaluated using the MR/DB benchmark in a separate paper [54], comparing its performance to Hadoop and Vertica. Results for DB-X were only included as extrapolations from the original benchmark.

5.3 Experimental Setup and Benchmark Adaptation

For our experiments, we used a cluster of 25 Dell PowerEdge 1955 machines, interconnected by a HP ProCurve 4208VL with a 48-port 1 Gb/s switched Ethernet module. Machines had two quad-core Intel Xeon processors and two SCSI disks configured in RAID 0. There were 3 racks: racks A and B held 17 machines with 2 GHz processors and 8 MB level 2 cache, and rack C had 8 machines with 2.66 GHz processors and 12 MB level 2 cache. The disks in racks B and C had a peak bandwidth for reads of about 155 MB/s, while the disks in rack A were smaller but faster, with a peak of about 195 MB/s. These speeds were measured using *hdparm*, a command line utility for viewing and tuning hard drive parameters, using its *-t* option. We dedicated a separate node in rack A, with similar specifications, for running the Hadoop name node, the Hadoop job-tracker, and the Cogset client processes. Using *iperf*, an open source cross-platform network testing utility, we measured the effective TCP payload bandwidth between nodes in our cluster to 112 MB/s.

Option	Value	Default
dfs.replication	2	3
dfs.block.size	256 MB	64 MB
dfs.datanode.scan.period.hours	-1 (disabled)	3 weeks
io.sort.factor	100	10
io.sort.mb	256	100
mapred.child.java.opts	-Xmx512M	-Xmx200M
mapred.job.reuse.jvm.num.tasks	-1	-1
tasktracker.http.threads	60	40

Table 5.1. Hadoop configuration

We used Java 1.6 and Hadoop version 0.20.2, configured as in the MR/DB benchmark, with a few options different from the default. These configuration changes are summarized in Table 5.1. We reduced the replication degree in HDFS to 2, to match Cogset's default configuration, and because disk space was a scarce resource in our cluster. Pavlo et al. reported that this change made no difference to Hadoop's performance [60]; based on initial test runs, we drew the same conclusion. To minimize the overhead of creating new processes, we configured Hadoop to re-use instances of the Java Virtual Machine (JVM). (This has become the default setting in Hadoop 0.20.2, but we state it explicitly here for clarity, since JVM initialization is a known cause of overhead in earlier versions of Hadoop [60].)

We reduced the amount of logging to the bare minimum by filtering out all but warnings and errors and logging to local disk only[1]. We disabled the periodic block scanning in HDFS, which performs low-priority background maintenance tasks such as checksum verification, to avoid any interference with our experiments. Like Hadoop, Cogset was configured to maintain 2 replicas of all partitions. We used a total of 500 partitions, assigning 40 partition replicas to each of the 25 nodes using the chained declustering algorithm described in Section 3.4.

In addition to the time it takes to execute each benchmark task using Hadoop, the MR/DB benchmark also reported the time to merge all output files into a single result file. This extra step is generally unnecessary when using the MapReduce paradigm, since subsequent MapReduce jobs can operate directly on a partitioned set of input files. Moreover, the merging step generally takes excessive time to execute using Hadoop, since Hadoop performs poorly for jobs that process numerous small input files. In Section 5.6 we explain how this is due to a weakness in Hadoop's task scheduling algorithm. With Cogset, a set of small files can generally be merged in negligible time, if a single result file (stored in a single partition) is required, but again, this is generally not necessary.

For practical reasons, we needed to make a few minor modifications to the MR/DB benchmark code. First of all, the benchmark was adapted to run either on Hadoop or on Cogset depending on a command-line option. Also, Cogset requires map and reduce functions to be thread-safe. Such functions are mostly thread-safe by nature, since they have no dependencies on other records or keys, but there are exceptions. In particular, a few shared instances of the *SimpleDateFormat* class were replaced with separate instances per thread, using the *ThreadLocal* class from the standard library.

Similarly, we adapted the benchmark data generation tools to optionally output generated data to Cogset, using the record distributor abstraction described in Section 4.7. We employed

[1]Logging was configured by modifying *log4j.properties* and certain environment variables.

the *WritableRecordFormat* record format described in Section 4.2 to use Hadoop's custom serialization protocol for storing records on disk. Like Hadoop, Cogset accessed all records sequentially for all benchmark tasks, without relying on any pre-generated index structures.

For each benchmark task, we executed 10 trials using the same input data. The error bars in our graphs show 95% confidence intervals based on these samples. Between each trial, we stopped all processes, cleared the file system read cache on all nodes (using the shell command *"sync; echo 1 > /proc/sys/vm/drop_caches"*) and restarted all processes. Hadoop was given one minute to settle into a steady state before initiating any MapReduce jobs.

5.4 Hadoop Optimizations

Hadoop's performance may be sensitive to its exact configuration. Options that may affect performance include the number of tasks to execute concurrently on each node, the amount of main memory to reserve for sorting, and the total number of map and reduce tasks to employ, which determines the number of input and intermediate partitions, respectively.

Furthermore, the performance of a Hadoop job may depend not just on the user-defined code implementing the map and reduce functions, but also on other hooks such as the input and output formats used for record encoding. These are typically provided by library code, but may be configured independently for each job.

As a consequence, our experiments must closely follow the MR/DB benchmark configuration to yield comparable results. At the same time, it may be possible to find alternative configurations of Hadoop that yield improved performance. When executing our experiments with the original benchmark configuration of Hadoop, Cogset's performance was strikingly better. While investigating the reasons for this, we discovered ways to significantly improve Hadoop's performance both by reconfiguring it and by plugging in alternative implementations for some of its user-defined hooks.

We therefore report two sets of benchmark results for Hadoop. The results labeled "Hadoop" are obtained by executing the benchmark using the configuration specified by the benchmark, while the results labeled "Optimized Hadoop" show the best performance we were able to achieve with Hadoop when applying our own optimizations. As the next section shows, these optimizations were surprisingly effective, and are interesting in their own right. We therefore revisit that topic in Section 5.6, giving a detailed account of how we optimized Hadoop's performance.

5.5 Benchmark Results

This section reports the results from executing the MR/DB benchmark with Cogset, Hadoop, and our optimized version of Hadoop. We first present the results for each of the five benchmark tasks, then conclude with an indirect comparison to previously benchmarked systems, based on the observed performance relative to Hadoop.

5.5.1 Grep Results

Grep exercises sequential scanning of data. Since our experiments start with a cold file system cache, the 1TB data set must be read into memory. The map function filters the records using a very selective predicate. Therefore, the bottleneck in this experiment ought to be how

fast data can be read from disk. Averaged over all nodes, the disks in our cluster have a peak read bandwidth of approximately 168 MB/s. During a traversal, Cogset employs a single process per node that continuously reads sequentially from the local file system. In the map phase, Cogset scans data at a rate of 142 MB/s. In contrast, Hadoop scans at just 56 MB/s. With the optimizations described in Section 5.6, this was improved to 76 MB/s. Figure 5.1 shows the resulting execution times.

5.5.2 Select Results

Select also scans and filters the input, but its predicate is less selective, so a non-trivial amount of map output must be persisted. Figure 5.2 shows the results: as before, Cogset benefits from its higher scanning speed. The relative difference between Cogset and Hadoop is larger here than for *Grep*—this is primarily because of CPU bottlenecks in Hadoop that become more prominent when the map function emits more records. By applying the same optimizations as for *Grep*, the performance of Hadoop improves significantly.

Parallel databases sort the input data by *pageRank* prior to executing this task. If the same pre-processing were made before executing the MapReduce version, it would be straightforward to write a custom input format that reads until encountering a specified threshold *pageRank* value. We also measured the performance when using such an approach, thus mimicking the effects of accessing the input data using an index. This resulted in approximately 50% shorter execution times both for Hadoop and Cogset, as shown in Figure 5.3.

5.5.3 Aggregate Results

A UserVisit record can comprise more than 200 bytes, but *Aggregate* only accesses the *sourceIP* and *adRevenue* fields of each record, for a total of 20 bytes. *Aggregate* thus highlights the situational advantages of column-oriented storage. For row-oriented databases, and for MapReduce, the overriding cost is to scan the entire input data set. The intermediate data set is much smaller than the input and can quickly be aggregated in the reduce phase. Again, Hadoop suffers from poor scanning performance, while Cogset is more than twice as fast. Our optimizations for Hadoop are able to close some of the gap. The execution times for *Aggregate* are shown in Figure 5.4.

The MR/DB benchmark also features a second variant of Aggregate that groups the output by a substring of the source IP, resulting in much fewer groups (i.e. fewer unique keys to be reduced). Both Hadoop and Cogset are insensitive to this change, and the results are practically identical using this variant, so we do not include them here.

5.5.4 Join Results

Join illustrates the benefits of index structures. MapReduce scans through 30 years worth of (randomly generated) user visit log records to extract the records from one specific week. Understandably, the cost of selecting the correct subset of the records using a brute force scan overshadows the cost of performing the actual join. For this task, our optimizations for Hadoop are very effective and make Hadoop perform almost as well as Cogset, as shown in Figure 5.5. Unoptimized, Hadoop performs poorly, with similar execution times as for *Aggregate*, which also requires scanning the entire set of user visits.

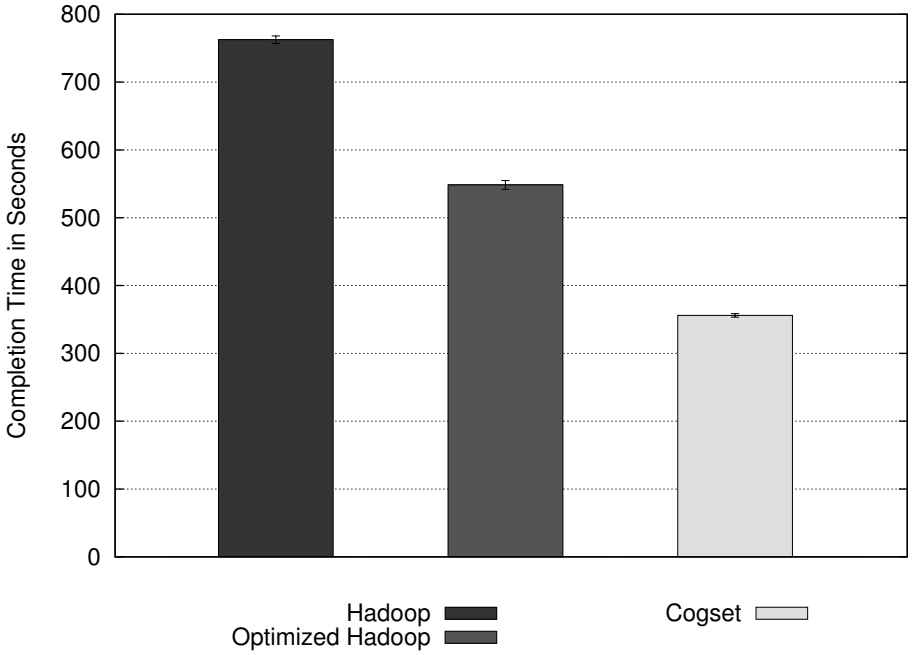


Figure 5.1. Grep execution times in seconds.

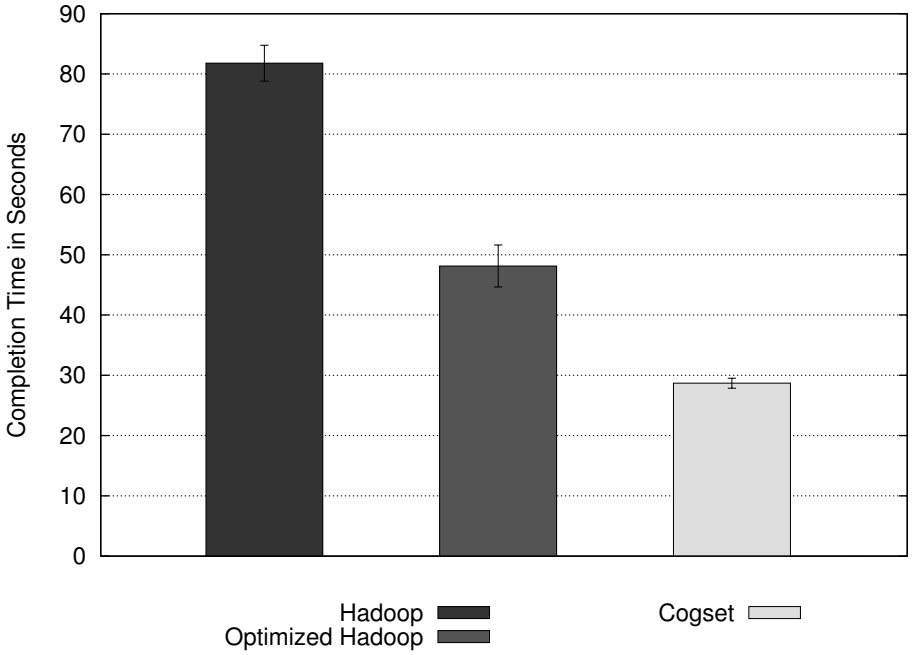


Figure 5.2. Select execution times in seconds.

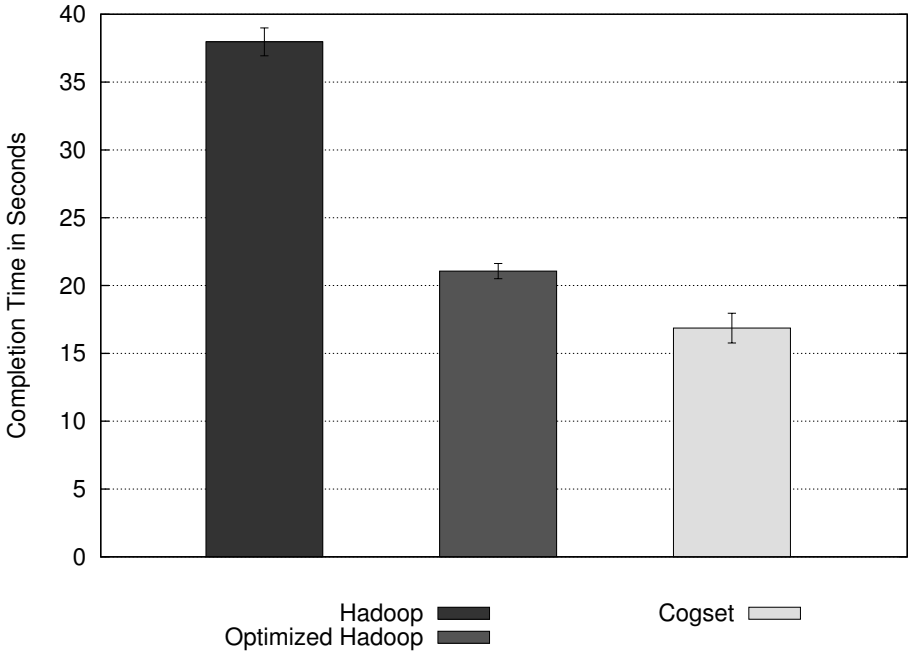


Figure 5.3. Select execution times in seconds, with an emulated index.

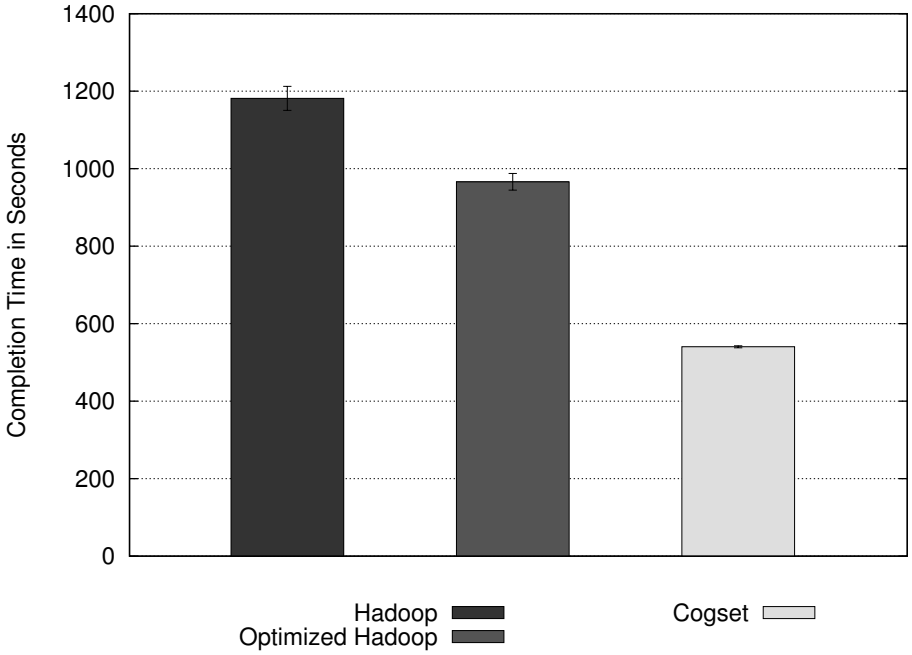


Figure 5.4. Aggregate execution times in seconds.

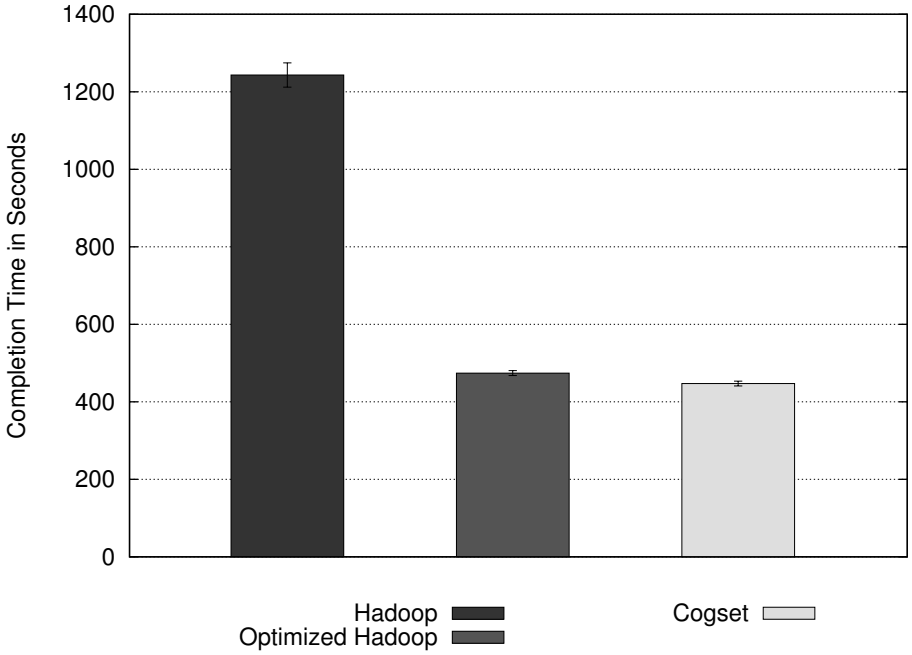


Figure 5.5. Join execution times in seconds.

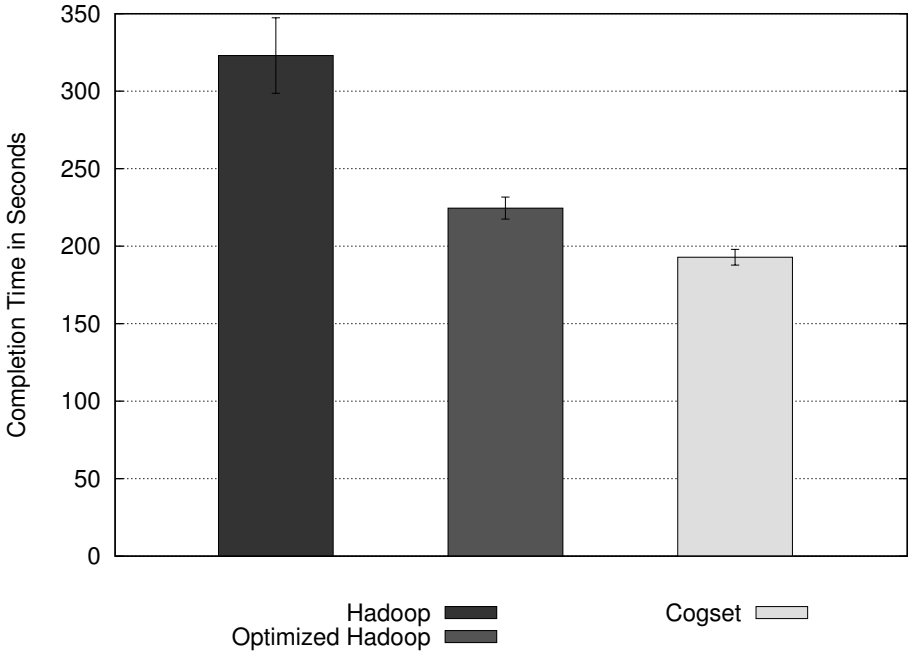


Figure 5.6. Join execution times in seconds, with an emulated index.

The log of user visits would realistically be stored in multiple files, with date ranges encoded into their file names. This "natural index" would allow MapReduce to only process input files that overlap with the desired input date range. Again, we would like to isolate the performance gains we could achieve using indexes, when the input format only supplies records from the correct date range. We emulate this by pre-filtering the UserVisits files, creating a new set of files to be used as input. The performance of this approach is shown in Figure 5.6.

5.5.5 UDF Results

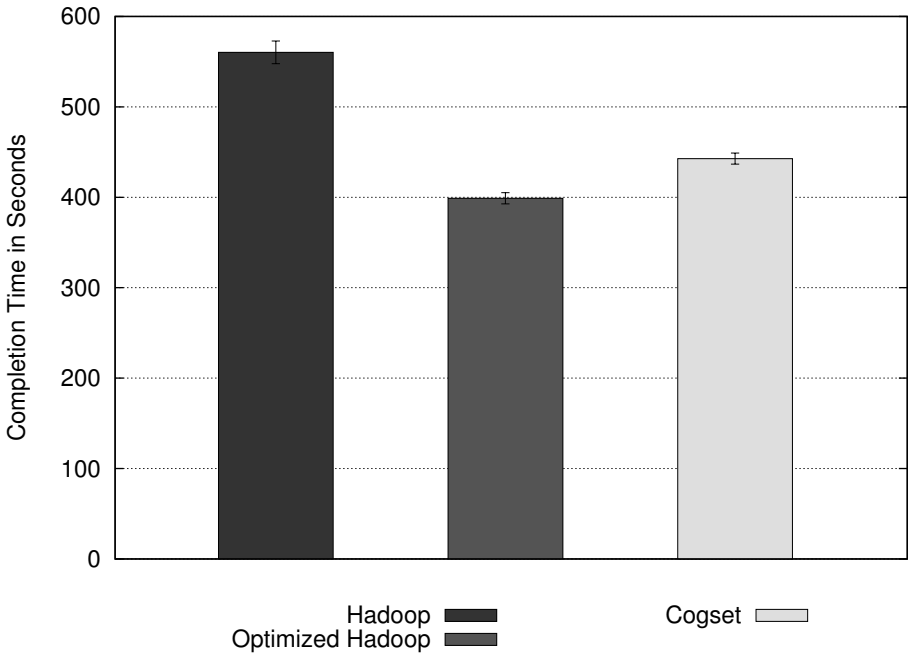


Figure 5.7. UDF execution times in seconds.

UDF employs a CPU-intensive map function that extracts hyperlinks using a regular expression. In line with previous results, Cogset completes the map phase faster than Hadoop. In return, Cogset takes slightly longer than the optimized version of Hadoop for sorting and reducing. This is because Hadoop starts the reduce tasks before all map tasks are finished, pre-fetching and pre-sorting map output partitions. This exploits some of the idle capacity that is available at the end of the map phase. Cogset executes each traversal independently, and has no MapReduce-specific optimizations, so there is no overlap between the map phase and the reduce phase. Unlike Hadoop, Cogset also stores map output reliably by replicating it. The execution times for *UDF* are shown in Figure 5.7.

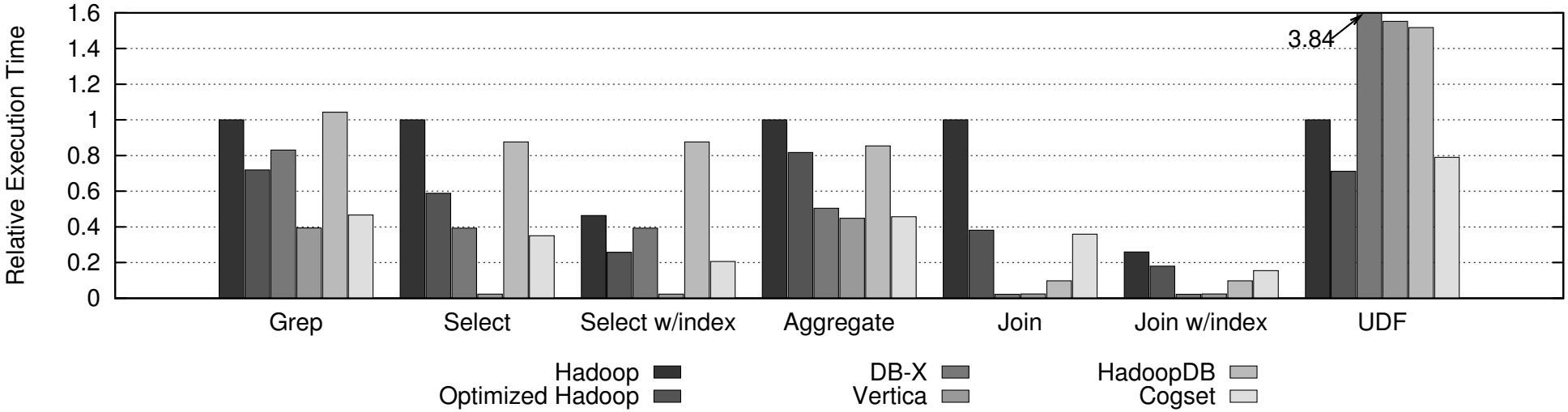


Figure 5.8. Relative execution times for the MR/DB benchmark.

5.5.6 Relative Performance

Figure 5.8 shows the *relative* performance of all systems evaluated using the MR/DB benchmark. All numbers are relative to Hadoop's performance; the numbers for Vertica and DB-X are from the original benchmark paper, and the numbers for HadoopDB are from the 10-node experiments in the paper introducing HadoopDB [54]. (They did not include measurements for a 25-node deployment). The usual pitfalls of extrapolating or indirectly comparing performance measurements apply; in particular, there could be elements of the experimental setup that emphasize different strengths and weaknesses in the various systems. For example, our cluster has relatively fast disks compared to the single-core CPU speed, which might accentuate CPU bottlenecks that would otherwise go unnoticed.

The relative performance results show that Cogset's performance rivals that of the parallel databases for the *Grep*, *Select*, *Aggregate* and *UDF* tasks, with the exception of Vertica's *Select* performance, which is considerably higher than any other system, presumably due to its data compression features[2].

For *Join*, the parallel databases benefit both from index structures and from the partitioning that was done at load time, which allows a local hash join. The join algorithm used in the corresponding MapReduce implementation is less efficient, and needlessly copies the entire UserVisits data set from the first to the second MapReduce job, rather than discarding irrelevant attributes. The biggest overhead still stems from scanning the entire UserVisits data set in order to extract a very narrow date range. We emulated the effect of an index on *visitDate* by prefiltering the data set: these results are shown in Figure 5.6 and labeled as "Join w/index" in Figure 5.8.

Implementing a join without first repartitioning the input data sets is somewhat impractical using Hadoop, but possible. Reduce-only jobs are not supported, so to avoid repartitioning, a map-only job must be used. A join could then be implemented by using a custom input format that merges records from multiple input files.

If the MapReduce interface is bypassed, Cogset can perform a local hash join using custom traversals. A straightforward implementation using one traversal to perform the hash join and a second to group and aggregate the output from the join by *sourceIP* resulted in execution times less than 90 seconds—almost four times as fast as the MapReduce version executed using Hadoop (see Figure 5.5), and faster than HadoopDB in relative terms. Thus, a join implemented using Cogset's core interfaces can approach the performance of parallel databases.

5.6 Analyzing and Optimizing Hadoop

The large gap in performance between Hadoop and Cogset prompted us to investigate the internals of Hadoop more closely. This was an attempt to determine whether the poor performance was due to shortcomings in the Hadoop implementation, suboptimal configuration of Hadoop, or due to more fundamental limitations intrinsic to the MapReduce programming model. This section gives a detailed account of how we analyzed Hadoop to answer these questions, and how we devised two specific optimizations that address some of the implementation weaknesses that we uncovered. As noted, the benchmark results obtained by using the

[2]The MR/DB paper is also unclear about the details of the *Select* task: it states that 36,000 records are selected per node, but the published code generates 4.1 million matching records per node.

optimizations described here were labeled as "Optimized Hadoop" in the previous section, and these optimizations markedly improved Hadoop's performance.

5.6.1 Task Scheduling

Our results show that Hadoop generally scans input data at a much lower rate than the peak disk read bandwidth. To investigate why, we first examined the *Grep* task, where Cogset scans data more than twice as fast as Hadoop, proving that there is substantial room for improvement. Using *vmstat -p*, we monitored the rate at which data was read from disk on various nodes. Surprisingly, disks frequently went idle for several seconds, even though *Grep* ought to be disk-bound.

To account for this idle time, we examined the high-level statistics maintained by the Hadoop job-tracker. It reported an average execution time for each map task close to 8 seconds for all *Grep* trials, while the total duration of the map phase was approximately 750 seconds. By default, Hadoop allows concurrent execution of up to 2 map tasks on each node, for a total of 50 execution slots in our case. Summing up the execution time for all of the 3777 map tasks indicates that their total execution time was around 600 seconds per slot. In other words, each of the execution slots was idle for around 20% of the time.

To investigate further, we parsed the verbose job history logs and extracted the start- and end times of each individual task, building up a detailed timeline of each trial run. We discovered that there was generally a delay of up to 3 seconds from the time one task was completed until another task was scheduled to execute in its slot.

An examination of the task-tracker code revealed why. Each task-tracker manages the execution of tasks on a specific node, but must communicate with the central job-tracker to obtain scheduling decisions. This communication is piggybacked on a periodic heartbeat RPC call from the task-tracker to the job-tracker; instructions about new tasks to execute are included in the heartbeat response. The task-tracker does not generate a heartbeat immediately upon task completion. Instead, it waits until the next regularly scheduled heartbeat, which occurs every 3 seconds. Therefore, completed tasks are only reported every 3 seconds, and if all of the tasks executing on a node finish within the same 3 second window, the node will go completely idle until the next heartbeat.

With an average of 1.5 seconds left of the heartbeat interval, and an average task execution time of 8 seconds, the probability of an execution slot being idle is $(1.5/9.5) = 15.8\%$. Each task-tracker has two execution slots, so with independent distributions of tasks start times, nodes should be completely idle around $(1.5/9.5)^2 = 1.66\%$ of the time. In reality, tasks only start executing at one of the 3 second marks, immediately after a heartbeat response is received, and thus tend to complete at similar times as well. This increases the probability of all execution slots going idle towards the end of a heartbeat interval. Our timeline analysis of the Grep trials revealed that each node was completely idle around 5-6% of the time during the early parts of the map phase; this *excludes* the idle time that results from the barrier synchronization point at the end of the map phase, which accounted for an additional 3-4% of the time in our experiments. The latter idle time is unavoidable, but can be minimized by using more fine-grained tasks—the backup task mechanism also helps, by avoiding delays caused by abnormally slow stragglers.

Allowing nodes to idle for extended periods of time is wasteful for a benchmark that ought to be disk-bound. The only way to make sure the disk is fully utilized, given the default heartbeat

interval, is to add more concurrent map tasks. Unfortunately, this leads to a less optimal access pattern, since multiple readers will then concurrently access separate regions of the disk. The heartbeat interval is automatically determined by Hadoop based on cluster size (and may be even longer than 3 seconds for larger clusters), so we had to patch two lines of code in the task-tracker to make it configurable. We then reduced the heartbeat interval to 50 milliseconds, to ensure that the job-tracker would react promptly to completed tasks. For *Grep*, this simple change resulted in an immediate improvement in Hadoop's performance by about 15%. A better solution to this problem would be to send a heartbeat message immediately upon every task completion—this would require a few additional changes to the code. If there are compelling reasons for communicating infrequently with the job-tracker, an alternative might be to add a short queue in each task-tracker holding tasks that are scheduled for execution, effectively making scheduling decisions slightly in advance to avoid idle nodes.

The delayed reporting of completed tasks has an even more dramatic performance impact when using Hadoop's default configuration. The MR/DB benchmark uses an HDFS block size of 256MB, which is four times larger than the default. Hadoop schedules one map task for each HDFS input block, so with a smaller block size, each task completes faster, and the probability of a node going idle during a heartbeat interval increases. In the extreme case, when Hadoop is used to process a set of very small files (each of which is stored in a separate block), the overhead is excessive: no matter how fast a single task completes, each node can only execute two tasks during each 3-second heartbeat interval. When executing *Grep* using the default HDFS block size, nodes were completely idle for around 34% of the time, resulting in an overall slowdown of about 80%.

5.6.2 Multi-Core Optimizations

Despite reconfiguring the heartbeat interval to reduce idle time, the performance gap between Hadoop and Cogset remained relatively large: Cogset still executed *Grep* roughly twice as fast as Hadoop. Now that map tasks were executing more or less continuously, we turned our attention to their internals. Centering around the flow of data, the activities of a map task can be broken down into the following distinct steps:

1. Reading raw data from HDFS over TCP.
2. Verifying checksums embedded in the data to detect corrupted data blocks.
3. Parsing the raw bytes into records using the specified input format.
4. Invoking the map function once per record.
5. (Optionally) pre-aggregating emitted records using the specified combiner.
6. Collecting all emitted output records and partitioning them by intermediate key.
7. Partially sorting[3] and flushing the collected output to local disk.

With the exception of a separate thread for spilling records to disk, Hadoop employs a single thread to drive the execution of all 7 steps, and is thus limited to a single CPU core. While

[3]Hadoop partially sorts map outputs to speed up subsequent merging by the reduce tasks.

developing Cogset, we experienced that data processing quickly becomes CPU-bound when records are small and must be parsed into numerous Java objects to be processed by user code. Furthermore, the presence of multiple user-definable hooks in the critical path makes hot-spots harder to predict and identify. Therefore, a single-threaded approach is vulnerable to CPU bottlenecks that in turn prevent maximum disk utilization. A better design would be to structure map tasks as a sequence of pipelined stages that execute concurrently in separate threads.

To quantify the relative cost of the individual steps outlined above, we first implemented a simple single-threaded program that reads (and discards) data from HDFS at the maximum possible speed. Surprisingly, we found that HDFS throughput peaked at about 131 MB/s even when reading locally[4] from a warm file system cache—this is significantly less than our disk bandwidth. If we also parsed the data to identify line breaks (using Hadoop library code), the program could only achieve 85 MB/s, which was actually slower than reading from a cold cache (i.e. from disk) without line parsing. In other words, a single thread reading from HDFS quickly becomes CPU-bound. To make the program disk-bound, checksum verification had to be disabled. Even then, line parsing could not be done by the same thread without limiting throughput. Note that this limitation was imposed by the program structure and not by the hardware: our machines have 8 cores and plenty of cycles to spare, but our single-threaded program could only utilize one core.

We concluded that in order to maximize throughput from HDFS, multiple threads must be employed, starting with step 2 above. A single thread must be dedicated to reading the incoming data from the TCP socket and then immediately hand the data off to other threads that perform the remaining steps. Unfortunately, the second step, checksum verification, is closely integrated into the HDFS client code, with no hooks for customization by user-supplied code. On the other hand, it seems clear that such verification could be done in a multi-threaded fashion without sacrificing throughput, as long as there is available CPU capacity: simply break the input stream into chunks, and verify them in parallel using a separate pool of threads. Therefore, we decided to disable checksum verification in order to discover what throughput could be achieved by such a re-engineered version of HDFS. Even if users had to choose one over the other, we expect many would prefer improved performance over checksum verification, since the latter can also be performed periodically by the HDFS data nodes, in periods of light load.

To address the remaining CPU-intensive steps, we also needed to perform record parsing in parallel. To achieve this, we implemented a custom input format for multi-threaded parsing of text files. The input format splits its input into byte ranges, which are delegated to a separate pool of threads for parsing. Each parsing-thread locates all line breaks in one byte range, and also peeks at the beginning of the next range in order to identify lines that cross range boundaries. The extracted lines are split into key/value pairs and then passed directly to the mapper and combiner functions; this is an improvised way of including steps 4-5 in our input format implementation. A similar effect could be achieved by also writing a custom map runner, but that would require additional user code. Instead, we use the default map runner with an identity map function as the mechanism for relaying output records from our input format to the output collector in step 6. To summarize, our custom multi-threaded input format dedicates a single thread to executing step 1 above, skips step 2, and allows steps 3-5 to execute in parallel using multiple threads.

[4]In this context, reading locally means reading from an HDFS data node running on the same machine.

Using our patched task-tracker, configured with a 50 millisecond heartbeat interval and our multi-threaded input format, we were able to improve Hadoop's performance significantly for all benchmark tasks, as shown in Section 5.5.

Grep, *Select*, *Join*, and *UDF* all employ CPU-intensive map functions and benefit from multi-threaded mapping. Additionally, *Aggregate* benefits from pre-aggregation in the map phase and performed significantly better with multi-threaded evaluation of combiner functions. Initially, we just implemented multi-threaded mapping and kept the existing code for combining records; this resulted in no improvement for *Aggregate*, since combining remained a single-threaded bottleneck. This underlines the general importance of parallelizing *all* steps of a processing pipeline to avoid bottlenecks.

5.7 Summary

In this chapter, we evaluated Cogset's performance by comparing it directly to Hadoop, and indirectly to parallel databases, using the established MR/DB benchmark. Our results show that Cogset performs much better than Hadoop on this benchmark, being more than twice as fast to complete several of the benchmark tasks. When investigating the underlying reasons for this gap in performance, we also found ways to improve Hadoop's performance by adopting some of the same implementation techniques used in Cogset. We patched Hadoop's task scheduling algorithm to prevent individual nodes from going idle, ensuring a more efficient I/O access pattern, and we adopted Cogset's multi-threaded approach to record parsing to improve Hadoop's performance on multi-core CPUs.

Our results support our thesis that static routing may underpin a high-performance MapReduce engine. Cogset's design was shaped by static routing as its guiding principle, which resulted in a generic processing abstraction—a traversal—that has proved to be highly efficient. During a traversal, the static assignment of data partitions to nodes allows each node to read data continuously from disk, with a minimum of coordination between nodes. By routing output data directly to other nodes, rather than storing it temporarily on local disk, the I/O access pattern is only minimally disrupted by writes and by random-access seeking.

Chapter 6

Higher-level Abstractions

MapReduce was originally motivated by a desire to process very large data sets in an efficient, scalable, fault-tolerant, and load-balanced manner in a distributed shared-nothing environment. The widespread adoption of MapReduce can be attributed to the simplicity of its programming model, where user-defined functions are executed in a generic framework that automates non-functional concerns.

With Cogset, we explore the thesis that a high-performance MapReduce engine can be based on static routing. Cogset was built from the ground up with static routing as its guiding design choice. The previous chapters demonstrate that Cogset meets the requirements commonly expected of a MapReduce engine, as outlined in Section 1.4, and is capable of efficiently executing benchmark applications developed for Hadoop.

A requirement that remains to be discussed is extensibility; the ability to layer higher-level abstractions on top of Cogset's core engine. One feasible approach would be to substitute Cogset for Hadoop in existing systems such as Pig [16] and Hive [41], which directly employ Hadoop as an underlying execution engine. However, Cogset also has a lower-level and more generic core interface, on which its MapReduce support is implemented as a thin layer. This core interface was designed to facilitate the construction of higher-level abstractions, avoiding any limitations imposed by going through the MapReduce interface.

To demonstrate the generality and extensibility of Cogset, we have built two systems that use Cogset's core engine, but provide alternative programming interfaces: Oivos and Update Maps. In this chapter, we present these systems, explain the semantics of their programming interface, and detail how they are built as separate layers on top of Cogset's core.

When discussing these systems, we show how they are representative higher-level abstractions by drawing parallels to other systems, and explain the complications of layering these abstractions on top of a traditional MapReduce engine. This puts Cogset into a new perspective, demonstrating its applicability not just as an efficient MapReduce engine, but also as a viable platform for a range of higher-level abstractions.

6.1 Oivos

As previously remarked in Section 2.5, certain limitations are inherent to the MapReduce programming model. Many computations cannot be expressed using a single MapReduce pass; for example, if they need to repartition the data set more than once. Such computations must be broken up into a number of separate MapReduce passes.

One of the main strengths of MapReduce is that non-functional concerns are handled by the execution engine, allowing the programmer to focus on the application logic. But when a computation is performed using multiple MapReduce passes, several non-functional issues concerning scheduling, synchronization, and fault tolerance resurface. For example, some process external to the MapReduce implementation must monitor the status and progress of passes, determining if and when to re-execute a failed pass or start the next one. The programmer must also determine a valid execution order for the MapReduce passes, considering that some of the data might be out of date and need to be regenerated. In the event of potential parallelism, arranging for multiple passes to execute concurrently is another task left to the programmer. Even if an optimal scheduling of passes is achieved, a typical MapReduce implementation will introduce a barrier synchronization point at the end of each pass, requiring every reduce task in one pass to complete before any of the map tasks in the next pass can start. This restriction reduces the potential parallelism in multi-pass MapReduce computations.

To address these limitations, we created Oivos—a system that allows computations spanning a heterogeneous collection of data sets to be expressed at a high abstraction level, in a declarative functional style familiar to many programmers. Oivos derives its name from the Sami word for the source of a river, and revolves around the composition of workflows. Applications employ abstractions provided by Oivos to programmatically specify how new data sets may be derived from existing ones, supplying user-defined functions as parameters to the various core operators, similar to how MapReduce applications specify map and reduce functions.

Unlike MapReduce, Oivos supports programs with multiple input data sets, from which any number of intermediate or output data sets may be derived. In general, the resulting data dependencies may form an arbitrary acyclic graph. Upon request, Oivos automatically determines how to materialize a derived data set, performing what may amount to multiple MapReduce passes. There is no need to implement a computation as a collection of small programs whose execution must be coordinated externally; a single program specifies everything, even if multiple data sets are involved.

Originally, we designed and implemented Oivos as a stand-alone system that encompassed a simple distributed file system, as well as a distributed task scheduling facility. One shortcoming we experienced using this approach was that the strict logical separation of data storage and task scheduling tended to restrict performance, due to the resulting poor data locality. This experience partly motivated our design of Cogset, where we opted for a closer integration between storage and scheduling systems.

In its second incarnation, which we describe here, Oivos is built as a layer on top of Cogset's core engine. While Cogset provides the required infrastructure for distributed storage and processing, Oivos adds the programmatic interfaces for conveniently manipulating multiple interdependent data sets. This serves as an example of how to build powerful higher-level abstractions using Cogset's core engine as a foundation.

Oivos facilitates the expression of complex computations in a more flexible and intuitive way than as a series of MapReduce passes. As with MapReduce computations, the program is compiled into a series of traversals, the fundamental processing mechanism provided by Cogset. This substantiates our idea that traversals may serve as a suitable building block for multiple higher-level abstractions.

Externally, Oivos has many similarities with other workflow composition languages such as DryadLINQ [42] and FlumeJava [4] (see Section 2.5 for details). Like MapReduce, these languages are embedded into a general purpose programming language, in the form of a library.

Oivos is implemented similarly, as a library on top of Cogset's core, and implementations of other workflow composition languages could potentially be layered on top of Cogset's core in an equivalent manner.

In the following, we first describe the high-level programming interface offered by Oivos, and the semantics of the core operators that it provides. We then present an example Oivos program, and explain how Oivos programs are compiled into a series of Cogset traversals.

6.1.1 The Dataset abstraction

The primary abstraction offered by Oivos is a *Dataset*, which represents a homogeneous set of records, and corresponds directly to a Cogset data set. Oivos programs generally revolve around the manipulation of such *Dataset* objects. An excerpt of the abstract *Dataset* class is shown in Figure 6.1. As in previous chapters, we omit non-essential code details such as access modifiers and exception types, to improve readability.

```
abstract class Dataset<A> implements Serializable
{
    abstract Dataset<A> setKeyFunction(KeyFunction<A, ?> keyFunction);

    abstract Dataset<A> setKey(String keyField);

    abstract Dataset<A> setName(String name);

    abstract Dataset<A> setFormat(RecordFormat<A> format);

    abstract <B> Dataset<B> map(Mapper<A, B> mapper);

    abstract Dataset<A> sort(KeyFunction<A, ?> keyFunction);

    abstract <B> Dataset<B> reduce(Reducer<A, B> reducer);

    abstract Dataset<A> combine(Combiner<A> combiner);

    abstract <B, C> Dataset<C> merge(Dataset<B> x, Merger<A, B, C> merger);

    abstract Dataset<A> filter(Filter<A> filter);

    final <B, C> Dataset<C> mapAndReduce(Mapper<A, B> mapper,
                                         KeyFunction<B, ?> intKeyFunction,
                                         Reducer<B, C> reducer)
    {
        return map(mapper).setKeyFunction(intKeyFunction).reduce(reducer);
    }
}
```

Figure 6.1. Oivos interfaces for manipulation of data sets.

Data sets may either be declared as input data sets, or they may be *derived* by applying operators to other data sets. The data set operators are generally parametrized by user-specified functions and may thus be viewed as higher-order functions that transform data sets into new data sets. By invoking various methods on *Dataset* objects, corresponding to the available operators, derived data sets are declared.

Oivos programs are declarative, and do not specify an order of execution; they merely declare all data sets, specifying how they relate to one another. A computation is initiated by specifying one or more desired output data sets; Oivos will automatically determine which traversals to execute in order to materialize those data sets.

Oivos thus employs *lazy evaluation*: a *Dataset* object specifies how to potentially produce a specific data set by processing existing data sets, but no such processing will actually occur until a materialized version of the data set is requested. To materialize a data set, the *Dataset* is passed to the *materialize* method of the *DatasetFactory* class, shown in Figure 6.2. There is also an overloaded version of the *materialize* method, used to materialize a collection of data sets. A *DatasetFactory* instance is coupled upon construction to a *Cogset* instance, which will be used to execute the required traversals when materializing data sets.

```
class DatasetFactory
{
    DatasetFactory(Cogset cogset);

    <A> Dataset<A> inputDataset(String name, Class<A> recordClass,
                                KeyFunction<A, ?> keyFunction);

    <A> Dataset<A> inputDataset(String name, Class<A> recordClass);

    void materialize(Dataset<?> dataset);

    void materialize(Collection<Dataset<?>> datasets);
}
```

Figure 6.2. Oivos interfaces for declaring and materializing data sets.

The *Dataset* class is parametrized with the record class of the underlying Cogset data set. User-defined functions that operate on records are similarly parametrized with the relevant record types. The use of Java generics thus aids in type-checking Oivos programs. For example, an attempt to apply a mapper function to an incompatible data set will trigger a compile-time error from the Java compiler. The need for explicit run-time type checking and casting is also greatly reduced.

Each data set also has an associated *key function*, as described in Section 4.4, used by Oivos for hash partitioning of the data set. The key function of a data set may be set using the *setKeyFunction* method. An alternative method, *setKey*, can be used for the common case where the desired key is a single field in the record class. This automatically constructs an appropriate key function using the Java reflection APIs. There are also methods to set the name of a derived data set, or its record formatter, as described in Section 4.2.

The remaining methods of the *Dataset* class correspond to the various data set operators, and are covered in detail in the next section. They generally accept a user-defined function implementing one of the interfaces in Figure 6.3, and return new *Dataset* objects.

To declare input data sets, Oivos programs use the *inputDataSet* method of the *DatasetFactory* class. There are various overloaded versions of this method, due to multiple optional parameters. The only required parameters are the name of the data set and its record class. Figure 6.2 also shows one overloaded variant of the method, to illustrate that the key function of the input data set can be specified explicitly. If the key function is omitted, Oivos will for convenience

```
interface Output<A>
{
    void emit(A record);
}

interface Mapper<A, B> extends Serializable
{
    void map(A record, Output<B> output);
}

interface Reducer<A, B> extends Serializable
{
    void reduce(Iterator<A> records, Output<B> output);
}

interface Combiner<A> extends Serializable
{
    A combine(A a, A b);
}

interface Merger<A, B, C> extends Serializable
{
    void merge(A left, B right, Output<C> output);
}

interface Filter<A> extends Serializable
{
    boolean filter(A record);
}
```

Figure 6.3. Function interfaces associated with Oivos operators.

attempt to infer a suitable key function for the data set using various heuristics, such as using reflection to find the first public field in the record class to implement the *Comparable* interface. A suitable record formatter will also be inferred in many cases using similar heuristics, for example by checking if the record class implements Hadoop's *Writable* interface.

There are some advantages to specifying Oivos programs through a programmatic interface, as opposed to inventing a new domain-specific language. Since there are few limitations to what a user-specified function such as a mapper could potentially compute, a domain-specific data set manipulation language would in reality have to include most if not all facilities of a general-purpose programming language. Our approach allows for seamless integration of Oivos programs with an existing code base. Functionality like MD5 check-summing, URL parsing, and date formatting can be implemented simply by invoking the standard Java libraries. Another useful property is that *Dataset* objects fully encapsulate the specification of how to produce a data set; as such, the principle of lazy evaluation can be used to defer the materialization until it is necessary. If an existing component A implements the logic required to produce a data set that another component B needs, A can simply construct an appropriate *Dataset* object that specifies how to produce the data set, and pass the object to B, in serialized form over a network connection if need be. *Dataset* objects may even be stored on disk, for later materialization.

6.1.2 Oivos Operators

In summary, Oivos programs declare data sets that may be transformed into other data sets using a number of operators. We now present the most important operators in Oivos, and describe their exact semantics. This includes the traditional *map* and *reduce* operators, which can be used to implement a traditional MapReduce job. (In fact, Figure 6.1 shows how Oivos provides a generic *mapAndReduce* function to do just that.) Crucially, there is also a *merge* operator that can be used to perform relational joins.

Map The *map* operator applies a user-specified mapper function to each record in a data set. The mapper function may emit zero or more output records per input record. The result is a new data set containing all emitted records. The record and key types of the output data set may differ from those of the input data set.

Sort The *sort* operator sorts the records within each partition of a data set according to the user-specified key. This operator is applied implicitly by other operators that require sorted input, and may be applied explicitly as well. The result is a new data set with the same record type as the input data set, although the keys of the input and output tables may differ.

Reduce The *reduce* operator applies a user-specified reducer function once per unique key in a data set; the reducer function accepts a key and an iterator that can be used to iterate over all records with that specific key. Like mapper functions, reducer functions may emit a varying number of output records; the result is a new data set whose record and key types may differ from the input table.

Combine The *combine* operator is used to combine all records with equal keys into a single record. The user specifies a combiner function that accepts two records and returns one; it is repeatedly applied to pairs of records with equal keys until a single record remains per unique key. The combiner function must be associative and commutative, such that the resulting record will be identical regardless of the order in which the function is applied to record pairs. Additionally it may not change the record keys. The result of the combine operator is a new data set that has the same record and key types as the data set, with exactly one record per unique key. This operator is similar to the family of higher-order *fold* functions known from functional languages, except that it does not imply a particular order of evaluation.

Unlike reducers, combiners cannot change record keys (a restriction enforced at runtime), so it is possible to preserve the ordering of a sequence of records while combining them. This is useful, since it allows an optimization for upstream aggregation to "lift" the combine operation, moving it upstream past a merge operation without disrupting the merge.

Like its MapReduce counterpart, the Oivos *combine* operator may be used for partial upstream aggregation, but the differing signatures of the combiner functions makes a subtle difference. MapReduce combiners have the same function signature as reducers and must rely on sorted input. Oivos combiners only require a pair of equal-keyed records and are thus better suited for hash-based aggregation. Technically, hash-based aggregation could be implemented by a MapReduce engine by passing record pairs to the combiner as a

sorted sequence of length 2, but additional run-time checks would have to be added to ensure that the combiner emits exactly one record of the correct type.

Merge The *merge* operator is a binary operator that merges two sorted input data sets with the same key type into one new data set, applying a user-specified merger function for each unique key. The merger function accepts two records and may emit zero or more output records. In the case of keys that appear in both input data sets, the arguments to the merger function are one record from each of the input data sets. For keys that only appear in one of the input data sets, a *null* record is passed as the "missing" argument. The merger function may thus implement outer or inner joins as desired. If the same key occurs multiple times in one of the input data sets, the merger function will be invoked once for each occurrence. If the same key occurs multiple times in both of the input data sets, a run-time error is triggered. The result of a merge is a new data set, whose record and key types may differ from those of the input tables. For simplicity, we chose to limit this operator to working on two input data sets, since a series of binary merges can implement a multi-way merge, for more complex joins.

Filter The *filter* operator filters out certain records of a data set, according to a logical predicate specified by a user-defined function. This is essentially a special case of the *map* operator, but there are some advantages to including it as a separate operator. Explicitly using the *filter* operator may make a program clearer and less error-prone, and when a data set is filtered, the ordering of the remaining records is preserved. This is an important property that cannot be inferred of mapper functions in general, and allows a filter operation to be "lifted" in the same way as a combine operation, for upstream aggregation.

The data set operators described here are inherently well-suited for parallel evaluation. This is because the user-specified functions are either applied once per input record (e.g., mapper functions) or once per unique key (e.g., reducer or combiner functions). In the former case, the function has no dependencies on other records and may thus be evaluated in parallel for all records. Similarly, reducer and combiner functions may be evaluated in parallel for all unique keys. In practice, the level of parallelism is determined by the number of Cogset nodes. Data sets are materialized by executing traversals, during which all nodes process separate partitions of the data sets, in parallel.

Note that using the combine operator when possible is generally preferable to using the reduce operator; this is because reducer functions require a collection of all records with equal keys in order to evaluate. In contrast, a combiner function only requires a pair of records with equal keys and may thus be evaluated earlier (using the principle of upstream evaluation [38]). In short, the combine operator is inherently more parallelizable than the reduce operator.

6.1.3 Example Oivos Application

In the initial publication on MapReduce, Dean and Ghemawat reported that Google's web indexing code was refactored from a number of ad hoc distributed processing passes to using a sequence of 5 to 10 MapReduce passes [6]. Web indexing is thus a representative example of computations that must be implemented as multiple MapReduce passes. In this section, we develop an example Oivos application that constructs a web index, to demonstrate how a real-life application is structured using Oivos. Specifically, our example application will:

- Process a collection of downloaded web pages (HTML documents). In a real-life scenario, this would typically be a batch of documents provided by a separate web crawler component.
- For each document, parse the HTML code to extract all index terms and hyperlinks. Terms occurring in anchor texts will be included as index terms for the documents they link to.
- Build an *inverted index* that maps each index term to the set of documents in which it occurs.
- Merge the new inverted index with an existing inverted index to form a single, new index. Incrementally merging inverted indexes is a common technique for minimizing indexing latency while bounding the number of separate indexes that must be searched in order to evaluate queries.

```
class Document implements Writable
{
    @DefaultKey
    String url;
    String html;
}

class Token implements Writable
{
    @DefaultKey
    String term;
    String url;
}

class IndexEntry implements Writable
{
    @DefaultKey
    String term;
    List<String> urls;
}
```

Figure 6.4. Record classes used in the web indexer example.

Hyperlinks are generally an important piece of input for web search engines, to improve the relevance of the search results [2]. In our example application, we wish to include the words used in anchor texts as index terms. Anchor texts are the segments of text that actually appear as clickable links in the web browser; if a page A links to a page B using the text "Main Menu", then the words "Main" and "Menu" should be used as index terms for B.

The inverted index structure, which constitutes the final output of our example application, is a data set where each record associates one index term (typically an alphanumerical text string) with a list of document identifiers, such as URLs. This is called an *inverted* index because it inverts the initial representation of the data, where each document identifier is associated with a list of terms to be indexed.

Our example involves a number of data sets, whose record classes are listed in Figure 6.4. The classes all implement Hadoop's *Writable* interface, so the default *WritableRecordFormat*

provided by Cogset can be used, as outlined in Section 4.2. The classes also include constructors to initialize the various record fields, and methods specified by the *Writable* interface to write or read the fields to or from a data stream. In the figure, we have omitted the implementation of these methods, since they are straightforward. The *@DefaultKey* annotations are inspected programmatically using reflection, and instructs Oivos to use the respective annotated fields as record keys, unless another key function is specified explicitly.

```
static void oivosExample(Cogset cogset)
{
    Dataset<IndexEntry> index, tmpIndex, newIndex;
    Dataset<Document> docs;
    Dataset<Token> tokens;

    DatasetFactory df = new DatasetFactory(cogset);
    index = df.inputDataset(``index'', IndexEntry.class);
    docs = df.inputDataset(``docs'', Document.class);

    tokens = docs.map(new Tokenizer());
    tmpIndex = tokens.reduce(new Indexer());
    newIndex = index.merge(tmpIndex, new IndexMerger());
    df.materialize(newIndex.setName(``newIndex''));
}
```

Figure 6.5. Main program of the web indexer example.

```
class Link
{
    String href;
    String anchor;
}

class Tokenizer implements Mapper<Document, Token>
{
    void map(Document document, Output<Token> output) {
        HTMLParser parser = new HTMLParser(document.html);
        for (String word: parser.parseWords()) {
            output.emit(new Token(document.url, word));
        }
        for (Link link: parser.parseLinks()) {
            for (String word: link.anchor.split(`` '')) {
                output.emit(new Token(link.href, word));
            }
        }
    }
}
```

Figure 6.6. Oivos mapper function to tokenize HTML documents.

The input to our computation is an existing inverted index, and a set of *Document* records. Each document should be parsed, extracting a set of *Token* records, which associate individual index terms with URLs. For the main text of a document, the extracted terms will be associated

```
class Indexer implements Reducer<Token, IndexEntry>
{
    void reduce(Iterator<Token> tokens, Output<IndexEntry> output) {
        Token first = tokens.next();
        IndexEntry entry = new IndexEntry(first.term, first.url);
        while (tokens.hasNext()) {
            entry.urls.add(tokens.next().url);
        }
        output.append(entry);
    }
}
```

Figure 6.7. Oivos reducer function to aggregate tokens into an inverted index.

```
class IndexMerger implements Merger<IndexEntry, IndexEntry, IndexEntry>
{
    void merge(IndexEntry left, IndexEntry right, Output<IndexEntry> output)
    {
        if (left == null) {
            output.append(right);
        } else if (right == null) {
            output.append(left);
        } else {
            IndexEntry entry = new IndexEntry(left.term);
            Set<String> urls = new TreeSet<String>(left.urls);
            urls.addAll(right.urls);
            entry.urls.addAll(urls);
            output.append(entry);
        }
    }
}
```

Figure 6.8. Oivos merger function to merge entries from two inverted indexes into one.

with the document's URL. For hyperlinks, the terms extracted from their anchor text will be associated with the URLs that they link to.

The *Token* records are grouped by URL and aggregated to form an inverted index by applying the *reduce* operator, performing a reduction that emits one *IndexEntry* record for each unique index term. This data set is merged with the existing inverted index, also comprised of *IndexEntry* records, using the *merge* operator.

Figure 6.5 shows the main program for this example, comprising just a few lines of code to declare the relationships between the various data sets. Figure 6.6 shows the accompanying *Tokenizer* class that implements the mapper function to extract tokens from documents, Figure 6.7 shows the *Indexer* class that implements the reducer function to aggregate tokens into an inverted index, and Figure 6.8 shows the *IndexMerger* class that implements the final merger function. We omit the hypothetical *HTMLParser* class used to parse HTML code, since the details of HTML parsing are tangential to our example. However, it should be noted that interfacing with a third party library for HTML parsing is a simple matter of invoking the library

from one of the user-defined functions used in the Oivos program. Similarly, the power of the Java standard library is readily available; for example, the *IndexMerger* class uses the standard *TreeSet* class to merge the occurrence lists associated with an index term.

6.1.4 Compiling Oivos Programs

The implementation of Oivos relies heavily on polymorphism. The abstract *Dataset* class has numerous subclasses, corresponding to the various ways in which new data sets can be derived. For example, the default implementation of the *map* method creates a new *MappedDataset* instance that holds a reference to the input data set and the mapper function to employ. To materialize a *MappedDataset*, the conceptual steps are simple: first materialize the input data set, and then execute a traversal to invoke the mapper function and generate the new data set.

In general, the various *Dataset* objects form a data dependency graph, constructed incrementally by the programmer when declaring derived data sets. A data set can thus be materialized by a recursive algorithm that first materializes its input data sets, and then performs a traversal to generate the new data set. However, Oivos makes a few optimizations to minimize the number of traversals, which leads to a slightly more complicated algorithm to compile the set of traversals to be executed.

One such optimization is automatic *function composition*. In certain cases, consecutive processing steps can be collapsed and performed as one, in a single traversal. A prime example is two (or more) consecutive map operations on the same data set. In a naive approach, this would lead to the creation of an intermediate data set to hold the records emitted by the first mapper function. A second traversal would then read the intermediate data set and evaluate the second mapper function, to produce the final data set. However, there is no need to repartition the intermediate data in this computation, so it would be possible to directly evaluate both mapper functions in a single pass over the input data set, using a single traversal.

Oivos implements a set of mechanisms to optimize cases like this, by automatically composing user-defined functions where possible. Figure 6.9 shows an example of how this works for consecutive map operations. As noted earlier, the default implementation of the *map* operation is to instantiate a new *MappedDataset* object to represent the new, derived data set. However, in the *MappedDataset* class, the *map* method is overridden with a special implementation. This implementation composes a new mapper function by instantiating a new *Mapper* object that wraps both of the mapper functions used by the two consecutive map operations. As shown in Figure 6.9, the *ComposedMapperAndMapper* class implements a mapper function that works by invoking two other mapper functions in sequence, passing the output from the first mapper function as input to the other.

Similar compositions such as reducers composed with subsequent mappers, and mappers composed with subsequent filters, are implemented in the same way, by overriding select methods to avoid the creation of superfluous intermediate data sets. This technique demonstrates that computations centered around user-defined functions may well have potential for optimization, even though the user-defined functions are opaque to the optimizer. This also highlights a potential advantage of using a higher-level abstraction to specify complex computations, as an alternative to multi-pass MapReduce computations. Making optimizations of this kind would be much harder, and likely a manual task, in the case of a computation implemented as a collection of individual MapReduce programs.

```
class ArrayListOutput<A> extends ArrayList<A> implements Output<A>
{
    void emit(A record)
    {
        super.add(record);
    }
}

class MappedOutput<A, B> implements Output<A>
{
    Mapper<A, B> mapper;
    Output<B> output;
    ArrayListOutput<B> tmpOutput = new ArrayListOutput<B>();

    void emit(A record)
    {
        tmpOutput.clear();
        mapper.map(record, tmpOutput);
        for (B x: tmpOutput) {
            output.emit(x);
        }
    }
}

class ComposedMapperAndMapper<A, B, C> implements Mapper<A, C>
{
    Mapper<A, B> mapper1;
    Mapper<B, C> mapper2;

    void map(A record, Output<C> output)
    {
        mapper1.map(record, new MappedOutput<B, C>(mapper2, output));
    }
}
```

Figure 6.9. Automatic function composition of two user-defined mapper functions.

6.2 Update Maps

MapReduce is well suited for batch-oriented processing, where a large number of related state changes are performed in bulk, as a single relatively time-consuming operation. This may yield excellent amortized performance, but many applications are designed with underlying assumptions or requirements that appear to preclude batch processing. Specifically, they assume or require that minor state changes can be applied synchronously, with low latency. Typically, these requirements motivate the use of databases as the underlying storage layer, even though a batch-oriented programming model such as MapReduce might otherwise be preferred, for example due to its scalability.

A popular abstraction for applications that require a synchronous storage interface are so-called *key/value databases* [5, 63, 64]. Like MapReduce, such databases model data as simple key/value pairs. Unlike MapReduce, a key/value database provides near constant-time operations to modify or delete the value associated with an individual key, and to add a new key/value

mapping. This flexibility is achieved by using on-disk index structures such as B-trees or hash tables.

The downside of a key/value database is that the underlying data structures tend to impose a seek-intensive I/O access pattern, which is inefficient on traditional storage media. In contrast, batch processing generally relies on large sequential I/O operations, achieving much higher I/O throughput. As such, the *amortized* cost of an individual state change can potentially be much lower if batch processing is employed.

By nature, some applications cannot employ batch processing, for example because they involve real-time interactions with users, which cannot be delayed. For other applications, the choice to use a key/value database may have been motivated by the convenience of a simple and intuitive programming interface. In the latter case, changing requirements with regards to load or scale can make the switch to a batch-oriented approach more attractive as the application evolves.

However, refactoring an existing application from using a database storage layer to using MapReduce or a similar batch-oriented infrastructure can be a challenge. With a synchronous key/value interface, modifications can be performed in an ad hoc manner, on any number of code paths. To efficiently utilize MapReduce for batch processing, a more systematic approach is needed, where a number of modifications must be accumulated asynchronously, until the entire batch of modifications is executed in bulk, as a MapReduce job.

In this section, we present the *update map* abstraction, which is designed to facilitate such transitions. It aims to combine the convenience of a key/value database with the performance of a batch-oriented approach. The update map interface resembles that of a key/value database, but its implementation can automatically accumulate updates and apply them in batches. Our design of the update map abstraction was motivated by a concrete effort to refactor a real-life web crawler application from relying extensively on a key/value database into using a more scalable batch-oriented approach.

In the following, we first present the update map interface, and present some simple examples to clarify its semantics. We then outline the implementation of update maps both using MapReduce and as an extension on top of Cogset's core, and discuss why the latter approach may be more efficient. Finally, we include a simplified pseudo-code version of the web crawler application that originally motivated our work, to show a more complete example of update map usage.

6.2.1 Update Map Interface

The update map interface attempts to mimic the basic interface of a key/value database, which can be condensed into the following:

`Get(Key) ⇒ Value`	Synchronous look-up
`Set(Key, Value)`	Synchronous overwrite

Keys can be mapped to values through the *Set* operation, and the value associated with a key is retrieved using the *Get* operation. Since the *Set* operation simply overwrites the old value, updates such as incrementing a value by one must be performed by *(1)* reading the old value using the *Get* operation, *(2)* calculating the new value, and *(3)* writing back the new value using the *Set* operation. Not only does this constitute a race condition, it is also inefficient in terms of I/O and latency.

Our update map abstraction addresses this limitation by introducing an *Update* operation that *asynchronously* updates the value associated with a key, possibly as a function of the previous value. Rather than directly specifying a new value, the *Update* operation accepts an *updater* function which is used to determine the new value. The updater function accepts the previous value associated with a key and returns the new value. Note that the updater function can be evaluated lazily, i.e. its evaluation can be deferred until the new value is actually required.

```
function two(key, value):
  return 2

function increment(key, value):
  if value != null:
    return value + 1
  else:
    return 1

function remove(key, value):
  return null

function multiply(key, value, factor):
  return value * factor

Update('abc', two)
Update('abc', increment)
Update('abc', remove)
Update('x', bind(multiply, factor=3))
Update('y', bind(multiply, factor=7))
```

Figure 6.10. Using the *Update* operation to asynchronously update values.

The second part of the update map interface is where the key difference to a traditional key/value database lies; rather than supporting a synchronous *Get* operation, the update map supports a bulk operation to visit all values in the database. This operation is named *Traverse*, alluding to the way it can be implemented using Cogset, and accepts a *visitor* function which is invoked once for each key/value pair in the database. By design, there is no facility to synchronously look up the value associated with a single key; a full traversal of the database is required whenever values are to be read. This allows implementations to collect batches of pending updates and apply them lazily at traversal time. For a given application, the suitability of an update map thus depends on the ratio of updates to traversals. The core interface of update maps is summarized below.

`Update(Key, Updater)`	Asynchronous update
`Traverse(Visitor)`	Visit all values

Figure 6.10 illustrates the use of the *Update* operation with various updater functions, using language-neutral pseudo-code. The first updater function always returns the value 2, unconditionally overwriting any previous value associated with the key. The second function examines the previous value and increments it by 1. If the key had no previously associated value, the

special *null* value is passed to the updater function, which in this case sets the initial value to 1. By a similar convention, the third updater function serves to remove a key from the database by always returning *null*.

The fourth updater function, *multiply*, is more generic and accepts an additional parameter whose value must be bound prior to invoking *Update*. In our example, it is used first to multiply one value by 3 and then another by 7. The value of the additional *factor* argument is specified using a hypothetical *bind* primitive. For functional languages, the natural way to implement parameter binding would be through partial function applications. In our Java implementation of update maps, updater and visitor functions are objects implementing a specific interface. Bound parameters can then be stored as object fields and specified at the time of construction. The same approach would be natural for other object-oriented languages.

```
function removeOdd(key, value):
  if value mod 2 == 1:
    value = null
  return value

function show(key, value):
  print key, value
  return value

Traverse(increment)
Traverse(removeOdd)
Traverse(show)
```

Figure 6.11. Using the *Traverse* operation to visit all key/value pairs.

These examples all ignored the *key* argument, so it might seem superfluous. The main motivation for including it is that it allows updater functions to double as visitor functions, because they have the same type signature. When the *Traverse* operation is used to iterate over all key/value pairs, the visitor function can return a new value for each key that is visited, or remove the key by returning *null*. The same functions can thus be used as updaters to update the values associated with individual keys, or as visitors to update *all* values in the database.

Figure 6.11 provides an example to illustrate the use of the *Traverse* operation. The example first increments all values by 1, then removes all keys with odd-numbered values, and finally displays all remaining keys and values (assuming the existence of a print statement for displaying values).

6.2.2 Implementation of Update Maps

The general idea motivating update maps is that they can be implemented efficiently using a sequential I/O access pattern. In the following, we first describe a generic approach to implementing update maps, regardless of the underlying platform, before going into the specifics of the implementation of update maps on top of MapReduce or using Cogset.

Given its asynchronous nature, the *Update* operation can be implemented by appending the updates to a log, deferring their actual evaluation until it is required by a *Traverse* operation. With appropriate buffering this only incurs an occasional burst of sequential I/O, which gives

the *Update* operation a low amortized cost. The resulting log is a sequence of *update records*, each of which contains an updater function, its bound parameters, the key to which the update applies, and a sequence number. Updates to the same key should be applied in order of increasing sequence numbers. Since it may be practical or even necessary to split the log into several files, it may be convenient to include the sequence numbers explicitly rather than relying on file offsets for ordering of updates.

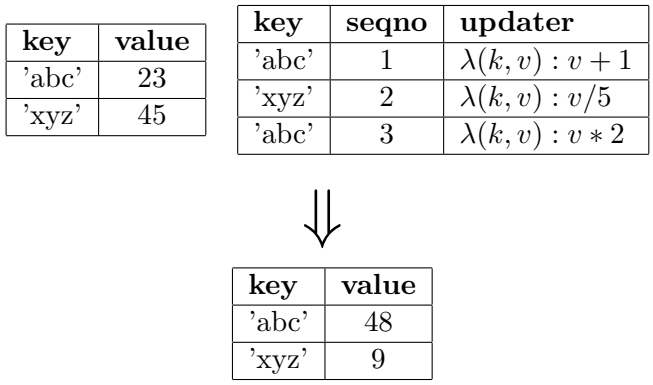


Figure 6.12. Applying a batch of updates, by joining a set of data records with a set of update records, producing an updated set of data records.

In addition to the update log, there must be a set of files containing *data records*, each of which contains a key and its most recently computed value. To implement the *Traverse* operation, all pending updates must first be applied, by joining the update log with the set of data records as illustrated by the example in Figure 6.12. The figure displays updater functions using lambda notation. There are two pending updates for the key "*abc*": Incrementing its value by 1, and multiplying the value by 2, in that order. The "*xyz*" key has one pending update: dividing its value by 5. The set of data records is joined with the set of update records, producing a new set of data records; once this is done, the two input sets can be discarded. The actual invocation of the visitor function is piggybacked on the join algorithm, by invoking the visitor once for each key in the database, *after* any pending updates have been applied, and before writing the new key/value pair to disk. The various ways in which update maps can be implemented differ mainly in how this join is implemented, and in the details of how update and data records are stored.

In our initial work on update maps [65], we developed a stand-alone single-node implementation based on a hash-merge join algorithm. In this implementation, the key space is hashed into a number of buckets, and each bucket contains a separate file for data records, as well as a separate update log. *Update* operations are implemented by hashing the key to be updated, and appending an update record to the corresponding update log. Data record files are maintained in sorted order. To apply the updates in a given bucket, the update log is sorted in-memory and the updates are merged with the data records, reading the old data record file sequentially while writing a new one sequentially. This procedure can be applied separately for each bucket, whenever an update log is about to grow too large to fit in main memory. When executing a *Traverse* operation, each bucket can be processed in turn, so the main memory only needs to accommodate one update log at a time.

Our stand-alone hash-merge implementation proved that a single-node update map could be implemented very efficiently. However, for larger data volumes, a distributed implementation is required. A natural consideration is therefore to implement the update map abstraction on top of MapReduce. Concretely, the update log and the set of data records could be stored in a distributed file system, and joined using a MapReduce computation that implements the *Traverse* operation. Unfortunately, this approach would suffer from poor data locality, and require a full repartitioning and reshuffling of all data for every MapReduce job in order to perform the join. Again, this highlights the shortcomings of MapReduce when applied for relational joins; the joining must be performed in the reduce phase, and the preceding map and data-shuffling phases constitute a very costly way to co-locate the relevant records to be joined.

With Cogset's static routing, the above problem is addressed by ensuring that all records are grouped correctly at the time they are first written to disk. Distributed update maps can therefore be implemented in a straightforward manner using Cogset's core abstractions, with one data set to hold update records and another to hold all data records. Both data sets are hash partitioned using the same keys, and Cogset ensures that corresponding partitions are co-located. The *Update* operation is implemented simply by adding a new update record to its data set, and the *Traverse* operation is implemented using a single traversal. As in the stand-alone hash-merge implementation, the traversal joins the two data sets by processing one partition at a time, so all of the available main memory on a node can be dedicated to holding a single partition of the update log. The obvious difference is that Cogset allows a number of nodes to collaborate in a *distributed* computation, automating the non-functional aspects of partitioning the data across the nodes, and coordinating their activities during a traversal.

In conclusion, both MapReduce and Cogset can be used as an underlying engine for distributed update maps. However, Cogset's static routing may allow for a more efficient implementation, given that update maps rely heavily on relational joins under the hood. The exact performance characteristics of an update map depend on several factors, such as the size and disposition of keys and values, the hardware employed, and not least the access pattern of the application. Intuitively, a complete rebuild of the entire database for every batch of updates may seem prohibitively expensive. However, given the disparity in the effective I/O bandwidth achieved when using sequential I/O as compared to a seek-intensive random access pattern, occasionally regenerating the full data set may well be faster than continuously maintaining index structures to allow in-place modification.

6.2.3 Web Crawler Example

The update map interface, whose simplicity allows for very efficient implementations, might in return seem overly restrictive for real life applications. We therefore present a more complete example in this section: a large scale web crawler—the application that was our original motivation for the update map abstraction. Web crawlers download web pages and parse them to extract hyperlinks to additional pages, which are again downloaded, forming a continuous work cycle.

In many deployments, the web crawler focuses on a narrowed subset of the web, for example a single top-level domain, and thus ignores many URLs. The original web crawler was designed primarily for such use cases, with very flexible policies for scheduling and filtering of URLs to be downloaded. We wanted to redesign it for larger, web-scale crawls, where throughput becomes the main concern.

```
# process downloaded URLs
for url, status in downloads:
  meta = Get(url)
  if status != 200:
    # download error - retry later?
    if meta.retries++ > maxRetries:
      meta.status = gone
    else:
      meta.status = error
  else:
    # download ok - parse links
    meta.timestamp = now()
    meta.status = ok
    for link in parse(meta.content):
      linkMeta = Get(link)
      if linkMeta != null:
        # seen this URL before
        linkMeta.linkCount++
      else:
        # never saw it before
        linkMeta = newMeta(link)
      Set(link, linkMeta)
  Set(url, meta)

# schedule more URLs for download
for url, meta in database:
  if meta.status = new:
    if meta.linkCount > 3:
      schedule(url)
  else if meta.status in (ok, error):
    if meta.timestamp < yesterday():
      schedule(url)
```

Figure 6.13. The main loop of a web crawler using a key/value database.

The main bottleneck in the original web crawler turned out to be the key/value database used to store various meta-data for each URL. Statistics such as the last download time, the number of failed download attempts, and the number of in-links encountered were recorded for each unique URL, which entailed enough random I/O accesses to severely limit the overall throughput.

Figure 6.13 gives a somewhat simplified view of the main loop of the original crawler: a separate download engine manages a large set of concurrent downloads, and the main loop iterates over the downloaded URLs and their HTTP status codes. Failed downloads are retried for a number of times before the URLs are flagged as permanently missing. For successful downloads, a time stamp is recorded and the downloaded content is parsed for additional hyperlinks. For each link, the in-link count of the target URL is incremented, or a new meta-data entry is created if it is the first time the URL is encountered. Finally, the database is traversed in order to schedule additional URLs for download. In this example, new URLs are scheduled for

download if they have been encountered in at least 3 links, while previously downloaded URLs are rescheduled for a new download after 24 hours.

Once the set of encountered URLs grows sufficiently large, the frequent meta-data look-ups of the crawler's main loop become a performance problem. The key to optimizing the loop is realizing that the URL meta-data is only ever updated as a function of its previous value. Synchronous look-ups are not required; for example, there is no need to synchronously read the previous in-link count of a URL in order to increment it. However, we are hampered by the interface of the key/value database, which forces us to do synchronous look-ups in order to modify the meta-data. Figure 6.14 shows how we overcame this in the new version of our crawler, by rewriting the inner loop to use an update map with three different updater functions and one visitor function. The update map interface is thus sufficiently flexible for this application; updates can be processed asynchronously and new batches of URLs for the download engine can be generated by an occasional traversal.

6.3 Summary

In this chapter, we discussed Cogset in a new perspective, focusing on its generality and extensibility, rather than its specific application as a MapReduce engine. We presented two new abstractions—Oivos and update maps—to substantiate our claim that Cogset's core engine can be employed as a foundation for multiple higher-level abstractions.

Oivos addresses the complications of structuring large computations as collections of related MapReduce programs, by introducing a higher-level declarative language in which multiple related data sets can be manipulated. New data sets can be derived from existing data sets through the application of a number of operators, whose behavior is customized through user-defined functions. Oivos automatically composes user-defined functions where possible, and compiles a set of traversals to execute in order to materialize the desired output data sets. Externally, Oivos resembles other workflow composition languages developed for Hadoop and Dryad, and demonstrates that equivalent abstractions can be built using Cogset's core engine.

With update maps, we set out to explore the feasibility of restructuring real-life applications to rely exclusively on sequential I/O. Applications that rely on synchronous look-ups in a key/value database abstraction would seemingly be hard pressed to make do without random I/O accesses. However, many synchronous look-ups can be rewritten as asynchronous updates, paving the way for batch processing using the update map abstraction.

We described how to implement update maps in a stand-alone manner on a single node, and how to implement distributed update maps on top of MapReduce and using Cogset. Given that the core operation in an update map is essentially a relational join, Cogset's static routing should be highly beneficial for this application. In this instance, Cogset's core engine may therefore be better suited than a traditional MapReduce engine as a foundation for higher-level abstractions.

```
function urlError(url, meta):
  if meta.retries++ > maxRetries:
    meta.status = gone
  else:
    meta.status = error
  return meta

function urlOK(url, meta):
  meta.timestamp = now()
  meta.status = ok
  for link in parse(meta.content):
    Update(link, addLink)
  return meta

function addLink(url, meta):
  if meta != null:
    meta.linkCount++
  else:
    meta = newMeta(url)
  return meta

function scheduleURL(url, meta):
  if meta.status = new:
    if meta.linkCount > 3:
      schedule(url)
  else if meta.status in (ok, error):
    if meta.timestamp < yesterday():
      schedule(url)
  return meta

# process downloaded URLs
for url, status in downloads:
  if status != 200:
    Update(url, urlError)
  else:
    Update(url, urlOK)

# schedule more URLs for download
Traverse(scheduleURL)
```

Figure 6.14. The main loop of a web crawler using an update map.

Chapter 7

Concluding Remarks

We conclude this dissertation by summarizing and re-stating our results, comparing them to our main thesis. Based on our results, we draw three main conclusions. Finally, we highlight certain open questions and unresolved issues, and outline how these could be investigated and addressed in future work.

7.1 Results

This dissertation explores alternative designs for distributed MapReduce engines, focusing in particular on the algorithm for *data shuffling*, where the output from the map phase is repartitioned and redistributed in preparation for the reduce phase.

Traditional MapReduce engines, following the original design of Google's engine, employ *dynamic routing* for data shuffling, an algorithm where intermediate data is stored temporarily on the nodes executing map tasks and subsequently fetched on demand by the nodes executing reduce tasks. The mechanisms for load balancing and fault tolerance are closely intertwined with this algorithm, and dynamic routing is thus a defining characteristic of traditional MapReduce engines.

An alternative approach is to use *static routing*, in which a predetermined configuration dictates where to process and store each data partition. While this general approach is known from parallel databases, the feasibility and ramifications of employing static routing as a core principle in a MapReduce engine are previously unexplored. As stated initially in Section 1.4, we conjecture that a MapReduce engine based on static routing *is* feasible, and could result in improved performance. Specifically, the thesis of this dissertation is that:

It is possible to build a high-performance MapReduce engine based on static routing.

To evaluate our thesis we designed and implemented Cogset, which deviates considerably from the traditional design of a MapReduce engine. The use of static routing for data shuffling and data distribution was adopted as a central design choice that was allowed to shape the remainder of Cogset's design. As such, Cogset is a system *based on static routing*. The question of whether or not our thesis holds can then be broken down into two parts:

1. Does Cogset qualify as a fully functional MapReduce engine?
2. Does Cogset qualify as a high-performance MapReduce engine?

To answer the first question, we first note that Cogset features a MapReduce API compatible with Hadoop, and is capable of executing existing Hadoop applications in a semantically equivalent way. As such, Cogset fulfills the *functional* requirements of a MapReduce engine. In Section 1.4, we also listed three common *non-functional* requirements for a MapReduce engine: reliable storage, fault tolerance, and load balancing.

Cogset employs *replication* to meet the requirement for reliable storage. All data sets are partitioned, and each partition is replicated on multiple nodes in order to protect against data loss. The replication of data set partitions also plays a crucial role in the algorithms for fault tolerance and load balancing. As described in Chapter 3, Cogset provides a core processing abstraction called a *traversal* which allows a user-defined function to process all partitions of one or more data sets in a fault-tolerant and load-balanced way. Even though partitions are statically assigned to certain nodes, the traversal algorithm can balance the load imposed on individual nodes by carefully selecting which of a partition's replicas to process. Slow nodes can then be off-loaded by other nodes that are hosting replicas of the same partitions. Similarly, replication ensures that no Cogset node is a single point of failure, and a traversal will continue to make progress even if one node fails, by processing its partitions elsewhere.

To answer the second question, of whether or not Cogset qualifies as a high-performance MapReduce engine, we refer to the results of our experimental evaluation in Chapter 5. Using the established MR/DB benchmark, we compared Cogset's performance to Hadoop and observed that Cogset performed markedly better for the majority of the benchmark tasks. In some instances, the performance gap was so large that we were compelled to investigate further, and discovered two concrete implementation weaknesses in Hadoop.

In particular, our experiments show that Cogset performs much better than Hadoop when sequentially scanning through a data set. Cogset's architecture allows such scans to be performed by processes reading directly from the local disk. With Hadoop's architecture, all reads from the distributed file system go through a network connection, imposing additional overhead and a more scattered I/O access pattern. Moreover, Hadoop's internal program structure leads to single-threaded CPU bottlenecks, reducing performance on modern multi-core processors. By adopting elements of Cogset's multi-threaded program structure in a custom plug-in for Hadoop, we were able to close some of the performance gap between Cogset and Hadoop.

Additionally, we observed a weakness in Hadoop's task scheduling algorithm that left nodes idle, leading to reduced throughput and increased completion times for individual jobs. Patching this weakness also improved Hadoop's performance. In contrast, Cogset has a fully distributed scheduling mechanism where nodes only coordinate with their immediate neighbors (as defined in Section 3.4), allowing them to make prompt scheduling decisions without introducing a potential central bottleneck.

To us, these unanticipated findings on Hadoop reinforce the importance of experimenting with alternative designs, challenging common assumptions. By building an untraditional MapReduce engine, based on static routing, we also discovered ways to improve the performance of engines following the traditional design.

As demonstrated in Chapter 6, Cogset also serves as a viable platform for higher-level abstractions, which can be built directly on top of its core engine, bypassing the MapReduce interface. Cogset traversals are a flexible building block that encapsulate the non-functional concerns associated with data distribution, fault tolerance, and load balancing, while imposing a minimum of semantical restrictions on the user-defined code. For certain applications, such as our *update map* abstraction, the ability to freely manipulate multiple data sets with

custom traversals allows a more efficient algorithm than an implementation restricted by the MapReduce programming model.

7.2 Conclusions

In summary, based on the work presented in this dissertation, we draw the following three main conclusions:

1. Cogset qualifies as a high-performance MapReduce engine, and thus confirms our main thesis that a high-performance MapReduce engine can be based on static routing.

2. Aspects of Cogset's implementation can also be retrofitted into a traditional MapReduce engine, to realize significant performance gains.

3. Cogset can also be used as a flexible core engine for alternative, higher-level programming abstractions.

7.3 Future Work

There are many interesting lines of inquiry to pursue in the future, to follow up on the work presented here or investigate related approaches. In the following, we list some of the most interesting areas for future work.

Further Benchmarking Although Cogset was subjected to a range of workloads with the MR/DB benchmark, it would be interesting to perform additional benchmarking, and to directly compare Cogset to other systems than Hadoop. One possible weakness of the MR/DB benchmark is its general focus on read-intensive workloads, for which Hadoop appears to be particularly inefficient. More extensive benchmarking with larger scale deployments and with write-intensive workloads would be useful to establish a more complete understanding of the potential trade-offs involved in designing a MapReduce engine.

Further experiments might also allow us to pin-point more exactly the reasons why Cogset performs better than Hadoop, and to what extent similar performance improvements can be realized for Hadoop by refactoring its implementation. It would be interesting (though potentially difficult) to separate the performance characteristics resulting from *architectural* differences from those resulting from aspects of implementation.

Fault Tolerance and Load Balancing A more theoretical analysis and formalization of the traversal algorithm employed by Cogset would be interesting, and an experimental evaluation focusing on the non-functional aspects of traversals is in order. Even though data skew may occur, the MR/DB benchmark that we adopted for this work does not explicitly evaluate load balancing as a separate aspect. Similarly, the benchmark only evaluates performance in failure-free scenarios. There are varying degrees of fault tolerance and ability to balance load, so further experiments along this axis would be very interesting. Our implementation of Cogset also requires stopping the system in order to add nodes or reintegrate failed nodes. Devising algorithms to perform these and other reconfigurations as on-line operations could pose an interesting challenge.

Multicast Communication As noted in Section 4.11, Cogset might benefit from employing a multicast protocol for network communication. This is due to the communication pattern where identical replicas of a page are distributed to multiple nodes. A related question is whether or not additional performance improvements can be achieved by sacrificing the property of ordered message delivery and changing the semantics such that the records of a data set partition have no well-defined ordering. In the future, Cogset might be an interesting test bed for experimental communication protocols.

Higher-level Abstractions When building abstractions on top of Cogset, we believe there is still great potential for innovation. In the future, we expect to both refine our existing higher-level abstractions and introduce new ones. This may also drive the development of new core features to include in Cogset, as new requirements are exposed. A broader range of higher-level abstractions would strengthen our confidence that Cogset can function as a generic and efficient core engine for a wide variety of data processing applications.

Appendix A

Publications

This dissertation is based on work presented in the following publications:

Paper I

> Steffen Viken Valvåg and Dag Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *Proceedings of the 2008 Tenth IEEE International Conference on High Performance Computing and Communications (HPCC 2008)*, pages 113–122. IEEE Computer Society, September 2008.

In this paper, we point out the inconvenience and potential inefficiency of structuring computations as collections of related MapReduce programs. To facilitate the expression of more complex computations, we introduce Oivos; a high-level declarative programming model for manipulation of data sets. We describe the semantics of this programming model, where a number of operators allow new data sets to be derived from existing data sets, and materialized on demand through lazy evaluation. As with MapReduce, user-defined functions play a central role, and the Oivos operators are parametrized by user-defined functions to customize their behavior.

We show how Oivos programs may specify computations that span a heterogeneous collection of interdependent data sets, and present an underlying run-time for Oivos which includes a block-based distributed file system and an engine for executing arbitrary sets of tasks connected through data dependencies into an acyclic precedence graph. We explain how Oivos computations are compiled into precedence graphs by partitioning data sets and arranging for independent processing of all partitions, in parallel. By adjusting the granularity of the partitioning, the degree of parallelism can be increased automatically, without altering the high-level user code. Compared to MapReduce, Oivos thus offers the same ability to execute distributed computations based on user-defined functions without concern for the practical details of data distribution, synchronization, scheduling, and fault tolerance, but has the added benefit that such computations can involve multiple related data sets.

In our experimental evaluation, we show how an Oivos program can outperform an equivalent sequence of MapReduce passes, by avoiding superfluous repartitioning steps and performing less I/O.

Paper II

Steffen Viken Valvåg and Dag Johansen. Update Maps — A New Abstraction for High-Throughput Batch Processing. In *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage (NAS 2009)*, pages 431–438. IEEE Computer Society, July 2009.

In this paper, we discuss the trade-offs between latency and throughput that arise when designing the storage system for an application, depending on the application's access pattern and the underlying on-disk data structures. Many applications are designed to use a synchronous storage interface, such as that provided by a key/value database, where data is modeled as key/value pairs and the value associated with a given key can be retrieved or modified synchronously, via index structures such as B-trees or hash tables. Such databases invariably have a random I/O access pattern, which is inefficient on traditional storage media.

To maximize throughput, an alternative is to employ batch processing, accumulating pending updates and applying them periodically, in batches. Batch processing may enable a more efficient sequential I/O access pattern and thereby trade latency for improved throughput. One way to implement batch processing is using the MapReduce programming model, which is particularly attractive for distributed applications due to its masking of non-functional concerns. With MapReduce, data sets are modeled similarly as in a key/value database, as a set of key/value pairs. However, MapReduce computations process data sets in bulk, by applying a user-defined function to *all* key/value pairs. Refactoring an application from using a key/value database to using a batch-oriented approach such as MapReduce can be difficult.

Such transitions are facilitated by our *update map* abstraction, which aims to combine the convenience of a key/value database with the performance of a batch-oriented approach. The update map interface resembles that of an ordinary key/value database, but its implementation can rely on batch processing and sequential I/O, for improved throughput. To demonstrate how this abstraction can be useful for real-life applications, we show how an example web crawler application based on a key/value database can be refactored to use update maps for storage. Based on this example, we conclude that update maps are particularly attractive when applications originally designed to use key/value databases must evolve to meet new requirements with regards to load or scale.

We evaluate three different implementations of update maps and study their performance trade-offs. One implementation employs the Hadoop MapReduce engine and executes periodic MapReduce jobs to apply batches of updates. Another implementation, which proved more efficient, operates similarly but employs a hand-crafted hash-merge join algorithm to apply updates, and ensures better data locality by partitioning both the set of key/value pairs and the set of pending updates in a consistent manner. Depending on the application's access pattern, this batch-oriented implementation can significantly outperform our third implementation, which is based on an underlying key/value database.

Paper III

> Steffen Viken Valvåg and Dag Johansen. Cogset: A Unified Engine for Reliable Storage and Parallel Processing. In *Proceedings of the 2009 Sixth IFIP International Conference on Network and Parallel Computing (NPC 2009)*, pages 174–181. IEEE Computer Society, October 2009.

In this paper, we introduce Cogset as an alternative to traditional MapReduce engines, presenting its overall architecture and main abstractions. We also note our vision to let Cogset serve as the foundation for a stack of higher-level abstractions. As our main motivation, we observe that the traditional algorithm for data shuffling may cause poor data locality. We therefore adopt static routing as a central design choice, and create a tighter coupling between the mechanisms for reliable storage and parallel processing.

We outline Cogset's traversal algorithm, describing how it tolerates failures and enables load balancing within the constraints posed by static routing. We also describe the mechanisms for distributing records, and the commit protocol used to ensure consistency. We then present Cogset's core programming interface and four benchmark applications using Cogset, drawn from the web indexing domain. In our experimental evaluation, we compare the performance of these applications to a set of equivalent Hadoop applications, with very favorable results.

Paper IV

> Steffen Viken Valvåg, Dag Johansen, and Åge Kvalnes. Cogset vs. Hadoop: Measurements and Analysis. In *Proceedings of the 2010 Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, pages 768–775. IEEE Computer Society, December 2010.
>
> An extended version of this paper has been submitted for review to the journal *Concurrency and Computation: Practice and Experience.*

In this paper, we describe Cogset's support for MapReduce computations through a Hadoop compatibility layer implemented on top of the core Cogset interfaces. We then leverage this compatibility to directly compare the performance of Cogset and Hadoop, using the MR/DB benchmark developed by Pavlo et al. We examine the various data sets and tasks comprising the MR/DB benchmark, describe the systems that have previously been evaluated using the same benchmark, and detail our experimental setup. Our results show that Cogset generally outperforms Hadoop by a significant margin.

To discover why, we analyze Hadoop's internals in more detail and uncover issues with Hadoop's implementation that adversely affect its benchmark performance. Adopting principles from Cogset's implementation, we modify Hadoop's task scheduling algorithm and develop a plug-in for multi-threaded record parsing, mapping, and combining. These optimizations of Hadoop are successful in closing some of the performance gap.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. EECS-2009-28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.

[2] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, Elsevier, 1998.

[3] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 335–350, USENIX Association, 2006.

[4] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pp. 363–375, ACM, 2010.

[5] 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*, vol. 26, no. 2, pp. 4:1–4:26, ACM, 2008.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th symposium on Operating Systems Design and Implementation*, OSDI '04, pp. 137–150, USENIX Association, 2004.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43, ACM, 2003.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pp. 135–146, ACM, 2010.

[9] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakomar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," in *Proceedings of the 36th International Conference on Very Large Data Bases*, vol. 3 of *VLDB '10*, pp. 330–339, VLDB Endowment, 2010.

[10] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, IOS Press, 2005.

[11] "List of Companies Using Hadoop." Online at http://wiki.apache.org/hadoop/PoweredBy, archived at http://www.webcitation.org/61N1GRihM, September 2011.

[12] "Amazon Elastic MapReduce." Online at http://aws.amazon.com/elasticmapreduce, archived at http://www.webcitation.org/61N2V3KxX, September 2011.

[13] D. J. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, ACM, 1992.

[14] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 26, no. 1, pp. 64–69, ACM, 1983.

[15] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, ACM, 2010.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pp. 1099–1110, ACM, 2008.

[17] M. Stonebraker, "The case for shared nothing," *Database Engineering*, vol. 9, no. 1, pp. 4–9, IEEE Computer Society, 1986.

[18] F. B. Schneider, "Byzantine generals in action: implementing fail-stop processors," *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 145–154, 1984.

[19] D. Swade, *The Difference Engine*. Penguin, 2002.

[20] P. J. Denning, "Is computer science science?," *Communications of the ACM*, vol. 48, no. 4, pp. 27–31, ACM, 2005.

[21] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, ACM, 1989.

[22] E. F. Codd, "Multiprogram scheduling: parts 3 and 4. scheduling algorithm and external constraints," *Communations of the ACM*, vol. 3, no. 7, pp. 413–418, ACM, 1960.

[23] J. L. McKenney, "Simultaneous processing of jobs on an electronic computer," *Management Science*, vol. 8, no. 3, pp. 344–354, INFORMS, 1962.

[24] E. S. Schwartz, "An automatic sequencing procedure with application to parallel programming," *Journal of the ACM*, vol. 8, no. 4, pp. 513–537, ACM, 1961.

[25] J. P. Anderson, "Program structures for parallel processing," *Communications of the ACM*, vol. 8, no. 12, pp. 786–788, ACM, 1965.

[26] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commununications of the ACM*, vol. 9, no. 3, pp. 143–155, ACM, 1966.

[27] A. Opler, "Procedure-oriented language statements to facilitate parallel processing," *Communications of the ACM*, vol. 8, no. 5, pp. 306–307, ACM, 1965.

[28] W. R. Sutherland, *The on-line graphical specification of computer procedures*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Tecnology, 1966.

[29] D. A. Adams, "A model for parallel computations," in *Parallel Processor Systems, Technologies, and Applications*, pp. 311–333, Spartan, 1970.

[30] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376, Springer, 1974.

[31] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, IEEE Computer Society, 1980.

[32] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pp. 99–108, ACM, 1988.

[33] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "Gamma - a high performance dataflow database machine," in *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pp. 228–237, Morgan Kaufmann Publishers Inc., 1986.

[34] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pp. 102–111, ACM, 1990.

[35] R. H. Arpaci-Dusseau, "Run-time adaptation in river," *ACM Transactions on Computer Systems*, vol. 21, no. 1, pp. 36–86, ACM, 2003.

[36] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick, "Cluster I/O with River: Making the fast case common," in *Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems*, IOPADS '99, pp. 10–22, ACM, 1999.

[37] 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 SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pp. 59–72, ACM, 2007.

[38] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *Proceedings of the 19th annual ACM symposium on Principles of distributed computing*, PODC '00, pp. 219–227, ACM, 2000.

[39] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pp. 1029–1040, ACM, 2007.

[40] "About Cascading." Online at http://www.cascading.org/about.html, archived at http://backupurl.com/qhsevi, September 2011.

[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, VLDB Endowment, 2009.

[42] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pp. 1–14, USENIX Association, 2008.

[43] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, VLDB Endowment, 2008.

[44] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pp. 223–236, ACM, 2010.

[45] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears, "Boom: Data-centric programming in the datacenter," Tech. Rep. UCB/EECS-2009-98, EECS Department, UC Berkeley, 2009.

[46] Y. Gu and R. L. Grossman, "Sector and sphere: the design and implementation of a high-performance data cloud," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1897, pp. 2429–2445, The Royal Society, 2009.

[47] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA-13, pp. 13–24, IEEE Computer Society, 2007.

[48] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pp. 198–207, IEEE Computer Society, 2009.

[49] W. Fang, B. He, Q. Luo, and N. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 608–620, IEEE Computer Society, 2011.

[50] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 260–269, ACM, 2008.

[51] Z. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, and D. Johansen, "The Nornir run-time system for parallel programs using Kahn process networks," in *Proceedings of the 2009 6th IFIP International Conference on Network and Parallel Computing*, NPC '09, pp. 1–8, IEEE Computer Society, 2009.

[52] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, pp. 471–475, North Holland, 1974.

[53] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 810–818, ACM, 2010.

[54] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, VLDB Endowment, 2009.

[55] C. Yang, C. Yen, C. Tan, and S. Madden, "Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database," in *Proceedings of the 26th International Conference on Data Engineering*, ICDE '10, pp. 657–668, IEEE Computer Society, 2010.

[56] "About PostgreSQL." Online at http://www.postgresql.org/about, archived at http://backupurl.com/bp1614, September 2011.

[57] J. W. Stamos and D. K. Gifford, "Remote evaluation," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 537–564, ACM, 1990.

[58] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of the 6th International Conference on Data Engineering*, ICDE '90, pp. 456–465, IEEE Computer Society, 1990.

[59] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 159–174, ACM, 2007.

[60] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pp. 165–178, ACM, 2009.

[61] "Vertica Analytics Platform." Online at http://www.vertica.com/the-analytics-platform, archived at http://backupurl.com/sturh2, September 2011.

[62] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," in *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pp. 553–564, VLDB Endowment, 2005.

[63] "Oracle Berkeley DB." Online at http://www.oracle.com/technetwork/database/berkeleydb, archived at http://www.webcitation.org/61N9DWbtp, September 2011.

[64] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220, ACM, 2007.

[65] S. V. Valvåg and D. Johansen, "Update maps - a new abstraction for high-throughput batch processing," in *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, NAS '09, pp. 431–438, IEEE Computer Society, 2009.

[66] S. V. Valvåg and D. Johansen, "Oivos: Simple and efficient distributed data processing," in *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pp. 113–122, IEEE Computer Society, 2008.

[67] S. V. Valvåg and D. Johansen, "Cogset: A unified engine for reliable storage and parallel processing," in *Proceedings of the 2009 6th IFIP International Conference on Network and Parallel Computing*, NPC '09, pp. 174–181, IEEE Computer Society, 2009.

[68] S. V. Valvåg, D. Johansen, and Å. Kvalnes, "Cogset vs. Hadoop: Measurements and analysis," in *Proceedings of the 2010 2nd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '10, pp. 768–775, IEEE Computer Society, 2010.

ISBN xxx-xx-xxxx-xxx-x