BIG DATA PROCESSING WITH PEER-TO-PEER ARCHITECTURES

GOH WEI XIANG

(*B. Comp. (Hons), NUS; Dipl.-Ing., Télécom SudParis*)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

"Tell me, Sir Samuel, do you know the phrase 'Quis custodiet ipsos custodes?'?"

It was an expression Carrot has occasionally used, but Vimes was not in the mood to admit anything, "Can't say that I do, sir", he said. "Something about trifle, is it?"

"It means 'Who guards the guards themselves?' Sir Samuel."

"Ah."

"Well?"

"Sir?"

"Who watches the Watch? I wonder?"

"Oh, that's easy, sir. We watch one another."

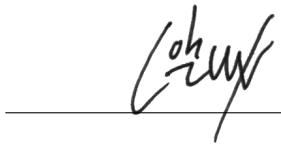
"Really? An intriguing point..."

– Terry Pratchett, Feet of Clay

Declaration

I hereby declare that this thesis proposal is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis proposal.

This thesis proposal has not been submitted for any degree in any university previously.



Goh Wei Xiang

18 June 2014

Acknowledgements

Nanos gigantium humeris insidentes.

I stand on the shoulders of giants in hope that one day, I too may provide the leg-up for those who come after. To the titans before me, I can only offer, for now, my words of gratitude:

I would like to thank Ms. Toh Mui Kiat, Ms. Loo Line Fong, Ms. Agnes Ang Hwee Ying, Mr. Bartholomeusz Mark Christopher, Ms. Irene Ong Hwei Nee and all the other management staffs for the administrative support; the endless correspondence of emails makes the world go round.

I would like to thank the entire Technical Services team for clearing up the mess when I screwed up the various systems one way or another; allow me to salute the unsung heroes of technical support.

I would like to thank Prof. Khoo Siau Cheng for helping me when I was in France and again when I came back; *je vous remercie infiniment.*

I would like to thank Prof. Chan Chee Yong and Prof. Stéphane Bressan for all the critical comments; the hottest fire makes the strongest steel.

I would like to thank Prof. Chin Wei Ngan for introducing me to functional programming languages; this has led me to delve into the *abstract nonsense* called *Category Theory*.

I would like to thank Prof. Ooi Beng Chin for introducing me to the works of structured peer-to-peer overlays; your lectures on *Advanced Topics in Databases (CS6203)* are the beginning of this work.

Most importantly, I would like to sincerely thank Prof. Tan Kian-Lee for ... *everything*. Thank you, sir.

Lastly, on a personal side, I would like to thank, as well as apologize to, my family — my father, mother and brother — for their continual support in all aspects of my life so that I can selfishly satisfy my personal indulgence in research work; some words are easier written than said: thank you, and sorry.

Contents

Summary

Recent developments in the realm of computer science have brought about the introduction of, what some may classify as, *disruptive technologies* into the peripheral of both researchers and developers alike. In present-day academic and industrial parlance, we frequently hear the mention of the adoption of the *Big Data* paradigm, or the deployment with *cloud computing*, or the *NoSQL* movement, or the use of the *MapReduce* framework. While some may have their reservations on the novelty or the longevity of these newly introduced concepts, their continual widespread adoption in the industry undoubtedly indicates previously unsatisfied needs for certain systemic providence from the software solutions of yesteryear. Three such desirable qualities of a system architecture can be identified: massive horizontal scalability, robust distributed processing, and elastic resource consumption.

Currently, the predominant architecture adopted for modern data processing system is that of the master/workers architecture; the main rationale for this adoption is said to be for the simplicity of the system design. However, it is perhaps profitable to investigate more elaborated alternatives, especially if systemic qualities may be enhanced as a result. Extrapolating from the desirables, it appears that *structured peer-to-peer*

(P2P) overlays present as a good match to the conditions established by the industry. This thesis sets out to demonstrate the feasibility of adopting a structured P2P overlay in the design of modern data processing system such that some of the identified systemic qualities may be magnified.

On horizontal scalability, work has been done to develop a generalized data processing framework, much like the MapReduce framework except that the programming model and the system architecture are completely decentralized. The *Katana* framework builds on the algebraic structure exhibit by many structured P2P overlays to materialize its programming model, which encompasses the expressiveness of the MapReduce programming model. Experimental results indicate that the augmented expressiveness, coupled with the decentralization of control, provides performance improvement in execution over widely scaled clusters.

In terms of robust processing, research has been conducted to investigate the incorporation of the decentralized fault-tolerance of structured P2P overlays into modern data processing system. In particular, the robust processing of the MapReduce framework can be generalized into an abstract model of fault-tolerant processing called the *COVER-CHARGE protocol* (CCP). The Katana framework is extended to incorporate the CCP so as to render its operations fault-tolerant. Experimental studies indicate that the overhead incurred by the CCP for the operations in the extended Katana framework, called *hardened* Katana framework, is comparable to, if not lesser than, that of the MapReduce

framework. Moreover, the robustness induced within hardened Katana is derived directly from its decentralized architecture, and not some external mechanism.

For the notion of elasticity, the feasibility of enhancing the elasticity of the MapReduce execution by embedding a structured P2P overlay into its execution architecture has been explored. By deploying the elastic overlay over the worker sites, the processing element of this new execution architecture, called *Elastic MapReduce Execution* (EMRE), is able to stretch or shrink in response to resource allocation, thus allowing elastic processing without any changes to the exposed interface. Furthermore, since the overlay also presents as a distributed index, the infamous *shuffle* phase of MapReduce can be pipelined, resulting to overall improvement in running times. In addition, simulated progressive availability of resources in experiments shows that EMRE has superior capability to handle such a situation as compared to unmodified MapReduce.

List of Figures

List of Symbols

Mathematical Symbols

$\mathbb{N}$	Natural number set, $\mathbb{N} \triangleq \{i \mid i \in \mathbb{Z}, i \geq 0\}$
$\mathbb{R}$	Real number set
$\mathbb{Z}$	Integer set, $\mathbb{Z} \triangleq \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Z}^+$	Positive integer set, $\mathbb{Z}^+ \triangleq \{i \mid i \in \mathbb{Z}, i > 0\}$

Generic Functions

$\lfloor \cdot \rfloor$	Floor function, $\forall x \in \mathbb{R}$, $\lfloor x \rfloor = \max(\{i \in \mathbb{Z} \mid i \leq x\})$
$\mathcal{P}(S)$	Power set of S, $\mathcal{P}(S) \triangleq \{S' \mid \forall S' \subseteq S\}$
$\max(x_1, x_2, \ldots, x_n)$	Multi-variable maximum function
$\min(x_1, x_2, \ldots, x_n)$	Multi-variable minimum function
$\arg\min_x f(x)$	Argument of the minimum of $f(x)$

Probability Notions

$\exp(\lambda)$	Exponential distribution with λ as the rate parameter

$\Pr(X)$	Probability that event X occurs
CDF	Cumulative distribution function
$E(X)$	Expected value of the random variable X

Other Mathemtical Notations

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph $\mathcal{G}$ is an ordered pair of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$

Type Notations

$v :: T$	Variable v is of type T
$[T]$	A list/array of type T
(T_1, T_2)	An ordered pair of type T_1 and T_2
$T_1 \to T_2$	A function mapping type T_1 to type T_2

Chapter 1

Introduction

1.1 Recent Developments

The perpetual acceleration in the growth of digital data handled has now been, more or less, taken as an irrefutable fact in all academic and industrial discussions in the database community; and it is rightfully so. Gantz and Reinsel (2012) estimated that the size of all digital data created and consumed in 2012 was about 2,837 exabytes and this number will double[1] approximately every two years from 2012 to 2020. It is believed that in 2012, 23% of the digital data created would be useful for analytics but only 3% was captured and curated (Gantz and Reinsel, 2012); even so, 11% of surveyed data managers already reported to have petabyte-scale data stores (McKendrick, 2012) indicating that we have not yet experienced the full potential of the continual digitalization of the world. Devlin (2011) projected that the compound annual growth rate (CAGR) of unstructured business data

[1]It will not be surprising if the actual size exceeds this estimate; previously, Gantz et al. (2007) estimated that the size of the digital data created and consumed in 2010 should be 988 exabytes when it was actually about 1,227 exabytes based on actual findings (Gantz and Reinsel, 2012).

is about 60% while the CAGR of structured business data is projected to be about one-third of that; therefore the below-par data acquisition also indicates that data sources will become increasingly varied. Boosted by such radical underlying change, there has been an unprecedented furor of activities in the database community:

Paradigms challenged. Increasingly, we have witnessed the database community accepting revisions to well-established ideologies. For example, the Atomicity-Consistency-Isolation-Durability (ACID) quadruplets have long been the fundamentals in database management for assuring reliable data processing. In seeking to cope with wider service demands, Fox et al. (1997) were the first[2] to propose using soft state and eventual consistency to augment availability but the idea was not immediately well-received partly because it was deemed as an antithesis to that of the ACID properties (Brewer, 2012). It was until Brewer (2000) explored this idea further with what is now known as the *Brewer's Theorem* (Gilbert and Lynch, 2002) that the community began to look into the consistency-versus-availability argument, thus promoting the movement that advocates the relaxation of the ACID properties at some levels in a system (Cattell, 2011). Currently, such a school of thought has become an legitimate consideration in mainstream system designs (Brewer, 2012).

Limits breached. The resources invested in handling data seem to mirror its exponential growth such that yesterday's limit becomes today's baseline. In May 2010, Facebook broke new ground by announcing that it had deployed the then-largest single Hadoop cluster consisting

[2]Though the idea of eventual consistency has always been a design consideration (Saito and Shapiro, 2005) and was conceptualized as early as 1975 (Johnson and Thomas, 1975).

of 2,000 nodes and 21 petabytes of storage (Borthakur, 2010). Just a year later, there were at least 22 reported petabyte-scale clusters, of which Yahoo! possessed the largest one, which consisted of a total of 42,000 nodes with about 200 petabytes of data (Wong, 2013); Monash (2011) estimated Yahoo!'s biggest single Hadoop cluster to be a little over 4,000 nodes. In fact, across the board from 2010 to 2011, the average Hadoop cluster size rose from 60 nodes to 200 nodes (Monash, 2011); adoption rate of Hadoop is also expected to double in the coming years (McKendrick, 2012).

Contexts evolved. As the world gets progressively digitalized, new environmental contexts are injected into the mix of database researches. Today, we talk about the concept of *Internet of Things* whereby every physical object may have a virtual representation on the Internet (Atzori et al., 2010). We experience an avalanche of social networking services (e.g., Facebook, Twitter and Google+) where even non-physical objects (e.g., personal relationships, human conditions and social community) may have virtual representations on the Internet. Furthermore, mobile computing have progressed to the point that, virtual presences on the Internet never cease and may be perpetually on-the-move. Uncovering these uncharted lands have brought about new foci of research in the database community (e.g., Aggarwal et al., 2013; Fernando et al., 2013; King et al., 2009).

While the sheer size of digital data has a direct impact on database developments, the latter also positively affects the former in return, creating the virtuous (perhaps vicious[3]) cycle of digitalization. Equipped with better data engineering and more sophisticated processing tools, not only the limit

[3]Just kidding.

on the size of managed data is lifted, the utility of data as deemed by the industry is also expanded, thus promoting the interest in further digitalizing information of all types. This is evident in that 19% of surveyed data managers indicated that already 25% or more of their data is unstructured (i.e., not trivially relational) and 65% of the respondents further confirm that the amount of unstructured data is expected to increase (McKendrick, 2012). Such is the perpetual dynamics on this commodity that we call "data".

Set in such a volatile backdrop, new ideas are continually being introduced into the landscape; there are some concepts, or buzzwords as some may prefer, that consistently come to attention. In the parlance of database, we frequently hear about the mention of the adoption of the *Big Data* paradigm, or the deployment with *cloud computing*, or the *NoSQL* movement, or the use of the *MapReduce* framework. Being rather novel, these concepts actually do not yet have globally-accepted definitions. As such, these concepts tend to have overlapping jurisdiction whenever they are brought up. To make matters worse, many refer to some of them as synonymous while others may deem a couple of them to be encompassing the others. While it may be pointless, and certainly futile, at this point, to try to give these concepts exact formal definitions, it is worthwhile to investigate the *raison d'être* of their frequent co-occurrences in the discussion of database as a prelude to the presentation of some desirable qualities of the architecture of a modern data processing system[4].

1.1.1 Big Data

Dealing with limit-breaking volume of data is not a novel theme; ever since the invention of direct-access storage in the 1960s, computer scientists have

[4]The term *data processing system* is used to refer collectively to any system that is devised to perform some form of data processing.

been pre-occupied with the management of ever-increasing data size. Then, Codd (1983) introduced in his seminar paper the groundbreaking concept of the *relational data model*, which basically requires that all information in a database be cast in terms of values in relations. Such formal and yet simple approach to data management sparked the mass adoption of relational database management systems in the industry. From then, the relational model remains the most fundamental model in the commerce of data. Though other alternatives (e.g., graph model and object model) or extensions (e.g., object-relational model) had been introduced, the underlying concept of mainstream database seems to be extracting some form of *structure* as a mean to manage and to process data. Thus, for some relational purists, it is blasphemy to accept revisions to such a time immemorial concept and yet current trends seem to be proposing precisely that.

Given that computer scientists have somehow always been dealing with data size that is too large, the fact that the adjective "big" is assigned to this particular paradigm does suggest certain degree of grandeur to the scale of data in question. Indeed, as previously mentioned, the data currently handled is already of petabyte-scale while, at the time of writing, the largest magnetic disk drives remain in the terabytes range. Moreover, the CAGR of the disk areal densities is projected to be about 19% from 2011 to 2016 (Fang, 2012) while the CAGR for data is projected to be 53% over the same period (Nadkarni and DuBois, 2013). If data size is the only issue, then the entire Big Data paradigm could have been resolved with a distributed storage solution; however, the changes do involve other dimensions that challenge traditional data management tools, particularly when the operations go beyond storage and retrieval (i.e., data analytics).

Typical description of the Big Data paradigm begins by identifying N "V-

word" dimensions, where $N \geq 3$; each dimension measures one aspect of the data handled such that the current state of digitalization is represented by the perpetual augmentation along all the axes. As expected, one of the dimensions cited is always *volume*, depicting the growth the data generated. The basic three dimensions (Douglas, 2012) definition also includes *velocity*, depicting speed of data generation, and *variety*, depicting the growth of unstructured data. Other definitions include dimensions such as *variability* (variance in meaning, in lexicon), *value* (industrial benefits), *veracity* (degree of correctness) and *visualization* (importance of graphical aggregation). However, given the unbounded extent of interest, trying to classify Big Data from a data-centric approach is almost like trying to know the "unknown unknowns"[5]. Instead it may be easier classify the novel industrial needs so as to understand the scope of Big Data. Cohen et al. (2009) identified three new aspects of data management and processing: *magnetic*, *agile* and *deep* (MAD). The authors intended them to be used to classify the skills set of a modern data analyst but when inversely applied, they also happen to be a succinct classification of the current industrial needs:

Magnetic sourcing. Due to the structured mentality towards data management, traditional data warehouses have an inclination towards processing "clean" data; thus in contrast, unstructured or semi-structured data has poor affinity under these systems. However, as evident in recent trend, regardless of causality, unstructured data is the principal driver of data growth; therefore, modern data management needs to be magnetic in that it should be able to attract and accommodate these "uncleaned" data sources.

[5] As in the (in)-famous "There are known knowns"-speech made by then United States Secretary of Defense, Donald Rumsfeld in 2002.

Agile processing. Traditional data analysis requires elaborate resource planning that may take multi-months preparation. Given that data acquisition gets increasingly fast (note the velocity dimension) and varied (note the variety dimension), such sophisticated design and planning phase may no longer be applicable in mission-critical data analysis for *ad hoc* decision making. Thus, modern data analytics have to be more agile to adapt to the rapid pace of changes; in particular, there is advantage now for data preparation to be kept minimal.

Deep analytics. With the expanded data sources, which are also increasingly more varied, data analytics have correspondingly become more sophisticated, possibly beyond that of traditional online analytics processing (OLAP) and data cube operations (e.g., slice, dice, roll-up). Such deeper analytics are often beyond the assistance of structure extractions and pre-computations. Furthermore, the excessive volume of data being analyzed makes deeper analytics particularly challenging.

The advent of relational database management systems promoted activities of business intelligence to center around the structuring of data. However, while the data model and the supporting computer system may be scaled to encompass the Big Data paradigm, the surrounding human activities already seem to be bursting at the seams; after all, it is well-known that *humans are not scalable*. All three aspects of the MAD classifications actually challenge precisely the "human"-aspect of the data analytics, thus providing considerable legitimacy to the revision suggested by the Big Data paradigm.

1.1.2 Cloud Computing

Cloud computing is perhaps the most fuzzily defined among all the recently popularized concepts. One reason for such ambiguity may be due to the fact

that similar or related notions have always been in development throughout the history of computer science. Each of these notions has now somehow become associated with cloud computing in one way or another. Some of the preceding developments include the following:

Utility computing. The most ancient notion of cloud computing most likely comes from the suggestion of utility computing by John McCarthy in 1961 (Garfinkel, 1999). The basic philosophy is to let computational resources be available under a "pay-per-use" basis much like public utility; the intention is to maximize their productivity. The feasibility of such a concept lies in the economies of scale and the exploitation of shared services via resource scheduling. Since then, computer science researchers have come a long way to materialize this vision to some extent with the current state of cloud computing.

On-demand services. The nomenclature of cloud computing frequently includes various "-as-a-service" hosted software architectures of different abstractions (e.g., *platform*-as-a-service, *software*-as-a-service, *database*-as-a-service) (Sakr et al., 2011). The basic idea is to apply the principle of *separation of concern* (Dijkstra, 1982) at the enterprise level such that various aspects of a system may be hosted by external service providers; this may be considered in some ways as utility computing being conducted at the enterprise level. Despite the common association with cloud computing, on-demand services actually predate that of cloud computing; as early as 2001, the industry of application service providers (ASP) is already a multi-billion dollar market (Tao, 2001), indicating that outsourcing of part of a system has been well incorporated into enterprise practices. Perhaps the ex-

periences of ASPs serve indirectly as a lead-in for cloud computing in terms of architectural integration and system implementation.

Distributed computing. Any study of processing and operations within a networked system can be considered as distributed computing, thus distributed computing is actually a very mature area of research. And in recent years, this field seems to have become the centerpiece of all computing disciplines. The main contributing factor for this phenomenon may very well be simply necessity due to the massive amount of data to be handled in operation (Sakr et al., 2011). Facing data size of limit-breaking scale, parallel solutions offer performance match-up where sequential ones fall short. Perhaps, this is the reason for the frequent tie-in between distributed computing and the Big Data paradigm. As cloud computing is deployed over an array of commodity servers (i.e., horizontal scaling), its operations are almost definitely based on some distributed solutions. Therefore, a cloud system may be deemed as a very large manifestation of distributed computing.

The above mentioned notions are by no means an exhaustive listing of all that is related to cloud computing. Nevertheless, it is noteworthy to indicate that it is the nature of cloud computing to seek to encompass all these notions and thus share their philosophies. Also, the descriptions are merely high-level gross overviews of the subject matter; part of the importance of cloud computing is the innumerable amount of details, be it technical, economical or even legal, that comes into play to bear fruit to the cloud computing that we know of. Notable critical technological improvements that catalyzed the development of cloud computing include improvements in hardware virtualization (Manohar, 2013), adoption of service-oriented

architecture solutions (Duan et al., 2012) and vastly improved network connectivity (Kachris and Tomkos, 2012). Each of these improvements deserves a detailed coverage that is relative to their importance but unfortunately, this has to be skipped for the sake of brevity.

1.1.3 NoSQL

As previously mentioned, the relational data model coupled with ACID-compliant relational database systems has been the principal platform for data management and data analytics. Since its establishment as the staple diet for enterprise system developments, attempts were made to extend or to replace the model for various systemic gains but they often resulted in limited adoption; that is until recently. With the introduction of the Big Data paradigm and the corresponding need to massively scale data management horizontally, the relational data model and the ACID transactional properties become rather restrictive for some operations. Thus, the NoSQL[6] movement began to gain traction in mainstream database systems; the movement advocates the relaxation of traditional data model and also processing guarantees to some extent in exchange for the qualities to cope with augmented amount of data.

The central ideology of NoSQL is the use of looser consistency model (i.e., eventual consistency) as a mean to increase horizontal scaling of the system. The proponents of this movement often cite the Brewer's Theorem as a justification for such relaxation, though it remains debatable whether the Brewer's Theorem has been correctly applied (Brewer, 2012). Nevertheless, such an approach is able to achieve scaling beyond that of relational database management systems (Cattell, 2011). Without a governing

[6]Possibly (and hopefully) stands for *Not Only SQL*.

consistency model like that of ACID, the relational data model cannot be sustained well. Therefore, NoSQL systems also employ the use of a myriad of alternative reduced data models (e.g., key/value store, document store and column store) (Hecht and Jablonski, 2011), which differ to some degree from one product vendor to another; this lack of standardization does provoke some very legitimate criticisms on the interoperability of incorporating NoSQL elements into a system (Mohan, 2013).

Through the relaxed consistency and the reduced data model, the operations exposed in a NoSQL system tend to be correspondingly limited (Hecht and Jablonski, 2011); in fact, NoSQL systems typically only allow key lookups, and reads and writes of a data element in contrast to the complex queries or joins of a relational database system. Note that these operations may be considered as *embarrassingly parallel*[7], thus explains how the massive horizontal scalability is achieved. The point here is not to criticize the sacrificial gain of NoSQL systems but rather to note the paradigm shift in the focus of operations. Traditional relational database systems are meant to be generalized solutions, allowing a wide range of queries from simple create-retrieve-update-delete (CRUD) operations to complex mathematical analysis. However, with the Big Data paradigm shift, executed operations have become increasingly specialized, resulting to the maladjustment of traditional systems (Stonebraker et al., 2007). One particular specification is precisely the need for simple CRUD operations to scale and achieve wide availability, which results in the rise of NoSQL. Another is the need for an easier way of expressing deeper processing (note the MAD properties) over a massive scale (Ordonez et al., 2010), which will be covered in the following section. In any case, it may very well be that the NoSQL movement is

[7] A *embarrassingly parallel* problem is defined as one for which little effort is required to separate the problem into a number of parallel tasks.

simply *a square peg satisfying a square hole that we have on hand*; as we shall see, this is a recurring theme throughout this chapter.

In a distributed system, especially a web-scale one, data replication[8] is perhaps the only practical technique currently available to controllably implement some form of reliability and fault-tolerance into the system operations. Adopting a looser consistency model means that NoSQL systems favor naturally the use of asynchronous resolution of inconsistent replicas as opposed to eager replicas synchronization. Note that the use of lazy replication is not flawless; in general, lazy replication suffers from reconciliation rate that is polynomial to the system size (Gray et al., 1996). However, recall that the specialization of operations is one of the trademarks of modern systems; under specific context, lazy replication can produce remarkable performance in actual production clusters (DeCandia et al., 2007).

1.1.4 MapReduce

Note that any form of operation of which the processing logic is separated into distinct tasks located at different sites can be considered as a form of distributed processing (Özsu and Valduriez, 1999, Chapter 1); such classifications include distribution according to functionalities and/or controls. However, due to the advent of the Big Data paradigm, *data-distributed processing* has implicitly become synonymous with this umbrella term; data-distributed processing refers to the distribution of processing logic according to the horizontally-partitioned data elements without distinction on the nature of the processing. With the size of data handled, otherwise simple operations (e.g., text searches and simple aggregations) become prohibitively

[8] On a side note, the discussion on replication can be seen as one of the pioneer discourses on the subject of availability versus consistency (Bernstein and Goodman, 1984).

heavy. In order to alleviate this workload due to input size, investigations into exploiting data-distributed processing solutions led to the development of the MapReduce framework (Dean and Ghemawat, 2008).

The MapReduce framework began as a data processing framework used internally by Google for parallel processing over immensely large data sets. It is said that the framework came about because developers at Google noticed many of the required data processing jobs may be accomplished with very similar steps (i.e., a distributive *map* phase followed by a local aggregative *reduce* phase), which gave the framework its iconic name; though as we shall see in later chapters, this phenomenon is hardly coincidental. Given that integrating similar processing jobs under a single framework facilitates the resource and jobs management, the MapReduce framework was created.

Its seminal publication brought about immediate interests and criticisms from both research and industrial circles alike. Since then, the MapReduce framework is gradually being established in the industry as the *de facto* distributed data processing solution for data-intensive applications (Sakr et al., 2013) despite continual questioning on its fundamentals (Pavlo et al., 2009). The popularity of using MapReduce is also significantly promoted by the fact that its most notable manifestation (i.e., Apache Hadoop) is an open source software suite and is also freely[9] available for all. Detailed examination of the MapReduce framework will be covered in later chapters; the interest of this section is to examine the impact that such a framework has made in the database community.

As previously mentioned, the MapReduce framework falls precisely within a focus of specialized operations that has gained much attention under the Big

[9] As in free beer.

Data paradigm: web-scale deep processing. Though the MapReduce framework has frequently been criticized for its lack of efficiency[10] (e.g., Anderson and Tucek, 2010; Pavlo et al., 2009; Rowstron et al., 2012), it is undeniable that the MapReduce framework achieves unprecedented horizontal scaling as indicated by previously quoted statistics on Hadoop cluster size; though that too has received some criticisms (Appuswamy et al., 2013). In addition, it is noteworthy to indicate that the MapReduce framework brings about not just the raw processing capability; there is actually much industrial emphasis on the elasticity and the fault-tolerance of the MapReduce processing model (Jiang et al., 2010; Lee et al., 2011; Sakr et al., 2013). Given the massive data and cluster size in the current context, processing failure is now taken as a relatively common phenomenon; this is arguably a novel viewpoint highlighted by the introduction of the MapReduce framework. Should the processing halt or restart upon singly failures, the framework cannot be deemed as "functioning" under the current context; therefore, the fault-tolerance mechanism of MapReduce framework whereby only failed tasks are restarted becomes a critical inclusion in the design of modern processing systems (Yang et al., 2010). Also, with the spread of resources consumed for processing, the elasticity provided by the MapReduce processing model is a much required relief to the immense task of resource management; with elastic processing, resources can be allocated through optimized scheduling, thus enabling better processing throughput.

The use of the MapReduce framework has close associations with the previously discussed concepts, though, as usual, it is difficult to determine the causality of the influence. Under the Big Data paradigm, it seems too much of a coincidence that MapReduce framework can be seen as the exact tool

[10]To be fair, it is Hadoop (and not Google's MapReduce framework) that is used for experiments most of the time.

required for modern data analytics; its semi-structured approach to data processing and the expressiveness of its programming model satisfy precisely the MAD qualities (Herodotou et al., 2011). Strictly speaking, even though the data model of MapReduce (i.e., key/value pair) corresponds to that of some NoSQL systems, it cannot be classified as NoSQL storage system since most common implementations do not allow record-level CRUD operations; at best, it can be seen as a "data analytics branch" of the NoSQL movement. Nevertheless, it is interesting to note that many NoSQL systems also offer MapReduce application programming interfaces (API) (e.g., MongoDB, Stratosphere and Riak) but this is perhaps an effect of the popularity of such processing mechanism. The popularity of the MapReduce framework, particularly its processing model, has become so widespread that many cloud vendors offer pre-configured Hadoop architecture optimized as a cloud utility (e.g., Amazon Elastic MapReduce).

1.2 Desirable System Qualities

Much can be gleaned from the discussion of these recent developments. Though it is not the position of this thesis to scrutinize their popularity or their justification, it is noteworthy to mention that their widespread continual adoption does infer certain boons in their constitutions. Among all the desirable qualities that has led to the preservation of these novel trends, this thesis identifies three core qualities that can be deemed as quintessential of the architecture of a modern data processing system: massive horizontal scalability, elastic resource consumption and robust system operations.

1.2.1 Scalability

One thing that all the previously mentioned trends have in common is the insistence on the handling of large data size. Recall that volume is one of the basic dimensions of the Big Data paradigm. Also, in a way, the utility nature of cloud computing can be seen as a mean to lower the entry-level of acquiring large data management through the economies of scale. And the advent of NoSQL systems and MapReduce systems is precisely due to the need to handle specialized operations on large data sets.

Note that some have disputed the emphasis on the data size (e.g., Rowstron et al., 2012); the argument is that even with the Big Data paradigm shift, most processing jobs are not "big". Indeed, Ananthanarayanan et al. (2012) revealed that about 90% of the processing jobs consist of input size of 100 gigabytes or less. Appuswamy et al. (2013) further indicated that the median size of processing jobs of two identified analytics production clusters is under 14 gigabytes. Processing tasks of these sizes can usually be handled rather comfortably with a relatively small cluster or even with just a single dedicated machine. However, it is undeniable that at least some jobs are still overwhelmingly large. This might seem to some as pushing the "nobody ever got fired for buying x" argument, where x is some product that requires considerable financial investment. Nevertheless, this thesis holds the position that it is of academic interest for computer scientists to devise solutions for the most adverse scenario, especially since through empirical observations on current trends, it is evident that the digitalization of the world is accelerating beyond the bounds of hardware; recall that the CAGR of data is projected to be almost three times that of disk areal densities.

For handling large data set, currently, distributed solution over a cluster of computers (i.e., horizontal scaling) is often preferred over the centralized processing with a single machine of significant capability (i.e., vertical scaling). One reason is perhaps to account for possible (and from the standpoint of this thesis, very probable) future growth of data set of interest. Understand that the capability of a distributed system can often be augmented through the addition of more resources, granted that the improvement is almost definitely sub-linear to the amount of resources added. On the other hand, improving the capability of a single machine is limited by the hardware technology available at that time and thus present as an unknown factor. Therefore, a system solution intended to meet the demands of current and future context should ideally possess the ability to scale horizontally.

1.2.2 Robustness

Perhaps one of the greatest problems of adopting massive horizontal scaling is that the aggregated *mean time between failure* (MTBF) may decrease with each added resource. Such a seemingly paradoxical phenomenon occurs when the overall wellbeing of the system depends entirely on the sum of all the states of the participating machines:

Proposition 1.2.1

As the cluster size increases, the minimum MTBF approaches zero.

Proof. Suppose that the machine failures are independent and follows a Poisson distribution, which is a legitimate simplification under the assumption of the *Law of Rare Events*[a], then the MTBF of each machine follows that of exponential distribution[b]. Let $X_1, X_2, \ldots, X_n$ be

n random variables representing the MTBF where

$$X_i \sim \exp(\lambda_i),\ \forall i = 1, 2, \ldots, n$$

Note that $E(X_i) = \frac{1}{\lambda_i}$ by the property of exponential distribution. Consider the minimum of all the variables, $X_{\min} \triangleq \min(X_1, X_2, \ldots, X_n)$:

$$\begin{aligned}
\Pr(X_{\min} > x) &= \Pr(\cap_{i=1}^{n} X_i > x) \\
&= \prod_{i=1}^{n} \Pr(X_i > x) && \text{by independence of variables} \\
&= \prod_{i=1}^{n} e^{-x\lambda_i} && \text{by } CDF \text{ of } \exp(\cdot) \\
&= e^{-x\sum_{i=1}^{n}\lambda_i}
\end{aligned}$$

Therefore, $X_{\min}$ also follows exponential distribution:

$$X_{\min} \sim \exp\left(\sum_{i=1}^{n}\lambda_i\right)$$

Furthermore, note that this observation has the ramification that the expected mean $E(X_{\min}) = \frac{1}{\sum_{i=1}^{n}\lambda_i} = O(\frac{1}{n})$. Extrapolating this value, this means that $\lim_{n\to+\infty} E(X_{\min}) = 0$, which is to say that as the cluster size increases, the minimum MTBF approaches zero! □

[a] Hopefully, it is legitimate to consider machine failures as "rare" events.

[b] To be really specific, the machine failures actually follow the *Weibull distribution* (Weibull, 1951) due to possible varied failure rate (e.g., "infant mortality" and aging hardware); however, such detailed exposition is beyond the scope of this discussion and besides, the conclusion drawn is actually the same.

As inferred from the mathematical model, such a horizontally scaled system will not be practical as its size becomes its Achilles' heel, which is ironic since the selling point of horizontal scaling is precisely its grandeur. Thus, the only solution is to ensure that the overall wellbeing of the system *does not* depend entirely on the sum of all the states of the participating machines.

From the previous discussions, it can be observed that this is generally the mentality adopted.

As previously mentioned, data replication is the primary technique used to ensure robust operations over horizontally scaled system. To put into the perspective of this section, replication allows operations intended for a failed machine to be redirected to the machine holding the appropriate replica such that on the whole, as long as at least one replica persist, the entire system will not be burdened by singly machine failures. However, the premise of the feasibility of such a mechanism lies again on the simplicity of the operations (e.g., CRUD operations). As expected, robust processing requires another level of fault-tolerance on the processing model itself.

The processing model of MapReduce framework is known to emphasize not just on its scalability and elasticity, from the original design, the developers has already built in fault recovery mechanism so as not to inhibit the completion of the job. Understand that fault-tolerant processing is not a groundbreaking concept (Aviziens, 1976); before the introduction of the MapReduce processing model, fault-tolerance has generally been *transparently* implemented. What makes the MapReduce framework particularly robust and efficient is that the fault-tolerant mechanism is, in a way, *explicit* in its processing model (i.e., idempotence of tasks) such that the model is able to retain as much intermediate work as possible (Dinu and Ng, 2012).

Given the horizontal extent of modern systems, fault-tolerance has actually become a critical prerequisite for the modern system architecture. As evident from the success of the MapReduce processing model, it seems acceptable to lose the transparency of system fault-tolerance for augmented attention on the efficiency of the overall processing.

1.2.3 Elasticity

The ability to dynamically adjust resource usage based on varying workload or resource allocation is generally known as elasticity. While this is a basic computing notion, elasticity has received renewed attention recently due to the scale of resources currently handled. It is noteworthy to mention that even though elastic resource consumption often co-occurs with horizontal scalability in the architecture of a modern data processing system, strictly speaking, they are actually orthogonal qualities; typically, scalability is taken as a planned providence while elasticity is more of a reactive behaviour (Fardone, 2012). From the discussion of recent trends, it can be observed that elasticity can be incorporated at different levels; a system deployed on a cloud platform often demonstrates the elasticity of a distributed system (Suleiman et al., 2012) while the MapReduce processing model exhibits the property of elasticity in processing (Jiang et al., 2010).

Elasticity has often been identified as one of the trademarks of cloud computing, even though some has suggested that elasticity is a side-effect from the utility nature of hosted services (Bias, 2010); in any case, it is undeniable that the dual qualities of scalability and elasticity are some of the main selling points of on-demand services. From the cloud subscriber's point of view, horizontal scalability coupled with "pay-per-use" pricing model allows better enterprise planning; in fact, this prevents precisely the mongering of fear, uncertainty and doubt (FUD) by overzealous ASP products salesmen. Furthermore, given that system workloads can be rather bursty depending on the nature of the industry (e.g., Ali-Eldin et al., 2012; Brebner, 2012), from the developer's point of view, implementing elasticity into a system allows more timely reaction to the required changes in resource allocation.

Now, most operations required of a cloud system are relatively short-lived and of direct nature (e.g., CRUD operations), which explains the compatibility of NoSQL systems on cloud platform (Konstantinou et al., 2011). However, when the operations are more complex (e.g., deep analytic jobs), the processing may have poor elasticity, Therefore, the elastic processing model of the MapReduce framework has additional appeal; with such an elastic processing mechanism, the system may scale even in the midst of processing a job. Moreover, because of the economy, and hence popularity, of hosted services, multi-tenancy of varied job profiles is a common phenomenon of the cloud platform; thus, elastic processing allows more optimal resource management at the vendor's side through appropriate scheduling. Even with a private cluster, recall that the MapReduce framework was devised for integrating jobs management, therefore having an elastic processing permits better throughput of multiple jobs over the resource usage. Therefore, there is actually a critical incentive in incorporating elasticity into the architecture of a modern data processing system.

1.3 Structured Peer-to-Peer Architectures

Currently, the predominant architecture adopted for modern data processing system is the *master/workers architecture* (e.g., Chen et al., 2012; Das et al., 2013; Isard et al., 2007). Simply said, the master/workers architecture consists of an assigned processing site[11] (i.e., the master site) that has unidirectional control of all the other participating sites (i.e., the worker sites) for the coordinations of operations in the system. Note that other than for the sake of simplicity (e.g., Dean and Ghemawat, 2008), there are

[11]The term *processing site* (or just *site*) is used to refer to a generic encapsulated processing unit that is logically distinguishable from one and other.

arguably not much other incentives in adopting the master/workers architecture; moreover, "textbook" computer science will dictate that the single master design necessarily presents eventual limitations (e.g., single point of failure and communication bottleneck). In fact, additional implementations often have to supplement the master/workers architecture so as to incorporate additional desirable system qualities. For example, in order to prevent the master site from being overwhelmed by massive horizontal scaling, delegation of control can be put in place to spread the loci of communication (e.g., Apache, 2012; Hindman et al., 2011). Also, in order to assure continual existence of a master site (i.e., high availability), hot standbys are often maintained to allow real-time fail-over whenever the master site fails (e.g., Myers, 2012).

As a disclaimer, it is critical to emphasize that the importance of simplicity should not be therefore undermined. In general, master/workers architectures elegantly segregate the system control from the processing mechanism; it is precisely such functionality-based distribution that facilitates desirable system qualities to be injected. However, as the *cliché* goes, "[solutions] should be made as simple as possible but not simpler"; therefore, it is perhaps profitable to investigate more involved alternatives through the perspectives of the previously discussed desirable qualities.

At the opposite end of the spectrum, peer-to-peer (P2P) architectures differ from master/workers architectures precisely in that there are no non-trivial distinctions on the role played by the sites participating in a P2P architecture. Without a centralized controller site, the participating sites have to keep track of each other, thus constructing a *logical network overlay* whereby each participating site maintains a small set of links to some other sites (i.e., *fingers*). Naturally, the ensemble of the fingers will form a *strongly connected*

graph. The ramification of such a construction is that all forms of control mechanisms have to be implemented in a decentralized manner (i.e., based on some graph algorithms); these mechanisms include data location, message passing and processing coordination, which can otherwise be directly controlled by the master site in a master/workers architecture. Therefore, it may seem that P2P architectures will have indeterministic operational performance; indeed, many P2P architectures support only operations of limited scale (Androutsellis-Theotokis and Spinellis, 2004). However, if some form of structured symmetry is enforced on the overlay, P2P architectures can actually provide many systemic qualities; these architectures are commonly known as *structured P2P overlays*. This thesis maintains the position that structured P2P overlays are able to sustain the previously discussed desirable system qualities:

Scalability. Without being limited by the capability of a single site (i.e., the master site), a decentralized architecture such as that of a P2P architecture typically allows an even higher number of participating sites simply because the system state maintenance is shared by all the sites. For example, Rasti et al. (2006) indicated that the Gnutella network grew beyond three million sites in 2006. For structured P2P overlays, such scalability of participation is retained; moreover, the operations are typically known to degrade only sub-linearly (e.g., logarithmically) to the overlay size (e.g., Ratnasamy et al., 2001; Rowstron and Druschel, 2001; Stoica et al., 2001; Zhao et al., 2004).

Robustness. As previously mentioned, the operational environment of structured P2P overlays is particularly unstable, therefore fault-tolerance of the system and its operations is usually one of the foci of the overlay.

Note that due to its decentralized nature, a structured P2P overlay does not suffer from single point of failure unlike its centralized counterpart (i.e., master/workers architecture). Moreover, data replication is usually part and parcel of the design of the overlay (e.g., Ratnasamy et al., 2001; Stoica et al., 2001).

Elasticity. Understand that structured P2P overlays are created for an environment that is much more malignant than that of a computer cluster; under the P2P paradigm, site displacements are expected to be much more dynamic and frequent (Androutsellis-Theotokis and Spinellis, 2004). Therefore, these overlays are designed for very efficient and robust adaptation to unstable site population. This is possible partly because of the distribution of system state maintenance; any update to the system state often affects only a small constrained subset of participating sites. Hence, the relatively fluid changes to resource allocation of a computer cluster will not pose a problem to structured P2P overlays.

It is interesting to point out that unlike the case for the master/workers architecture where these qualities have to be intentionally injected through *deus ex machina* (i.e., extra-architectural) reinforcement, they are already inherent in the design of structured P2P overlays. Therefore, there are several notable examples of modern data processing systems that have adopted some form of structured P2P overlays in their designs (e.g., DeCandia et al., 2007; Kallman et al., 2008). However, most of the time, these systems treat the P2P overlay as an embedded substrate and as not the primary defining architecture; this is to say that the individual characteristics of the structured P2P overlays are not really being exploited in any way in the

systemic operations. Here lies the central theme of this thesis, which shall be elucidated in the following section.

However, to be fair, things are not as simple as depicted so far. As previously mentioned, it is relatively direct to introduce elasticity and robustness when operations are simple and short-lived; therefore, the structured P2P overlays embedded in modern data processing are used mainly for CRUD operations. Recall that a focus of modern data processing centers around deep analytics, therefore it is particularly intriguing to question if a web-scale data processing model can be decentralized and if such a model may retain elasticity and robustness.

1.4 Contributions

This thesis is a documentation of a broad-scale investigative research on structured P2P overlays in the current changing world. Using the scalability-robustness-elasticity triplets as the principal dimensions, the intent is to uncover possibilities whereby structured P2P overlays may be used as the underlying architecture of a modern data processing system such that performance may be augmented. The collective contribution of this research is two-fold. Firstly, the results obtained from the experiments conducted are the direct testimony of the suitability of structured P2P overlays in handling various profiles of data processing. Secondly, even though the prototypes implemented are products of academic investigations, there are reasons to believe that they may be the basis of new data processing systems given the time-tested advantages of structured P2P overlays. Specifically, within their individual dimensions, the contributions of this thesis are documented in the following. Work of this thesis can also be found in (Goh and Tan, 2013, 2014)

1.4.1 Scalability: The Katana Framework

While structured P2P overlays are known to exhibit demonstrable scalability in their systemic operations (i.e., CRUD queries), a distributed programming model, like that of the MapReduce framework, on P2P overlays is rather unheard of. As such, structured P2P overlays are not known for executing deep queries (e.g., analytics) despite their obvious architectural advantages. The challenge lies in that fact that such a programming model has to be decentralized as well, so as to conform to the P2P philosophy. The *Katana* framework is a novel creation that seeks to provide such a feasible solution.

The Katana framework consists of a generalized distributed programming model much like the MapReduce programming model except that it is entirely decentralized; this possibility comes from the identification of certain algebraic structure within commonly known structured P2P overlays. The result of a formal approach is that many popular structured P2P overlays may be adapted to use the Katana programming model. At the same time, due to the affinity between the programming model and the formal structure of the overlay, the Katana programming model exhibits augmented expressiveness, as compared to the MapReduce programming model, thus providing better execution performance. Furthermore, the Katana programming model is shown to be able to emulate any MapReduce algorithm.

1.4.2 Robustness: Hardened Katana

Fault-tolerance has become indispensable in modern data processing systems especially when horizontal scalability is the mainstream consideration

in the design of system architecture. With regards to implementation, one can argue that the more challenging aspect of fault-tolerance in an architecture is the assurance of robustness during processing, especially if query is deep and if agility is of utmost concern (i.e., MAD qualities). In this regard, the MapReduce framework shines as it can be said to be a framework that implicitly embeds fault-tolerance into its processing with its programming model, which is heuristically known to be expressive. Such a mechanism can be generalized into an abstract model of fault-tolerant processing called the *COVER-CHARGE protocol* (CCP).

If a processing task conforms to the requirements of the CCP, the protocol describes a demonstrable fault-tolerant mechanism of processing that task. It happens that the protocol is applicable to many system-wide operations of a structured P2P overlay. *Hardened* Katana is an extension to the Katana framework that seeks to incorporate the CCP into the Katana programming model. The approach taken by the hardened Katana framework can be said to be superior to some of the common reinforcement techniques (e.g., fast fail-over backup site) in that the robustness induced within the enhanced framework is derived directly from the architecture, and not some *deus ex machina* mechanism.

1.4.3 Elasticity: Elastic MapReduce Execution

The industry has known the MapReduce framework for its massive horizontal scalability, thus promoting wider and more intensive usage. However, with increasingly larger deployments, the MapReduce framework begins to face technical deficiencies in its execution architecture. In order to cope with such limit-pushing amount of resources, there are independent developments of supplementary frameworks (e.g., Apache, 2012; Ching et al.,

2012; Hindman et al., 2011) that isolate resource management from the job coordinations. While these resource managers are meant to support job executions, they also expose potential increased elasticity in job execution that has not been fully exploited by the current state-of-the-art execution architecture.

The *Elastic MapReduce Execution* (EMRE) is an enhanced execution architecture for the execution of MapReduce jobs. It leverages on a structured P2P overlay to induce elasticity into the job execution without compromising on the fault-tolerance. As opposed to the current execution architecture where the processor elements (i.e., the worker sites acting as mapper or reducer) are relatively oblivious to the overall execution, the worker sites under EMRE *are* the execution architecture itself in a P2P manner. EMRE presents an example where even if the global architecture is of master/workers relationship, embedding a structured P2P element in the operations of the system can inherit the qualities of structured P2P overlays, thus vastly improving the overall performance.

Note that the improvement on the execution architecture is entirely transparent such that compatibility of job definition can be retained under EMRE. On top of the induced elasticity and the backward compatibility, because a decentralized, distributed index (i.e., the structured P2P overlay) is used, the execution architecture is pipelined such that execution under EMRE also experiences improvement in running times.

1.5 Organization

This chapter is essentially a broad overview of current developments and aggregated observations; its purpose is to serve as a prelude to the conducted research work proper. The rest of the thesis is organized as follows:

- Work that is related to the research development is presented in Chapter 2. Two main areas of work are covered in this chapter. Firstly, structured P2P overlays are presented. The concept of *consistent hashing*, which is employed in many structured P2P overlays, is introduced. Then, two specific structured P2P overlays that are used in the implementations are presented: Chord (Stoica et al., 2001) and BATON (Jagadish et al., 2005). This is followed by an introduction of *bounded broadcast*, a technique used in some structured P2P overlays to broadcast efficiently. The second area of work presented is a more detailed description of the MapReduce framework and its related concept (e.g., programming model and system architecture).

- Chapter 3 is dedicated to the discussion of the Katana framework. The Katana programming model is formally introduced, followed by a derivation of a mechanism to realize the model on typical structured P2P overlays. The overview system architecture with the details of some of its more important internal mechanisms are briefly described so as to demonstrate the operations within the Katana framework. Experimental results from extensive explorations are finally presented; these results consist of comparisons between algorithms on the Katana framework, algorithms on the MapReduce framework, as well as emulated MapReduce algorithms on the Katana framework. In all the studies, the algorithms on the Katana always outperform that on the MapReduce framework.

- The issue of robust processing is explored in Chapter 4. The mechanism employed by the MapReduce framework to handle site failures during processing is first reexamined to extract the premises in which the fault-tolerant processing of MapReduce is built on. From the iden-

tified premises, the CCP is constructed and it is demonstrated to be a generalization of the fault-tolerance mechanism of the MapReduce framework. The hardened Katana framework is introduced as an extended version of the Katana framework with incorporation of the CCP into its internal operations. Comparative experimental studies indicate that the overhead incurred for the corrective measures taken by the COVER-CHARGE protocol for operations of the hardened Katana framework is comparable to, if not lesser than, that of the MapReduce framework.

- The EMRE architecture is covered in Chapter 5. The chapter starts with an overview of how the current state-of-the-art MapReduce execution architecture suffers maladjustment to the newly introduced execution environment. Then the core system components of the EMRE architecture are described with respect to their roles (i.e., master site and worker site) in EMRE; in addition, a modified version of BATON that is more suitable in the context of MapReduce is introduced. The job execution under EMRE is described with details, in which another foreign technique used in EMRE is introduced: work stealing. Work stealing helps to even up the load imbalance that is inherent in all structured P2P overlays. Through a series of benchmark executions, experiments indicate that MapReduce jobs run faster under EMRE as compared to the MapReduce framework with varied amount of available resources under various execution conditions (i.e., stable and dynamic environments).

- Finally, Chapter 6 concludes the thesis with a few afterwords on the recent developments and on the contributions offered by the research works presented.

Chapter 2

Related Work

In this chapter, some important precedent works are presented; the works covered are exploited, in one way or another, by the developments documented in later chapters; therefore, it is of academic interest to lay out a compilation of the central concepts of these works as a prelude to the presentation proper of the research work conducted for this thesis.

2.1 Structured Peer-to-Peer Overlays

2.1.1 Consistent Hashing

While literatures on structured P2P overlays are rather diverse in their implementation approach, majority of them distributes data (and index structures) via distributed hash table (DHT). With the DHT approach, data elements are identified and distributed by deterministic hashing on their identity values (i.e., primary key in the relational model and fully qualified filenames for files); in this case, fair distribution of the workload

is largely dependent on the uniformity of the hash function. The table of hash values itself is distributed across the processing sites to form the routing table that directs the message passing. The various DHT-based P2P overlays differ largely in the manner the hash tables are distributed but in general, they follow the principles of *consistent hashing* (Karger et al., 1997) allowing relatively dynamic participation of processing sites without resorting to much rearrangement due to changes of the network parameters.

Strictly speaking, the concept of consistent hashing does not dictate the way data elements are allocated to the processing sites; it formalizes a number of properties that when achieved, can have some performance guarantees. In the following, the formal definition of consistent hashing will be presented; note that there are some refinements made on the authors' original presentation.

Let $\mathcal{I}$ be the set of data elements and $\mathcal{B}$ be the set of *buckets*; a *view* v is defined to be a non-empty subset of $\mathcal{B}$, therefore $v \in \mathcal{V}_{\mathcal{B}} \triangleq \mathcal{P}(\mathcal{B})\backslash\emptyset$. Given $v \in \mathcal{V}_{\mathcal{B}}$, a *ranged hash function* is defined as $f_v : \mathcal{I} \to v$; the indexed collection of functions $\mathcal{F}_{\mathcal{B}} \triangleq \{f_v \mid v \in \mathcal{V}_{\mathcal{B}}\}$ is called a *ranged hash family*. A hashing scheme for a particular $\mathcal{I}$ is defined entirely by $\mathcal{B}$ and the corresponding $\mathcal{F}_{\mathcal{B}}$; it is considered as consistent hashing if it satisfies the following properties:

Balance. $\mathcal{F}_{\mathcal{B}}$ is balanced if given $v \in \mathcal{V}_{\mathcal{B}}$, with high probability,

$$\forall f \in \mathcal{F}_{\mathcal{B}}, \forall b \in \mathcal{B}, \ \frac{|f^{-1}(b)|}{|\mathcal{I}|} = O\left(\frac{1}{|v|}\right)$$

A balance distribution is actually a standard requirement on any hash function. However, a balanced $\mathcal{F}_{\mathcal{B}}$ further assures that regardless of

the view, the data elements will remain well distributed among the buckets.

Monotonicity. $\mathcal{F}_{\mathcal{B}}$ is monotone if $v_1 \subseteq v_2 \subseteq \mathcal{B}$,

$$\forall i \in \mathcal{I},\ f_{v_1}(i) \in v_2 \Rightarrow f_{v_1}(i) = f_{v_2}(i)$$

The crux of the problem with using standard hash function for decentralized distribution of data elements is that there is no constraints on the data re-allocation upon changes on hash parameters; though, some may define this as a mark of a good hash function (i.e., cryptographic hash function), however this is obviously unacceptable in the distributed context. Therefore, a monotonic $\mathcal{F}_{\mathcal{B}}$ is important as it will assures that data elements migrate only for the preservation of even distribution.

Low spread. Given $V \subseteq \mathcal{V}_{\mathcal{B}}$, spread σ_V of a data element $i \in \mathcal{I}$ is defined as

$$\sigma_V(i) = |\{f_v(i) \mid v \in V\}|$$

The spread of $i \in \mathcal{I}$ tracks the migration of i with varying views. $\mathcal{F}_{\mathcal{B}}$ has low spread if the maximum spread of data elements under an arbitrarily chosen $V \subseteq \mathcal{V}_{\mathcal{B}}$ is low (i.e., $O(\log\ |I|)$). Having a low spread on $\mathcal{F}_{\mathcal{B}}$ means that the distribution of $\mathcal{I}$ remains relatively stable as the view evolves.

Low load. Given $V \subseteq \mathcal{V}_{\mathcal{B}}$, the load λ_V of a bucket $b \in \bigcap_{v \in V} v$ is defined as

$$\lambda_V(b) = \left|\left\{f_v^{-1}(b) \middle| v \in V,\ b \in \bigcap_{v \in V} v\right\}\right|$$

The load of $b \in \bigcap_{v \in V} v$ tracks the changes to the number of items

held at b with varying views. $\mathcal{F}_{\mathcal{B}}$ has low load if the maximum load of buckets under an arbitrarily chosen $v \subseteq \mathcal{V}_{\mathcal{B}}$ is low (i.e., $O\big(\log\ |I|\big)$). The load of $\mathcal{V}_{\mathcal{B}}$ is an alternative interpretation of its spread but taken from the perspective of the buckets; having a low load on $\mathcal{V}_{\mathcal{B}}$ means that the capacity remains stable as the view evolves.

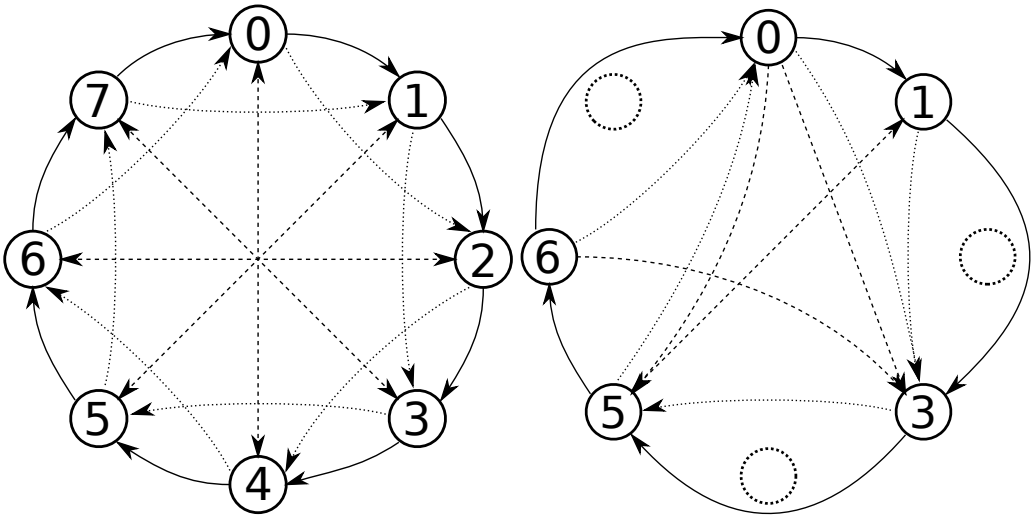
Karger et al. (1997) provided a construction of $\mathcal{F}_{\mathcal{B}}$ whereby two random hash functions are identified: $r_{\mathcal{B}} : \mathcal{B} \to [0, 1)$ and $r_{\mathcal{I}} : \mathcal{I} \to [0, 1)$. Given $i \in \mathcal{I}$ and $v \in \mathcal{V}_{\mathcal{B}}$,

$$f_v(i) \triangleq \arg\min_{b \in v} \big(|r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|\big) \tag{2.1}$$

Note that this construction of $\mathcal{F}_{\mathcal{B}}$ is well-defined if and only if $r_{\mathcal{B}}$ is injective such that $\arg\min_{b \in v} \big(|r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|\big)$ is a singular value. Fortunately, this is a common assumption on hash functions. As it turns out, this is very similar to the hashing scheme adopted by the Chord overlay (Stoica et al., 2001). However, not all hashing mechanisms has to follow such a definition; for example, the Content-Addressable Network (CAN) (Ratnasamy et al., 2001) does not adopt such a scheme though there are reasons to believe that the hashing mechanism of CAN is consistent hashing.

2.1.2 Chord

At the turn of the millennium, a series of structured P2P overlays were introduced into the research circles that brought about renewed interests in DHTs; some examples include CAN, Tapestry (Zhao et al., 2004) and Pastry (Rowstron and Druschel, 2001). Among these work, Chord stood out for its simplicity, robustness and performance as a DHT. For these qualities,



(a) (b)

Figure 2.1: Cayley graph for $(\mathbb{Z}_8, +_8)$ with the generating set $S = \{1, 2, 4\}$ (2.1a) and a corresponding imperfect Chord topology (2.1b)

a myriad of literatures has been established on the Chord overlay (e.g., Lee et al., 2011; Leong et al., 2006; Park et al., 2010; Sánchez-Artigas and García López, 2010).

In Chord, the buckets $\mathcal{B}$ of the processing sites are assumed to inherently unique and can be independently determined (e.g., the IP addresses). Given the difficulty to handle $[0, 1)$, which has the same cardinality as $\mathbb{R}$, the codomains of $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ are approximated with $\mathbb{Z}_N$ where N is a preconfigured system parameter such that $N \geq |\mathcal{B}|$. In a fully-assigned Chord, the processing sites will have a table of routes (i.e., *fingers*) to exponentially-distanced processing sites. In this case, the network is exactly isomorphic to $Cay(\mathbb{Z}_N, +_N, S)$ where $S = \{2^i \mid 0 \leq i < \lfloor \log_2 N \rfloor\}$ (refer to Definition A.3 and Definition A.9). In fact, Figure 2.1a will correspond to the Cayley graph representation of a fully-assigned Chord with eight processing sites and $N = 8$. In reality, the assignment is almost definitely less-than-perfect; therefore the edges that point to inexistent vertices will be directed to the processing site with the smallest identifier that is larger than the intended

one (i.e., the immediate successor) such that the Cayley graph is emulated by each processing site seemingly having multiple identities. For example, in Figure 2.1b, all edges intended for site 4 will have to be directed to its immediate successor (i.e., site 5); conversely speaking, the physical site 5 holds the identities of both site 4 and site 5.

Remark 2.1.1

Recall that all Cayley graphs are vertex transitive (refer to Theorem A.2). Note that a vertex transitive graph has the property that the graph "looks the same" from any vertex of the graph due to the automorphism. This quality has many interesting implications in implementation of network topology (i.e., the overlay). Firstly, for example, load may be uniformly distributed through all sites since each site is indistinguishable from one another meaning there is no topological reason that one site will be burdened more than others. Secondly, vertex transitivity also means that the same algorithm can often be initiated indiscriminately at any site since they are essentially the same; this simplifies the definitions of system-wide operations (e.g., routing and broadcasting). Thirdly, the fact that none of the sites is "special" means that none of the sites is essential and indispensable to the overall functioning of the overlay; this is a very useful quality for the enforcement of fault-tolerant operations.

Due to the inexact emulation, Chord modifies slightly the definition of $\mathcal{F}_{\mathcal{B}}$ of (2.1) for its data allocation. Under Chord, given $i \in \mathcal{I}$ and $v \in \mathcal{V}_{\mathcal{B}}$,

$$f_v(i) \triangleq \arg \min_{b \in v} \Big(dist\big(r_{\mathcal{B}}(b), r_{\mathcal{I}}(i)\big) \Big) \tag{2.2}$$

where

$$\forall x, y \in \mathbb{Z}_N,\ dist(x,y) = \begin{cases} x - y & \text{if } x \geq y, \\ N - y + x & \text{otherwise.} \end{cases} \tag{2.3}$$

Under the definition of (2.2), data elements will be allocated to the immediate successor of an inexistent processing site so as to suit the emulation by Chord of the Cayley graph. In this way, the routing of messages will be same as selecting the routes corresponding to the elements of the generating set S to obtain the destined value. Consider the case of finding a route from site a to site b. Trivially, $b = a +_N a^{-1} +_N b$. Since S is the generating set, $\exists s_1^a, \ldots, s_p^a$, $a^{-1} = s_1^a +_N \cdots +_N s_p^a$ and $\exists s_1^b, \ldots, s_q^b$, $b = s_1^b +_N \cdots +_N s_q^b$. Therefore, the route from site a to site b consists of the iterative following of the fingers corresponding to the following order: $s_1^a, \ldots, s_p^a, s_1^b, \ldots, s_q^b$. Now, by the closure of $+_N$, $\exists c \in \mathbb{B}$ such that $c = s_1^a +_N \cdots +_N s_p^a +_N s_1^b +_N \cdots +_N s_q^b$. A particularity of Chord is the construction of its generating set (i.e., $S = \{2^i \mid 0 \leq i < \lfloor \log_2\ N \rfloor\}$); therefore, if $N = 2^x$ for some x, then every element of $\mathcal{B}$ can be expressed as a linear combination of distinct elements from the generating set, which is rather optimal at the logical level; note that the length of the routing path is thus $|S| = O(\log\ N)$. Under Chord, such a selection is implemented in a decentralized manner by greedily choosing the "largest" possible element in the generating set.

The inserting of a new processing site involves, first, finding the appropriate position by the routing mechanism as mentioned. Then, the new site has to be inserted before the located site due to the emulation; this is assisted by maintaining a route to the predecessor. When found, the new processing site has to share the load with its predecessor and propagate the updates to all the neighboring processing sites, which requires $O(\log^2\ N)$ time. The

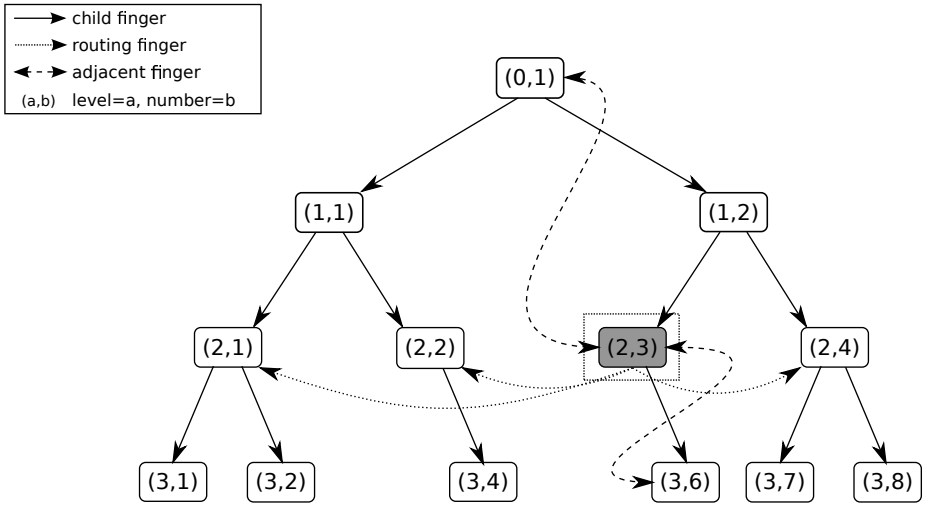


Figure 2.2: BATON with 13 sites and fingers of site $(2, 3)$

removal of a processing site is practically the reverse process.

2.1.3 Balanced Tree Overlay Network

While the literatures on modern P2P overlays are dominated by DHT-based approaches, the Balanced Tree Overlay Network (BATON) (Jagadish et al., 2005) is a unique entity because it is the first P2P overlay based on a balanced sorted binary tree structure with range-based partitioning. As a result, BATON supports both exact matches and range queries on the data elements while DHT-based overlays only support exact matches in their original conceptions.

Each site keeps track of a continuous sub-range of keys that it is governing. In addition, like all P2P overlays, each site in a BATON overlay will maintain a table of routes (i.e., fingers) to other sites; in particular, there are three types of fingers to other sites:

- fingers to its children if they exist,

- fingers to its adjacent sites (where adjacency is determined by the in-order traversal of the tree), and
- selective fingers to sites at the same level.

Figure 2.2 shows an example overlay and the different fingers. Due to the joining algorithm, the tree structure of BATON will always be balanced. Each new left (right) child will share lower (upper) half of the load of its parent, thus maintaining a non-overlapping partitioning over the entire key-space. In this way, as long as the data elements are sorted locally, the adjacent fingers allow us to traverse the data elements in a sorted manner, thus allowing range queries to be done.

Note that a fully-assigned BATON is not really isomorphic to a group. However, it can be said that a fully-assigned BATON emulates a Cayley graph of an integer group $(\mathbb{Z}_N, +_N)$ with $\{2^i \mid 0 \leq i \leq \log\ N\}$ as generating set, much like Chord. The emulation is done by adding a "virtual vertex" together with some "virtual edges" to complete the graph (Lupu et al., 2008); the omission of the site corresponding to the virtual vertex and the routes in the fully-assigned BATON is not an issue because the routing of messages does not require them at all.

Since BATON is ranged partitioned, allocation of data elements is trivial; the routing of messages is done in a similar manner as compared to Chord except that the routes are bidirectional in BATON, thus the length of the routing path is $O(\log\ N)$. As compared to DHT-based overlays, the joins and departures of a processing site in BATON is relatively more complicated. The problem comes from the maintenance of all the routes, particularly the adjacent routes that maintain the sorted order of the processing sites. The mechanism employed is much akin to the rotation of

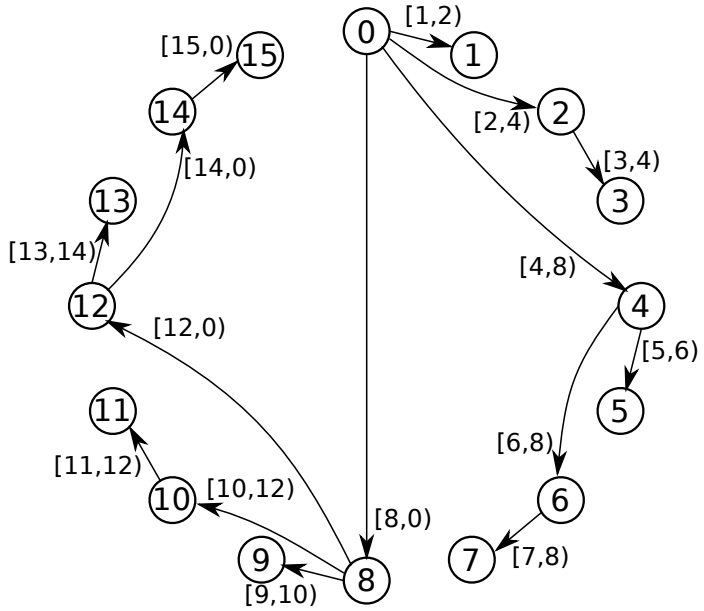


Figure 2.3: Example of bounded broadcast on Chord from site 0

an AVL-tree; the complexity taken for performing joins and departures of processing sites in BATON are both $O(\log\ N)$.

2.1.4 Bounded Broadcast

Bounded broadcast is a technique employed in some DHTs (e.g., Vishnevsky et al., 2008) to efficiently broadcast messages without resorting to flooding. To broadcast, each message will be tagged with a particular bound on some form of site identification; the recipient will forward the message to sites among the fingers that fall within the bound and each forwarded message will be tagged with an appropriate partition of this bound. Thus, a requirement for efficient bounded broadcast is a way to enforce *total order* on the participating sites; for example, bounded broadcast can be done easily on a Chord overlay with bounds on the site IDs because its site IDs constitute a *factor ring*.

Figure 2.3 shows an example of a bounded broadcast on a fully-assigned Chord overlay with 16 sites starting from site 0. Site 0 has four fingers:

site 1, site 2, site 4 and site 8. These identity numbers will form bounds covering $\mathbb{Z}_{16}$: $[1, 2)$, $[2, 4)$, $[4, 8)$ and $[8, 0)$ respectively. Thus, site 0 will forward the message to all its fingers with their respective bounds. Each of these fingers, upon receiving the message together with the bounds, will forward the message to its own fingers under that bound. For example, site 4 will receive the bound $[4, 8)$ and it will forward the message to site 5 with bound $[5, 6)$ and site 6 with bound $[6, 8)$; it will not forward to site 8 and site 12, which are its fingers, because $8 \notin [4, 8)$ and $12 \notin [4, 8)$. Due to the total order enforced on the identities, the message is guaranteed to reach all sites and each site will receive the message only once (i.e., optimal broadcast); this is the case even for imperfect Chord assignment.

2.2 MapReduce Frameworks

When discussing the original MapReduce framework, it is important to note that the framework is an encapsulation of both a parallel programming model and a system architecture equipped with a complete distributed file system (e.g., Ghemawat et al., 2003; Mundkur et al., 2011). It is generally because of the prowess of these two components in handling large data sets in a massively parallel manner that boosted the popularity of the framework and the emergence of the hybrid MapReduce frameworks.

2.2.1 MapReduce Programming Model

As previously mentioned in Section 1.1.4, the programming model arises as an effort to generalize special-purpose computations at Google that process large amount of raw data. In their own words, the authors said that the

generalization is inspired by the map and reduce primitives in Lisp; they claim that most of their computations involve mapping each data element to a set of intermediate key/value pairs and then reducing the values according to the key, or formally:

$$map :: (k_1, v_1) \rightarrow [(k_2, v_2)] \tag{2.4}$$

$$reduce :: (k_2, [v_2]) \rightarrow [v_3] \tag{2.5}$$

Actually, it should not come as a surprise that a lot of computations on a (very long) list of data elements can be expressed as a map action followed by a reduce action. It is largely unknown if this model was a formal development or an empirical conception but such a model is the reminiscence of catamorphism on list; recall that any function (i.e., homomorphism) on a list may be uniquely represented with a map (i.e., list functor) followed by a fold (i.e., catamorphism).

Adopting the MapReduce programming model requires the programmer to obey, at the minimum, the type signatures of the map and reduce functions. Conceptually, the map function is applied on each data element and a list of intermediate key/value pairs is generated accordingly. The MapReduce framework will handle the grouping of the intermediate key/value pairs according to the key. And finally, the reduce function is applied to each list of values of a particular key and the final result is written to disk. Notice that the programming model does not require the programmer to explicitly control the parallelism of the processing; this is accomplished by the framework. The class of operations expressible with the MapReduce programming model can be said to be embarrassingly parallel due to the fact that it is possible to distribute the work load such that minimal inter-site communication is required during the bulk of the operation. Thus,

Gustafson's law (Gustafson, 1988) predicts that such a framework will be massively scalable and it has been demonstrated to be so.

Empirically, the MapReduce programming model is rather expressive in implementing operations on large data sets. Works have been done to adopt this model as the query language despite being criticized as being too low level (Pavlo et al., 2009). For example, SQL/MapReduce (Friedman et al., 2009) adopts the MapReduce programming model to express user-defined functions in a SQL database management system (DBMS). Other works have been done to create a higher-level language on top of the MapReduce programming model to achieve better expressiveness for end-user API; some examples include Pig Latin (Olston et al., 2008), Sawzall (Pike et al., 2005) and HiveQL (Thusoo et al., 2009).

However, due to the specificity of the model, it is difficult to reason formally about the MapReduce programming model. Notice that the map and reduce functions are in no way similar to their functional counterparts, other than that both have the spirit of generic programming on lists of data elements. It has been demonstrated that the map and reduce functions can be implemented using the map and reduce primitives of a functional programming language (Lämmel, 2007) in a similarly parallel manner; this suggests that the MapReduce programming model may very well be a specification of catamorphism.

2.2.2 MapReduce System Architecture

The system architecture of the MapReduce framework usually works in conjunction with some distributed file system (DFS); in its original design, the DFS used is naturally the Google File System (GFS) (Ghemawat et al.,

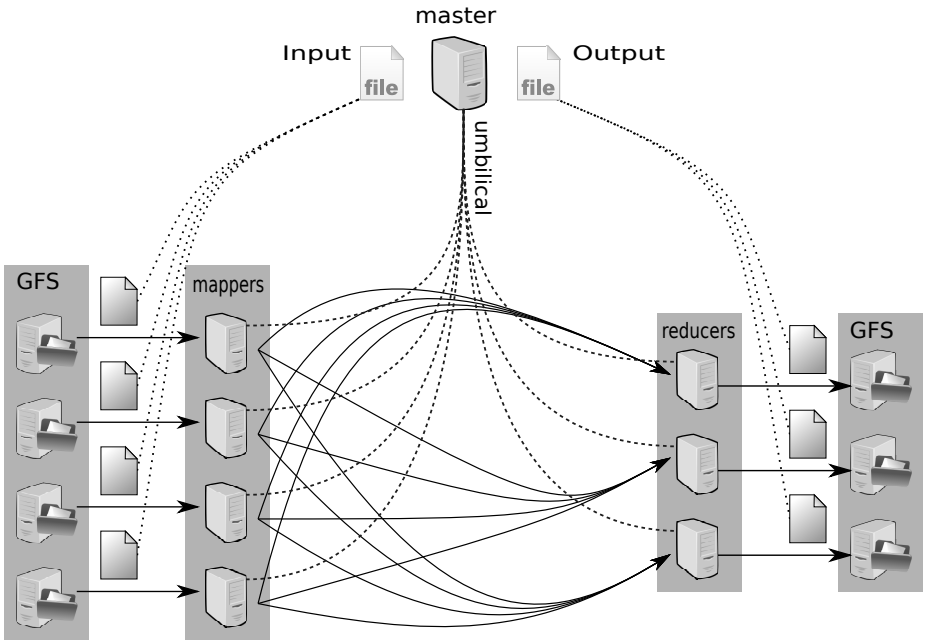


Figure 2.4: MapReduce system architecture

2003). Even though both systems are independent from one another, they are intended to be implemented over the same large cluster of commodity hardware; though this is not compulsory (e.g., Heintz et al., 2013). Under each system, a processing site will be identified as the master site while the others are worker sites (refer to Figure 2.4); this means that, in actuality, the system architecture of a MapReduce framework consists of *two* overlapping master/workers architectures: one for the DFS and another for the processing of MapReduce jobs.

For many DFSs used for MapReduce systems, the abstract concept of a "file" consists actually of a collection of physical files (typically about 64 megabytes) under the underlying file system of the operating system; the term *sub-file* shall be used to refer to these underlying physical files. The worker sites, under the DFS, will hold these sub-files possibly under some replication scheme. On the other hand, the master site will manage all the meta-data (e.g., physical location, filename and pathname) corresponding

to the sub-files. The master site will also maintain the consistency of the sub-files and manage the state of the worker sites.

The processing architecture of a MapReduce framework also consists of an identified coordinating master site with a large collections of worker sites; the master site will keep track of the progress and validity of the worker sites and the tasks in their charges. Upon accepting a MapReduce job, the master site coordinates the entire operation. The input file is first split up into chunks; for most DFSs (e.g., GFS), the optimal split is the same as the sub-file size such that each input sub-file corresponds to a map task. Then the master identifies the idle worker sites and assigns them map tasks (i.e., mappers) and reduce tasks (i.e., reducers) in accordance to the processing phase. The mappers will begin by retrieving the appropriate chunks; it will be optimal if the chunk happens to be local to the mappers. The mappers proceed on applying the map function on each key/value pair and produce the intermediate key/value pairs; typically, these intermediate pairs have to be sorted locally after the map task. The completed mappers will inform the master site about its completion upon which an identified reducer will retrieve a portion of the intermediate pairs corresponding to a particular key. The number of reducers is a user-defined parameter of the job. When a reducer has all the required portions, it will apply the reduce function on each key/values block and the result is written directly to file system. Note that the retrieval phase by the reducer (i.e., shuffle phase) is usually the single most expensive phase out of the entire MapReduce job, which is to be expected since this is the phase when the bulk of the inter-sites communication is done.

As noted in Section 1.2.2, deploying a web-scale system such as the MapReduce framework over a large cluster of machines does provoke the chal-

lenge of augmented aggregated MTBF. Furthermore, even if MTBF is not a problem, given that MapReduce jobs are often relatively long-running, it is extremely inefficient to restart the processing upon failure. This is the reason why the design of the MapReduce processing defines a job as a direct summation of relatively idempotent tasks. In this way, whenever a task is detected to have failed, that task can be restarted without affecting the progress of the other in-progress tasks; recall that the map task depends on only the corresponding sub-file while the reduce task depends on all the map tasks and none of the other reduce tasks. Such idempotent nature also helps with tasks that take abnormally long time to process (i.e., *stragglers*) by running a *backup* task in parallel so that the first task among the two to complete will be chosen to commit. On a side note, it is interesting to highlight that such fine granularity processing of idempotent tasks as a mean to complete a job is only possible because it is explicit in the MapReduce processing model.

2.2.3 Resource Managers

With deployments over increasingly massive cluster, developers begin to feel the need to re-examine the internals of the MapReduce framework as the single master design necessarily presents eventual limitations (e.g., single point of failure and communication bottlenecks) on both the storage engine and the execution architecture. With regards to the storage engine (e.g., HDFS, which is an open-sourced clone of GFS for Hadoop), high availability is induced through the maintenance of a backup master site (i.e., `NameNode`) on hot standby such that fast fail-over may be done in times of failure (Myers, 2012). In handling the execution aspect of MapReduce,

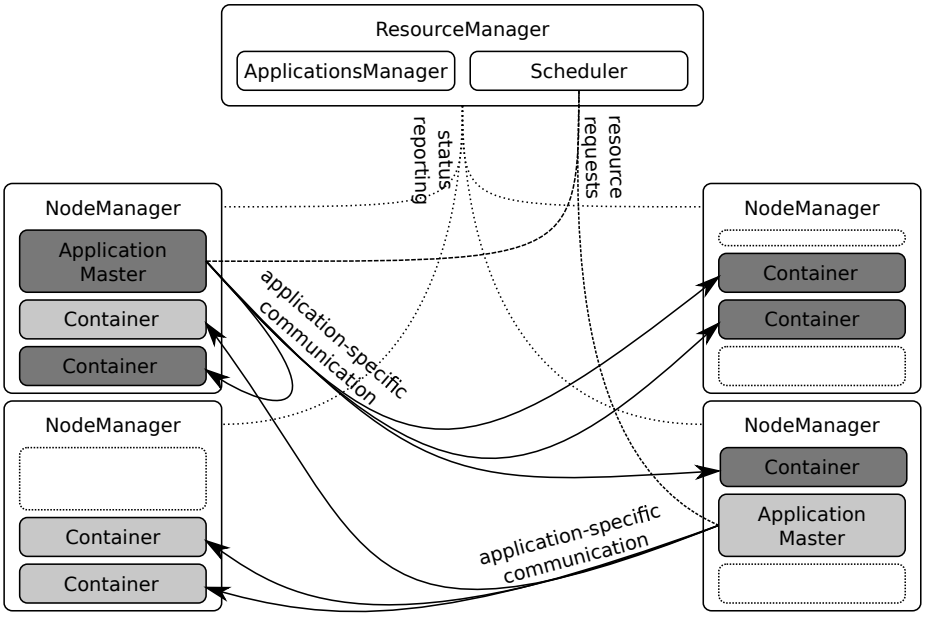


Figure 2.5: YARN architecture

there are several independent developments of resource management frameworks for MapReduce deployment (e.g., Ching et al., 2012; Hindman et al., 2011) The purpose of these frameworks is to isolate resource management from job coordinations by providing multiple "virtual" master/workers clusters such that singly failures of the master sites (i.e., `JobTracker`) will not be propagated to the entire cluster and that each cluster manages its own job coordinations.

YARN (Apache, 2012) is the result of an overhaul effort in version 0.23 by the Apache Hadoop development team to address the deficiencies of the Apache Hadoop MapReduce framework with regards to scalability and performance given the observed trends towards larger deployed clusters. The following presents a high-level description of this architecture as an overview of the new execution environment; the discussion assumes the nomenclature typical of the Hadoop MapReduce framework.

The fundamental idea of YARN is to split up the role of the traditional MapReduce `JobTracker` site into two separate entities: the `ResourceManager`

and the per-application (i.e., job) `ApplicationMaster`.

On deployment, one processing site in the cluster will be assigned as the `ResourceManager` while the others will each be running a `NodeManager` daemon. Figure 2.5 depicts the architecture of a cluster with one `ResourceManager` and four `NodeManager`s. Each `NodeManager` manages the collection of resources (e.g., memory) on that particular site where the resources are encapsulated in units of `Container` based on the amount (e.g., memory size) requested by the application through the `ResourceManager`. The `ResourceManager` consists of two main components: an `ApplicationsMaster` managing the application submissions and a customizable `Scheduler` handling the allocations of the `Container`s.

When an application (e.g., a MapReduce job) is submitted to the `ResourceManager`, it registers with the `ApplicationsMaster` and one `Container` is instantiated in one of the `NodeManager`s to run the application-defined `ApplicationMaster`. The `ApplicationMaster` will then negotiate with the `Scheduler` for the collection of `Container`s required to execute the application in question. The `Scheduler`, at the discretion of its implementation, will instantiate and allocate the `Container`s in the `NodeManager`s and communicate the allocations to the `ApplicationMaster`. The `ApplicationMaster` then performs its computations, whatever it may be, on these allocated `Container`s. The `ApplicationMaster` is free to release the `Container`s or request for more `Container`s throughout its execution lifetime.

Under YARN, a MapReduce job is just one type of the applications submittable to a Hadoop cluster; the developers envisioned other processing frameworks to be deployed within a Hadoop cluster.

Upon submission, the MapReduce `ApplicationMaster` will request for at least one `Container` for each map task. As the `ApplicationMaster` receives the `Container` allocations, it will launch a mapper process in each `Container` for each map tasks with preference on the data locality. The mapper process will first fetch the data from the file system as indicated in the definition of the map task and then proceed to apply the map function on each key/value pair. After each mapper process completes its work, it will sort and partition the intermediate output according to the pre-defined number of reducers and store them locally. The `ApplicationMaster` will have to keep track of the locations of all these partitions.

Once all the mappers are launched, the `ApplicationMaster` will make requests for the `Container`s required by the reduce tasks. In each of the then-allocated `Container`s, the `ApplicationMaster` will launch a reducer process. The reducer process will first fetch the appropriate partitions of the intermediate map output as located by the `ApplicationMaster` (i.e., the shuffle phase) and then merge them to build the input for the reduce function. Once the reduce input is ready, the reducer process will apply the reduce function on each key/values chunk of this intermediate input and produce the final output. The completion of all the reducer processes will complete the MapReduce job.

2.3 Summary

Structured P2P overlays are well-studied system architectures that have led to much developments in both the research and the industrial circles. Most structured P2P overlays function as a DHT in operation, in which case, the principle of consistent hashing applies most of the time. By following the

principle of consistent hashing, these DHTs are guaranteed systemic qualities on their functioning as distributed indices. Being one of the pioneer implementations of structured P2P overlays, Chord stands the test of time[1] arguably because of its algebraic simplicity and implementation robustness; the beauty of Chord as an overlay lies with its association with group theory such that formal theories can be formed on the practical system. Among all the structured P2P overlays, BATON presents as a special class on its own because it distributes data elements efficiently through range partitioning rather than hashing, which allows range queries to be executed. The logical topology formed by the fingers of a structured P2P overlay permits algorithms to be implemented logically; one of such algorithms is the bounded broadcast whereby messages are broadcast with identification of the fingers and without resorting to flooding.

The MapReduce framework is made up of a specific programming model executed over a distributed system. The MapReduce programming model consists of two user-defined primitive functions (i.e., the map and reduce functions) that is able to express a wide range of algorithms due to their resemblance to formal generic programming (i.e., catamorphism). The MapReduce system architecture is actually the collaboration between a DFS (e.g., GFS) and the MapReduce execution architecture, both of which are typically based on some master/workers architecture. Through the use of replication, the MapReduce system is able to assure fault-tolerance in both storage and the execution itself. Recent developments in MapReduce introduces the externalization of resource managers such that each job is processed in a "virtual" cluster; the purpose is to provide an additional level of delegation such that scalability may be augmented.

[1] Literally; Chord shares the 2011 ACM SIGCOMM Test of Time Paper Award together with CAN (Ratnasamy et al., 2001).

Chapter 3

Scalability: Katana

3.1 Motivation

As the MapReduce framework gains popularity as a generalized, massively parallel processing framework, developers begin to observe insufficiency in its expressiveness to cater certain operations; a particular example will be the notorious relational join operation. As such, various hybrid versions of the MapReduce framework appear in recent literature. Most of these hybrids involve the inclusion of an additional step other than the original map phase and reduce phase. For example, the Map-Reduce-Merge framework (Yang et al., 2007) adds an additional *merge* phase after the reduce phase. The merge function takes the outputs of two separate MapReduce tasks and produces another output of its own based on the keys of the outputs of the reduce tasks. Yang et al. (2007) demonstrate that traditional relational join algorithms, such as sort-merge join, hash join and nested-loop join, can be expressed in this hybrid Map-Reduce-Merge framework. On the other hand, the Map-Join-Reduce framework (Jiang et al., 2011)

includes an optional *join* phase after the map phase. The join function takes the outputs of several map functions and performs equi-join based on the keys of the outputs of the map tasks. For the Map-Join-Reduce framework, although the join is restricted to equi-joins, multi-way joins are allowed. Other works that seek to expand or incorporate the MapReduce programming model includes SQL/MapReduce (Friedman et al., 2009) and Nephele/PACTs (Battré et al., 2010). These hybrid frameworks are rather efficient in augmenting the expressiveness of the original MapReduce programming model[1]; however, they may not be addressing the core of the problem with regards to the lack of expressiveness.

Furthermore, as previously mentioned in Section 1.3, it is said that the MapReduce framework uses a master/workers architecture for the sake of simplicity; while the MapReduce framework has been demonstrated to scale up to thousands of processing sites (Monash, 2011), the single master design necessarily presents eventual limitations (e.g., single point of failure and communication bottleneck). In fact, works have been committed precisely in these regards. Resource management frameworks (e.g., YARN (Murthy et al., 2011), Mesos (Hindman et al., 2011) and Corona (Ching et al., 2012)) are developed to isolate the resource management from job coordinations. On the other hand, high availability is induced through the maintenance of a backup `NameNode` on hot standby such that fast failover may be done in times of failure (Myers, 2012). Instead of building up on the pre-existing architecture, the problem is approached by adopting an alternative architecture: structured P2P overlays. In Section 1.3, it has already been suggested that structured P2P overlays are worthy alternative architectures to consider since structured P2P overlays are known to be more scalable than the

[1]Note that these works are orthogonal to those that seek to achieve expressiveness by building *on top of* the MapReduce programming model (e.g., Olston et al., 2008; Pike et al., 2005; Thusoo et al., 2009).

master/worker architecture; they also resolve the issues with single point of failure and communication bottleneck.

In addition, the MapReduce programming model may not be apt for processing under a P2P environment, thus there is a need to devise a different model for the framework. Revisiting the origin of MapReduce, recall that the MapReduce programming model is inspired by the *map* and the *reduce* functions of functional programming languages. Both functions operate on the list data type, which corresponds to the list-based input and output under MapReduce. Now, the combination of the two functions is a specification of a generic programming mechanism known as catamorphism (Meijer et al., 1991), which operates on arbitrary algebraic data types (i.e., lists and trees). Given that many of the structured P2P overlays can be abstracted as a Cayley graph (Lupu et al., 2008), this becomes the bridge to connect P2P overlays and generalized data processing.

This chapter presents a novel P2P-based generalized processing framework: the *Katana* framework. It can be deployed on many of the currently known structured P2P overlays. The following are some of the contributions of the Katana framework:

- The framework provides a programming model in which processing logic may be implicitly distributed with universality and expressiveness, much like the MapReduce framework.
- The Katana programming model encompasses that of MapReduce; any MapReduce algorithm can be emulated by the Katana programming model without going into the details of the definition.
- The Katana framework is deployable over a broad class of structured P2P overlays, including Chord and CAN.

- On top of the enhancements to the systemic qualities, experimental results indicate that the expressiveness of the programming model also accounts for much better running times.

3.2 Programming Model

The programming model can be distinguished into a data model and a processing model. The discussion in this section is conceptual; the logical correspondence between the programming model and the structured P2P overlays (i.e., the model realization) will be covered in Section 3.3. Note that the data model and the processing model, together with the model realization, are symbiotic in nature but for the presentation, they have to be discussed sequentially.

3.2.1 Data Model

The Katana framework adopts a key/value data model with possible duplicated keys to represent the data elements; in other words, each key corresponds to a *list* of values. As such, a *data set* can be defined as follows:

Definition 3.2.1: Data Set

Under the Katana programming model, a data set is defined to be a distinguishable collection of data elements with the same type signatures for the keys and values. Given a data set Δ with key type K and value type V, it may be defined as a total function: $\Delta :: K \rightarrow [V]$.

3.2.1.1 Conceptual Structure

The processing of many frameworks assumes the data models to be list-based. For example, relational model considers a relation to be a list of tuples and MapReduce assumes a list of key/value pairs as input and output. Such conceptual representation does not necessarily indicate the physical storage; a relation may be stored actually as a B+-tree while the data under the MapReduce framework is distributed into chunks.

The Katana data model adopts a conceptual graph structure. Similar to its list-based counterparts, the conceptual graph does not necessarily indicate the manner in which data elements are stored though they are definitely related. Astute readers may notice that this alludes to Cayley graphs and ultimately to structured P2P overlays. However, for now, readers may consider the theoretical analogy whereby each key corresponds to an unique processing site; as such, the graph structures discussed in the following are derived directly from the Cayley graph. The actual realization is slightly more involved and will be elaborated in Section 3.3.

The introduction of a conceptual graph structure beseech the re-examination of a trivial notion in list-based data models: the traversal of data elements. Generally speaking, in order to assure program correctness, list-based data models give the *illusion* of an one-by-one sequential traversal of the entire conceptual list. For example, relational operations seem to iterate the required relation completely, disregarding the possible presence of indices; the MapReduce framework maintains the list-based processing by also distributing the processing logic and having a shuffle phase between the executions of its two signature functions. In the case of the Katana framework, the

conceptual graph structure requires a more sophisticated illusionary traversal of data elements, which is a post-order traversal of a spanning tree. Such a manner of traversal of data elements will be re-visited in the discussion of the processing model.

3.2.1.2 Type Graph and Data Graph

Definition 3.2.2: Type Graph

Given a key type K, there exists an unique directed graph $\mathcal{G}_K = (\mathcal{V}_K, \mathcal{E}_K)$ called the *type graph* whereby the vertices $\mathcal{V}_K$ are all the keys with the type K. For now, it shall be assumed that the edges $\mathcal{E}_K$ are arbitrary but they *strongly connect* the graph. Recall that a strongly connected graph is one whereby there exists a path (i.e., a sequence of connected edges) from each vertex to any other.

Definition 3.2.3: Data Graph

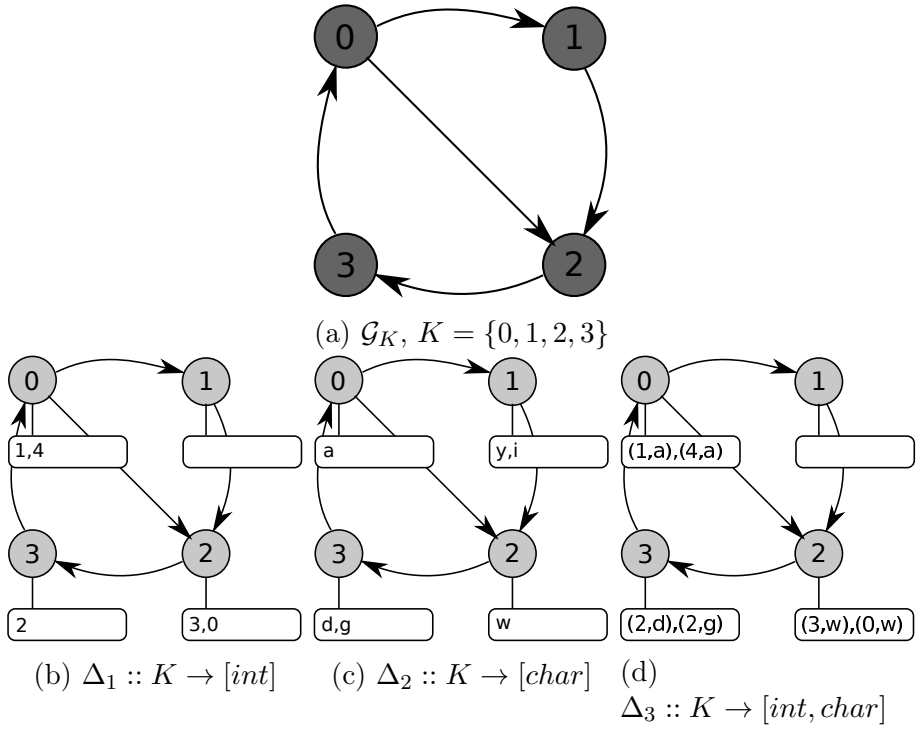
For each data set $\Delta :: K \to [V]$, a graph called *data graph* can be derived such that the vertices and edges are defined as

$$\mathcal{V}_\Delta = \left\{ \big(k, \Delta(k)\big) \,\middle|\, k \in \mathcal{V}_K \right\}$$

and

$$\mathcal{E}_\Delta = \left\{ (v_1, v_2) \,\middle|\, (k_1, k_2) \in \mathcal{E}_K, v_1 = \big(k_1, \Delta(k_1)\big) \wedge v_2 = \big(k_2, \Delta(k_2)\big) \right\}$$

respectively. $\mathcal{G}_\Delta$ can be said to be the *actual* representation of the data set Δ under the type graph of K. Analogically, the type graph represents the overlay while the data graph represents the allocation of



(a) $\mathcal{G}_K$, $K = \{0, 1, 2, 3\}$

(b) $\Delta_1 :: K \to [int]$ (c) $\Delta_2 :: K \to [char]$ (d) $\Delta_3 :: K \to [int, char]$

Figure 3.1: Example of type graph, data graphs and joint data graph

data elements; thus the data graph naturally inherit the edges of the type graph.

Definition 3.2.4: $\odot$ Operator

Now given two data sets with the same key type, $\Delta_1 :: K \to [V_1]$ and $\Delta_2 :: K \to [V_2]$, the $\odot$ operator is defined such that $(\Delta_1 \odot \Delta_2) :: K \to \big[(V_1, V_2)\big]$. The result of the $\odot$ operator is the canonical join of the two data sets, meaning $\forall k \in K$, $(\Delta_1 \odot \Delta_2)(k) = \Delta_1(k) \times \Delta_2(k)$. The resulting joint data set will be represented by the data graph $\mathcal{G}_{(\Delta_1 \odot \Delta_2)} = \big(\mathcal{V}_{(\Delta_1 \odot \Delta_2)}, \mathcal{E}_{(\Delta_1 \odot \Delta_2)}\big)$. The execution of the $\odot$ operator is immediate since data graphs with the same key type has the same type graph (i.e., data co-location).

Figure 3.1 shows a type graph $\mathcal{G}_K$ with $K = \{0, 1, 2, 3\}$ having two example

instantiations (i.e., data graphs) for $\Delta_1 :: K \to [int]$ and $\Delta_2 :: K \to [char]$ and the two instantiations have a $\odot$-product (i.e., $\Delta_3 = \Delta_1 \odot \Delta_2$).

3.2.2 Processing Model

Instead of the signature map and reduce functions of the MapReduce model, the Katana framework introduces a different couple of functions: ana and kata[2].

$$ana :: \big(K_1, [V_1]\big) \to \big[(K_2, V_2)\big] \tag{3.1}$$

$$kata :: \Big(\big(K_3, [V_3]\big), \big[V_4\big]\Big) \to [V_4] \tag{3.2}$$

Contrary to the MapReduce programming model, the ana and kata functions are independent from one another and are not required to be executed one after another. Just like MapReduce, the definition of an ana function or a kata function implies a series of applications of the function in a determined manner over the data elements (i.e., a *job*). Similar to that of MapReduce, the Katana processing model intends to hide the parallelism mechanism from the users. Conceptually, users can consider the data elements to be processed sequentially by a single processing site. As previously mentioned, the manner of access will be a post-order traversal of an arbitrary spanning tree of the corresponding data graph.

The ana function is applied to each vertex (i.e., key K_1 with its list of values $[V_1]$) of the input data graph as traversed using the spanning tree. Each application of the ana function may produce a list of key/value pairs (i.e., of type $[(K_2, V_2)]$). The ensemble of these lists will be recombined

[2]The name *Katana* comes from the concatenation of the two Greek words, κατά (katá) and ἀνά (aná), which symbolize *cata*morphism and *ana*morphism respectively; please don't ask why it is not called *Anakata*.

and grouped according to the key; the result will then be used to produce another data graph based on the type graph (i.e., $\mathcal{G}_{K_2}$) of the output key. The execution of an ANA job does not require a specific traversal of the spanning tree; this specificity is meant for the execution of the KATA job, which is reflected in the *recursive*-like call of the KATA function on the tree.

With the spanning tree, the KATA function is applied on each vertex *together* with the output values from the application of the function on its child vertices as the second input variable (i.e., of type $[V_4]$). Hence this explains the post-order traversal since the applications on the child vertices must be completed before the application on the vertex can execute. The leaf vertices will have empty lists as input for the second variable. The application of the KATA function on the root of the spanning tree will produce the final result. This idea of recursion comes from the original definition of catamorphism. Note that the definition of the KATA job requires neither the specification nor the fixture of the spanning tree; the spanning tree is completely arbitrary.

Generally speaking, KATA jobs are used to perform aggregation of some sort over the data elements, hence the size of the result is envisioned to be relatively small; therefore the result is simply a list of arbitrary values (i.e., of type $[V_4]$) and is not represented under the data model (i.e., no corresponding data graph). On the other hand, ANA jobs are used to build (large) data sets based on currently existing data elements; this means to produce data graphs out of a data graph.

3.2.3 Examples

Some examples shall be discussed at this point to illustrate the usage of the programming model and to demonstrate its prowess in expressing some typical queries.

Algorithm 1 ANA job for Inverted Index

input: $(Index :: K \to [V])$
output: $(InvertedIndex :: V \to [K])$
1: **procedure** ANA(K *source*, $[V]$ *targets*)
2: **for each** *target* **in** *targets* **do**
3: emit (*target*, *source*)
4: **end for**
5: **end procedure**

3.2.3.1 Inverted Index

Given a mapping *Index* from *sources* to *targets* in the form of a data set (hence, data graph), this example seeks to generate the inverse mapping *InvertedIndex* from *targets* to *sources*. This can be done with a single ANA job (refer to Algorithm 1).

After the execution of this ANA job, the sources as "indexed" by their targets will be recombined and grouped according to the targets. Recall that the data model permits the possibility of duplicated keys, thus the result of the ANA job will be the required inverted index in the form of a data set (hence, data graph).

3.2.3.2 *Document-Length*

Given a (text) data set (hence, data graph) with the lines in the document as values under some arbitrary key (e.g., line number), then the number of words can be counted with a single KATA job (refer to Algorithm 2).

Figure 3.2 shows the execution of the KATA job on an example spanning tree. With the post-order traversal, the applications of KATA function will begin at the leaves (i.e., vertices a and b). A leaf application will have empty *others* as input, so it will simply count the number of words of the lines

Algorithm 2 KATA job for *Document-Length*()

input: (*Document* :: $K \rightarrow [String]$)
output: (*DocumentLength* :: $[int]$)

```
1: procedure KATA(K key, [String] lines, [int] others)
2:     int total ← 0
3:     for each line in lines do
4:         total ← total + num_words(line)
5:     end for
6:     for each other in others do
7:         total ← total + other
8:     end for
9:     emit total
10: end procedure
```

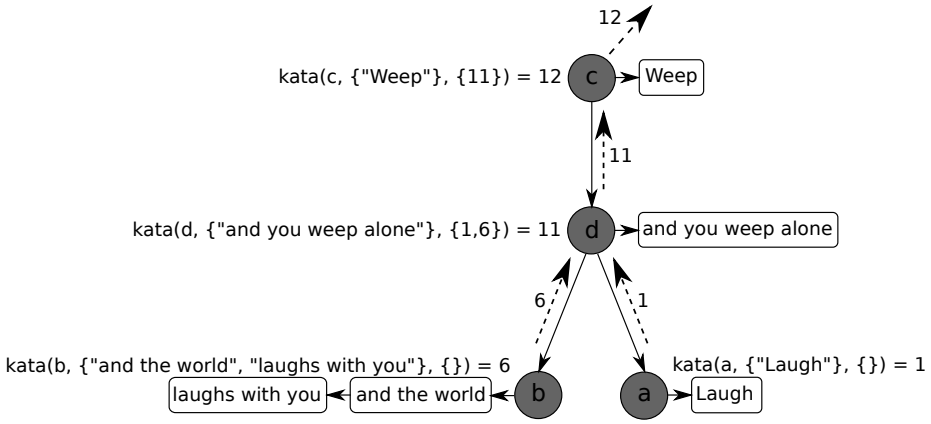


Figure 3.2: Example execution of KATA job for document length

at that vertex. A non-leaf application (i.e., vertex d) will have the emitted values of its child vertices as the *others* parameter; the application on the vertex d will use the emitted values 1 and 6 as *others*. The application on the root vertex (i.e., vertex c) will emit the total count, which is 12 as seen in Figure 3.2.

From the document length example, it can be inferred that many aggregation-like queries (e.g., top-K query and extrema query) can be trivially and efficiently expressed with a KATA job. Note that this may not be the case for MapReduce as similar queries under MapReduce require either explicit mapping to a single reducer or external processing to further aggregate the

Algorithm 3 ANA jobs for MapReduce emulation

input: $(DataSet :: K_1 \rightarrow [V_1])$
output: $(Temp :: K_2 \rightarrow [V_2])$
1: **procedure** $\text{ANA}_{map}(K_1\ key, [V_1]\ values)$
2: **for each** $value$ **in** $values$ **do**
3: emit $map(key, value)$
4: **end for**
5: **end procedure**

input: $(Temp :: K_2 \rightarrow [V_2])$
output: $(Result :: K_3 \rightarrow [V_3])$
1: **procedure** $\text{ANA}_{reduce}(K_2\ key, [V_2]\ values)$
2: emit $reduce(key, values)$
3: **end procedure**

result, both of which can be rather inefficient.

3.2.3.3 Generic MapReduce Algorithms

In fact, the Katana programming model is able to emulate any MapReduce algorithm without going into the details of the definition. Given

$$map :: (K_1, V_1) \rightarrow \big[(K_2, V_2)\big] \tag{3.3}$$

$$reduce :: \big(K_2, [V_2]\big) \rightarrow \big[(K_3, V_3)\big] \tag{3.4}$$

the MapReduce algorithm can be emulated on a data set (hence, data graph) with the executions of the following two ANA jobs (refer to Algorithm 3).

This shows that the Katana programming model is at least as expressive as MapReduce. However, such direct translation may not guarantee the most efficient solution; for example, instead of a single ANA job execution to generate the inverted index, the emulated approach will require two. Similarly, this also highlights the edge that the Katana programming model has over the MapReduce programming model; just as seen for examples on

Algorithm 4 ANA job for *Equi-Join*

input: $\big((\Delta_1 :: K \to [V_1]) \odot (\Delta_2 :: K \to [V_2])\big)$
output: $\big(Result :: K \to [(V_1, V_2)]\big)$
1: **procedure** ANA(K *key*, $[(V_1, V_2)]$ *values*)
2: **for each** *value* **in** *values* **do**
3: emit (*key*, *value*)
4: **end for**
5: **end procedure**

inversed index and word count, instead of two steps (i.e., map followed by reduce), often only one (i.e., either KATA or ANA) is required to express the same query.

3.2.3.4 *Equi-Join*

The Katana programming model can express equi-join naturally. This is an example whereby the Katana framework is able to outperform the MapReduce framework due to its data model. Suppose there are two data sets with the same key type (i.e., $\Delta_1 :: K \to [V_1]$ and $\Delta_2 :: K \to [V_2]$), an equi-join can be performed with the following ANA job with the $\odot$ operator (refer to Algorithm 4).

For a foreign-key join, it can be accomplished with an additional ANA job to "re-hash" the foreign data set according to the foreign key before performing the $\odot$ operator; in essence, this will be similar to performing a distributed hash-join. Notice that unlike approaches in implementing equi-join on the MapReduce framework, the expression by the Katana programming model does not require any extra-algorithmic constructs (e.g., artificial tagging of data elements for reduce-side join (Blanas et al., 2010)) in order to express the join.

3.2.4 Expressiveness of Programming Model

As mentioned in Section 2.2.1, the expressiveness of programming models such as MapReduce can be quite vague to specify. For one, we understand that the primitive functions used in MapReduce (i.e., the map and reduce functions) are not actually equivalent to their counterparts in functional programming languages (Lämmel, 2007). Furthermore, one could trivially implement *any* algorithm under MapReduce by having all the data elements mapped to a single reducer and execute the algorithm in question locally. Therefore, any definition of expressiveness of such programming models should include some notions of efficiency in it in order to be absolutely honest.

Fortunately, for the Katana programming model, there exists a substantial mathematical foundation to provide a glimpse of its expressiveness in this regard. In the following, categorical notions (refer to Appendix B) shall be used to describe the expressiveness of the ANA and KATA jobs.

The spanning tree constructed for the execution of a KATA job is conceptually an algebraic data structure. By construction, this tree can be abstracted with an appropriate polynomial endofunctor F on the category of sets, **Sets** (i.e., F : **Sets** $\rightarrow$ **Sets**), such that for an object X in **Sets**, $F(X) = 1 + K \times X$ where K is the coproduct of k terminal objects and k is the maximum number of leaves of a node in the spanning tree; note that this is a canonical depiction of a tree algebraic data type (Bird and Meertens, 1998). Defining the endofunctor naturally induces the definition of the category of F-algebras (and dually, of F-coalgebras); note that the category of F-algebras (F-coalgebras) on **Sets** has an initial (terminal) object (refer to

Proposition B.10). Thus, defining the KATA function is equivalent to defining the F-algebra, $\varphi : F(X) \to X$, in the following commutative diagram where the isomorphism $in : F(\mu F) \to \mu F$ is the initial object:

$$\begin{array}{ccc} F(\mu F) & \underset{in^{-1}}{\overset{in}{\rightleftarrows}} & \mu F \\ {\scriptstyle F(\langle\!|\varphi|\!\rangle)}\downarrow & & \downarrow{\scriptstyle \langle\!|\varphi|\!\rangle} \\ F(X) & \xrightarrow{\varphi} & X \end{array} \tag{3.5}$$

As such, the execution of the KATA job is exactly the evaluation of a catamorphism (i.e., $\langle\!|\varphi|\!\rangle = \varphi \circ F(\langle\!|\varphi|\!\rangle) \circ in^{-1}$) on the induced category. Therefore, a KATA job is able to express all recursive functions on **Sets** (i.e., catamorphism) based on F much like generic programming of functional programming languages (i.e., the *fold* operator) (refer to Remark B.23) since $in : F(\mu F) \to \mu F$ is the initial object in the category of F-algebras (Hutton, 1999); recall that an initial object is an unique object with unique arrows to all other objects. This recursion execution is evident in example execution depicted in Figure 3.2. Furthermore, if the branches of the tree are clustered according to locality, as seen later in Section 3.3, the execution of a KATA job can essentially achieve data parallelism across the different branches.

On the other hand, despite the origin of its name, an ANA job is not quite directly related to an anamorphism on the induced category of F-coalgebras (i.e., $[\!|\vartheta|\!] = out^{-1} \circ F([\!|\vartheta|\!]) \circ \vartheta$ where $out : \vartheta F \to F(\vartheta F)$ is the terminal object). Though staying true to the significance, ANA serves the purpose of data sets construction: an ANA job indeed creates a new data graph from a pre-existing data graph. By analogy of category theory, the result is somewhat similar to that of anamorphism in that it transforms a set into some "structured set". However, the execution is unlike that of anamorphism where the set is transformed directly into an algebraic structure (i.e.,

expressible with a polynomial endofunctor). Instead, it is much more useful, and certainly more efficient, if the set is transformed to the "intermediate" data graph such that the tree structure is implicit since the endofunctor F is only (implicitly) defined upon the execution of the KATA job.

Therefore, the dual functions of KATA and ANA are meant to be able to express analytic jobs on data sets such that the ANA jobs are used to construct increasingly rich information as preparation, if required, while the KATA jobs are used to aggregate these informations.

3.3 Model Realization

This section shall elaborate the *raison d'être* of the particularities mentioned in Section 3.2. In essence, the data model is conceptualized for the sake of being a bridge between the processing model and structured P2P overlays.

A trivial approach to realize the programming model is through the theoretical analogy that considers the abstracted Cayley graph of a structured P2P overlay as the type graph since Cayley graphs are strongly connected by nature. A data graph will then represent the actual allocation of the data elements in the overlay. This will effectively distribute the execution of an ANA job or a KATA job across the processing sites. However, such an approach is infeasible because it will require a rather oxymoron definition of an *isomorphic* hash function together with an overlay size that is as big as the key space so that each key is allocated to a distinct processing site. Therefore, a different approach is required.

3.3.1 Cayley-Metric Distributed Hash Table

Definition 3.3.1

Let $\mathbb{T}$ be the set of all data types. Given $t \in \mathbb{T}$, $\mathcal{D}_t$ is defined to be the set of data elements of the type t.

Definition 3.3.2

A *hash function* on type $t \in \mathbb{T}$, h_t, is a function mapping from $\mathcal{D}_t$ to a set of *hash buckets*, $\mathcal{B}$.

Given a common set of hash buckets $\mathcal{B}$, an indexed family $\{h_t : \mathcal{D}_t \to \mathcal{B}\}_{t \in \mathbb{T}}$ of hash functions can be identified. A DHT can be said to be built on top of an identified $\{h_t\}_{t \in \mathbb{T}}$, which is usually based on some cryptographic hash (e.g., SHA).

For many DHTs, distribution is achieved by first assigning an unique identity value from $\mathcal{B}$ to each participating site and data elements are assigned to the site where its bucket is the "nearest" to the identity value.

Definition 3.3.3: Identity Assignment

Given a site $s \in \mathcal{S}_0$, where $\mathcal{S}_0$ is the set of *all* possible processing sites, its association with its assigned identity value defines an *injective* function $id : \mathcal{S}_0 \mapsto \mathcal{B}$. $id(\cdot)$ is usually implemented with a cryptographic hash[a] on the IP address of the processing site.

[a] A cryptographic hash function being injective in this case is a common assumption in practice.

Definition 3.3.4: Metric DHT

A *Metric DHT* (M-DHT) is a DHT such that there exists a *metric function* $d : \mathcal{B} \times \mathcal{B} \mapsto \mathbb{R}^+$ such that for the set $\mathcal{S} \subseteq \mathcal{S}_0$ of participating sites, the bucket $b \in \mathcal{B}$ is allocated to the processing site $s \in \mathcal{S}$ if and only if $\forall s' \in \mathcal{S}$, $d\big(b, id(s)\big) \leq d\big(b, id(s')\big)$.

Definition 3.3.5: Allocation Functions

The allocation mechanism of an M-DHT defines an indexed family of allocation functions $\{alloc_{\mathcal{S}} : \mathcal{B} \mapsto \mathcal{S}\}_{\mathcal{S} \subseteq \mathcal{S}_0}$ where given $\mathcal{S} \subseteq \mathcal{S}_0$ and $b \in \mathcal{B}$, $alloc_{\mathcal{S}}(b)$ is the site allocated with the bucket b. Note that $alloc_{\mathcal{S}}(\cdot)$ is *surjective* since $\forall s \in \mathcal{S} \subseteq \mathcal{S}_0$, $alloc_{\mathcal{S}}\big(id(s)\big) = s$; there is at least one bucket allocated to any site.

The definition of hash buckets $\mathcal{B}$ often defines a *integer group* (refer to Definition A.3) (e.g., Chord) or a *direct product* (refer to Definition A.6) of integer groups (e.g., CAN), which is a group with the canonical modular arithmetic as its binary operator. Thus, a Cayley graph of such hash buckets can be identified based on some *generating set* (refer to Definition A.8). For many DHTs, the routing algorithm indirectly employs this kind of Cayley graph to build the routing table such that we can essentially superimpose a Cayley graph on top of the overlay (Lupu et al., 2008).

Definition 3.3.6: Cayley DHT

A *Cayley DHT* (C-DHT) is a DHT whereby given a set $\mathcal{S} \subseteq \mathcal{S}_0$ of participating sites, there exists a Cayley graph $\mathcal{G} = (\mathcal{B}, \mathcal{E})$ such that

$\forall b_1, b_2 \in \mathcal{B}$, if $(b_1, b_2) \in \mathcal{E}$ then either $alloc_{\mathcal{S}}(b_1) = alloc_{\mathcal{S}}(b_2)$ or there is an entry in the routing table of $alloc_{\mathcal{S}}(b_1)$ to $alloc_{\mathcal{S}}(b_2)$.

For most C-DHTs, the routing table has "redundant" routes as part of the design, usually for the sake of routing efficiency or structural robustness. This is reflected in the fact that the generating set in question is not *minimal* (i.e., the smallest generating set). Therefore, we can also distinguish the generating set into the disjoint union of a minimal generating set and an *auxiliary* set; the auxiliary set corresponds to the "redundant" routes. We refer to the Cayley graph generated by the minimal generating set as the minimal Cayley graph.

Definition 3.3.7: Cayley-Metric DHT

A Cayley-Metric DHT (CM-DHT) is meant to be both an M-DHT and a C-DHT. In addition, the metric function used in a CM-DHT has to be a *word metric*[a] based on the minimal Cayley graph.

[a] A *word metric* measures the length of the shortest path in the Cayley graph between two elements.

The requirement of using word metric as the underlying metric function ensures that the corresponding Cayley graph also encapsulates the notion of locality, which, as seen in Section 3.5, will aid the efficiency of the implementation. For the sake of brevity, it shall be claimed without proof that some examples of CM-DHT include CAN, Chord, Viceroy (Malkhi et al., 2002), and Cycloid (Shen et al., 2006); though not explicitly alluding to CM-DHT, Lupu et al. (2008) demonstrates how one may proceed in proving them.

3.3.2 Realization

Proposition 3.3.1

The definition of $\{h_t\}_{t\in\mathbb{T}}$ of a CM-DHT realizes the type graph while the definition of its $\{alloc_{\mathcal{S}}\}_{\mathcal{S}\subseteq\mathcal{S}_0}$ realizes the data graph.

Proof. With a CM-DHT, the type graph of a particular type $t \in \mathbb{T}$ can be constructed by distinguishing it into two levels: the global level graph and the local level graph. The global level graph is exactly the P2P overlay. It is strongly connected because firstly, a Cayley graph is strongly connected and secondly, $\forall b_1, b_2 \in \mathcal{B}$, if there is a path from b_1 to b_2 in the Cayley graph, then there will be a path from $alloc_{\mathcal{S}}(b_1)$ to $alloc_{\mathcal{S}}(b_2)$ in the overlay since $alloc_{\mathcal{S}}(\cdot)$ is surjective. The vertices of a local level graph are the keys assigned to a particular site; suppose $\mathcal{V}_s$ is the set of vertices of the local graph at site $s \in \mathcal{S}$, then $\mathcal{V}_s = \{d \mid d \in \mathcal{D}_t, alloc_{\mathcal{S}}(h_t(d)) = s\}$. The local level graph can be considered to be arbitrarily strongly connected since this depends on the local data structure, which will be covered in Section 3.5.1.

As a result, the entire overlay gives a global level graph of local level graphs. This graph of graphs can be "collapsed" by assigning an edge between two arbitrary vertices of different local level graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ for each edge that connects the site of $\mathcal{G}_1$ and the site of $\mathcal{G}_2$ in the global level graph. Notice that the resultant collapsed graph is necessarily strongly connected. This collapsed graph is a type graph because the definition of $\{h_t\}_{t\in\mathbb{T}}$ assures that the assignment of the edges to the vertices only rely on the key type (i.e., hash values). The allocation of

data elements (i.e., key/value pairs) according to the hashing of keys will, thus, induce the data graph since hashing of keys of the same type will preserve the groupings. □

Note that due to the nature of M-DHT, we are given the implicit assurance that data elements with the same key will be allocated to the same site; as we shall see in Section 3.5.1, this will facilitate the implementation of the $\odot$ join operator. In addition, the number of vertices in a local level graph (i.e., data elements assigned to a processing site) is expected to be significantly larger than the number of vertices in the global level graph (i.e., the number of sites), data parallelism is effectively induced across the processing sites for the execution of the ANA job or the KATA job. The dual levels of graphs also provide the flexibility of optimization of the actual processing both at a local level within a processing site and at a global level across the overlays.

3.3.3 Emulation of MapReduce Combine Phase

The original design of the MapReduce framework incorporates a possible optimization via an intermediate combine phase. If permitted, a definition of the combiner function allows intermediate output of the map function to be "pre-reduced" *in situ* before the shuffling phase. This will potentially reduce the amount of data shipping required. Such a mechanism could have been built into the Katana framework; alternatively, an emulation can be adopted.

In Katana, there is a system-defined key called the *site key*, which allows the local level graph to be collapsed while preserving the global level graph. In most implementation, the site key is simply the identity value of the site. Thus by defining an ANA function that maps to the site key, no data

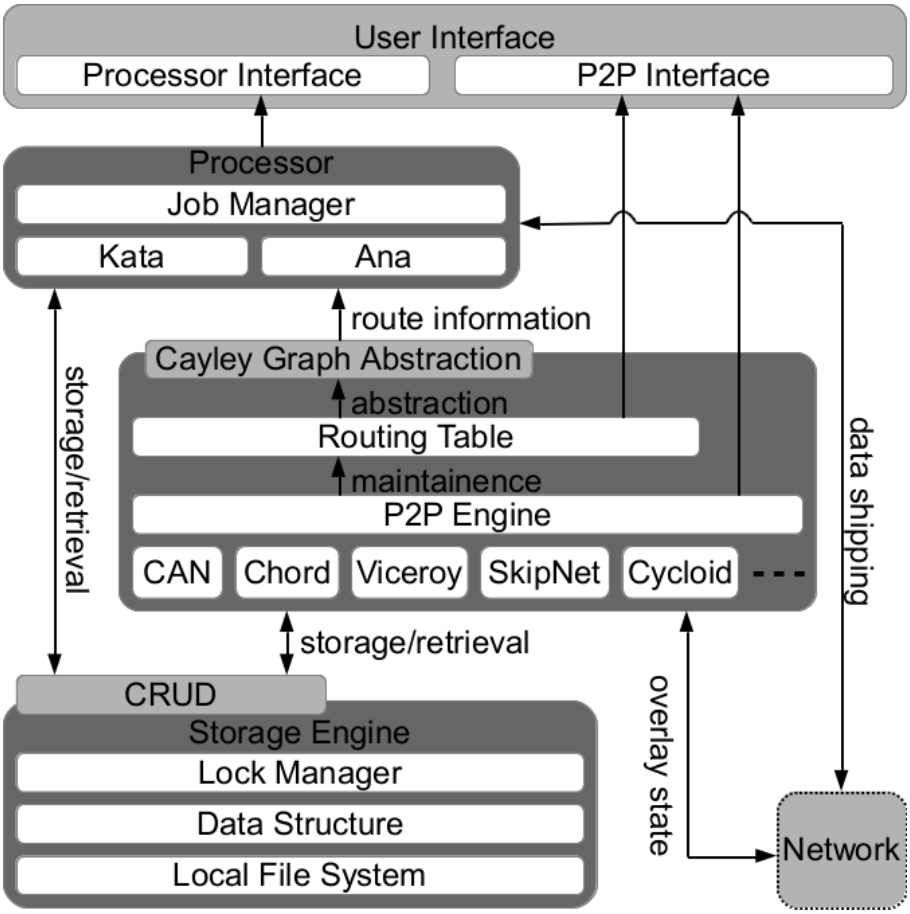


Figure 3.3: System architecture of a processing site in the Katana framework

shipping will be done (i.e., M-DHT). Then, an intermediate ANA function can be defined to "combine" the values. Essentially, the combine phase is just an additional (ANA) job; as such the use of combiner functions have not been considered in the experiments so as not to compound on the fundamental comparisons of the frameworks.

3.4 System Architecture

The decentralization of the implementation is reflected in the system architecture (refer to Figure 3.3). It is maintained that Katana is that of a *framework*, meaning that it serves as a foundation in which generalized data

processing can be done in parallel under a decentralized architecture. The architecture depicted is not the absolute extend; other related work may be incorporated to augment its operative qualities.

3.4.1 Storage Engine

The storage engine is oblivious of the network; its sole purpose is to store the data elements in the form of key/value pairs in an appropriate manner so as to facilitate the processing. The data structures, of which the exact implementation will be detailed in Section 3.5.1, manifest as flat files in the storage engine; each data subset assigned to the processing site is stored as a single file identified by the data set name. The lock manager is responsible for the location and locking of the files on the file system. The lock manager exposes the usual CRUD interface to the processor and the P2P engine. Much of the operations is expected to be done in bulk, therefore the CRUD interface has file level granularity. A finer granularity of operation is not forbidden but at this point, it is deemed that such coarser granularity suffice well.

3.4.2 Peer-to-Peer Engine

The P2P engine together with the storage engine will form the canonical structured P2P overlay. The exposed interface allows the data owner to load and retrieve data elements as per normal. An additional requirement on the P2P engine is that it has to expose the routing table as an abstraction of a Cayley graph to the processor; this is for the executions of the ANA and KATA jobs. Notice that the P2P engine component presents as a variable in the system. This is possible because the programming model only requires the

structured P2P overlay to be a CM-DHT (refer to Section 3.3); therefore, there will be a myriad of candidates that can be considered for the P2P engine of the Katana framework. The choice of the P2P engine has been kept open because each type of overlay has its suitability depending on the context.

3.4.3 Processor

The processor component is the main novelty of the Katana framework; it handles the execution of the ANA and KATA jobs, of which the exact mechanism will be detailed in Section 3.5.3 and Section 3.5.4 respectively. Recall that the data model is realized into two levels of strongly connected graphs (refer to Section 3.3); this is reflected exactly in the relationships the processor has with the storage engine and the P2P engine. At the local level, the processor accesses the storage engine for the local data elements required for the job. And at the global level, the processor has to access the network for data shipping.

Within the processor, the job manager handles the coordination and the flow control of the data shipping. In addition, due to possible concurrent executions, the isolation and the queuing of the jobs is also done by job manager. The exposed interface allows user to submit job request; this involves the user writing the required job definition. In the implementation, an ANA job will require the user to indicate input data set names[3] as well as the output data set name on top of the ANA function definition. On the other hand, a KATA job will require only the input data set names with the KATA function definition, the resulting data set will be shipped to the user upon completion.

[3]Using multiple data set names implies the use of $\odot$ join on all the input data sets.

3.5 System Internals

The definition of the Katana framework is rather liberal so as to provide the flexibility of adopting different solutions to suit the applicative needs. However, there are aspects of the framework that provoke the interest for detailed discussion.

3.5.1 Data Storage

Recall that as per normal functioning of most structured P2P overlays, when the data elements are loaded into the system, they are implicitly partitioned and distributed across the processing sites in an entirely decentralized manner; this is a systemic advantage of adopting structured P2P approach over a master/worker architecture. It is the intent of the Katana framework to exploit this implicit quality.

Up until now, it has been asserted that the local data structure manifests as a complete graph of data elements so that when combined with the global complete graph of the overlay, the required data model is realized. However, implementing and maintaining a persistent graph structure can be rather costly and inefficient in practice. Furthermore, as seen later, the logical existence of the complete graph is actually not required for the execution of the ANA job and the KATA job.

In actuality, the local data structure is a list structure. The data elements in the form of key/value pairs are grouped first according to the bucket then to the key. For most implementation, this can be done with an external sort operation with an appropriate comparator on the key type (i.e., comparing

on the hash value before the key). Such arrangement facilitates the re-distribution of data elements for the execution of the ANA job. For the KATA job, recall that there is a need to identify an arbitrary spanning tree; given that a list is also a spanning tree, this does not deviate from the data model.

Even with the logical manifestation of the list structure, the conceptual complete graph of data elements is still relevant. This is because a conceptual graph model provides the flexibility to "rearrange" the data elements without corrupting the execution. This, as seen later, allows many opportunities for runtime optimization.

Both the ANA function and KATA function require input of local data elements in *chunks* of values of the same key (refer to Section 3.2.2). This can be done easily by loading the key/value pairs into a temporary buffer until the next key before submitting to the ANA or KATA function. Furthermore, since the key/value pairs are already sorted according to the keys, the $\odot$ join operator will not pose additional problem; each of the participating data sets only requires one pass to create the joint chunks similar to that of a relational sorted join operation.

The data set chunks can be considered as basic unit of work for the ANA function and KATA function; the term *data chunk* is hereby used to refer to the in-memory list of values of the same key. In fact, opening a file handler to read a data set effectively gives an *iterator* of data chunks to the processor during the execution of a job.

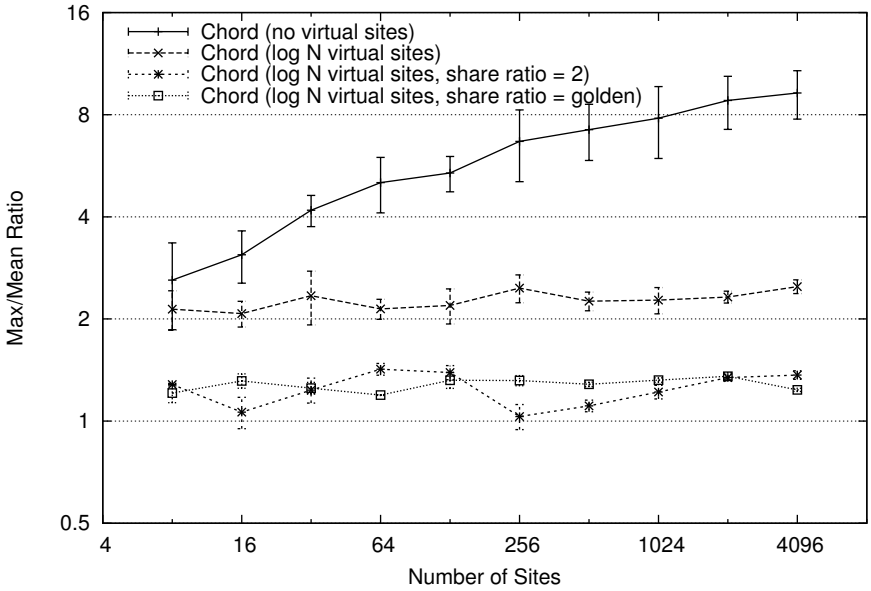


Figure 3.4: Max/Mean ratios of different Chord schemes under simulation

3.5.2 Load Balancing

The performance of a distributed processing framework is noticeably tied to the uniformity of the distribution of data elements. Therefore, the *Achillies' heel* in using structured P2P overlays for data processing is the inherent load imbalance; this is because the use of hashing in structured P2P overlays cannot guarantee uniform distribution of data elements across the sites. As such, the design of DHTs based on consistent hashing (refer to Section 2.1.1) advocates the employment of *virtual* sites to alleviate the load imbalance.

With the use of virtual sites, each *physical* processing site emulates several logical (i.e., virtual) site instances, each with a different identity value coupled with a complete routing table. In this way, the summation of the load of each virtual site at a physical one will tend to be better balanced. In Figure 3.4, it can be seen that a simulated Chord overlay with $\log_2 N$ virtual sites has rather constant load imbalance while Chord without the

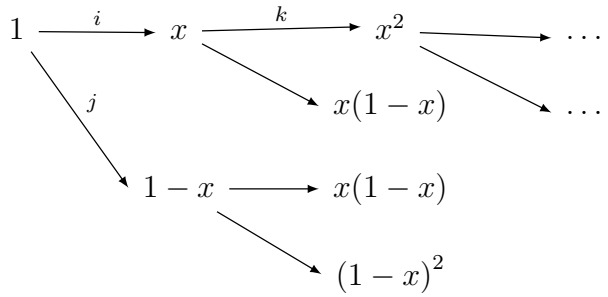
use of virtual sites experiences degrading load imbalance as the cluster size increases.

Note that in Figure 3.4, the metric *max/mean ratio* is used as a mechanism to measure load imbalance. This value is derived simply from the quotient of the load of the most loaded site over the average load; a max/mean ratio of 1 will indicate perfect distribution (i.e., the most loaded site is as loaded as the average). Roughly speaking, suppose the processing task is linear to the load, a max/mean ratio of 2 will indicate that the processing time taken will be twice of that if the load was to be perfectly distributed; this is because the total time taken is often determined by the slowest site. Despite the fact that early version of the Katana framework performs rather well with such an implementation (Goh and Tan, 2013), such a revelation bodes ill for the prospect of using structured P2P overlay to support a processing framework. This is because judging from the simulated max/mean ratio of vanilla Chord (i.e., slightly more than 2), the performance achieved will always be less than half of the theoretical capability of the cluster. Much can be improved in this aspect.

The linchpin for the improvement of load balancing lies in the observation that most structured P2P overlays are designed for a much harsher environment whereby site participations are very dynamic. In comparison, even though distributed processing frameworks (e.g., MapReduce) generally consider site failures as a regular phenomenon, a deployed cluster still has a relatively benign operating environment as compared to the ones structured P2P overlays typically handle.

Under Katana, the joining procedure of Chord has been re-designed such that each new (virtual) site will actively seek out the most loaded site to

share the load by setting its identity value between that site and its predecessor. Note that such a mechanism will probably not work well under the dynamic P2P environment but given the relative permanence of a cluster, it can be argued that the compromise on the robustness for a much better load balancing is a fair deal. With regards to sharing of load, the newly joined site can simply halve the load of the most loaded site by setting its identity value in the midpoint of the range governed by that site; however, such a sharing mechanism leads to rather inconsistent, albeit occasionally optimal, load balancing as seen in Figure 3.4 (i.e., share ratio = 2). To determine an alternate share ratio, consider the following trace of sharing, where each binary branching represents the joining of a new site and the resulting leaves represent the load distribution:



(3.6)

Note that $x \in (0, 1)$ and $\frac{1}{x}$ is the share ratio. Without loss of generality, it can be assumed that $x \geq 1 - x$ (i.e., branch i and j in (3.6)). An additional guard condition can be enforced: $1 - x = x^2$ (i.e., branch j and k in (3.6)). This condition assures that the leaves only differ by at most a factor of x. Solving, one will obtain the solution of $x = \frac{\sqrt{5}-1}{2}$. Incidentally, the share ratio is then $\frac{1}{x} = \frac{1+\sqrt{5}}{2} = 1.618\ldots$, which is the *golden ratio*. Thus, the sharing can also be determined based on the golden ratio such that the load balancing is relatively consistent across the number of sites, as evident in Figure 3.4 (i.e., share ratio = *golden*).

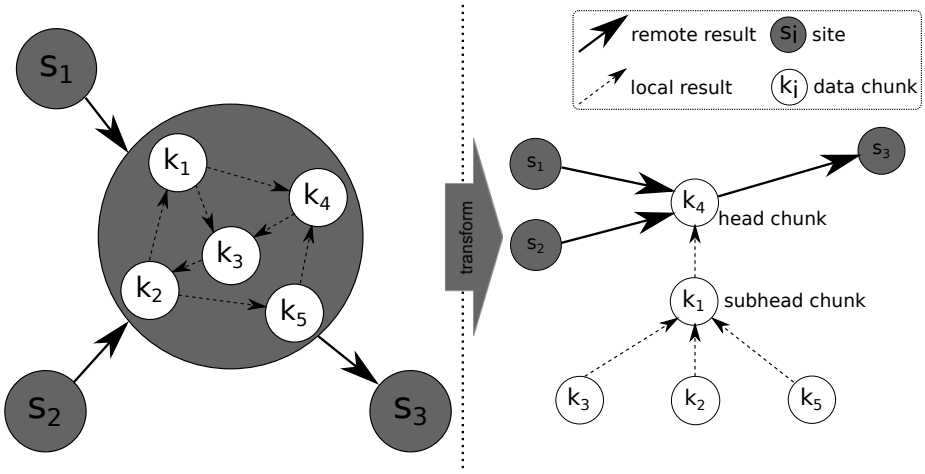


Figure 3.5: Identification of a spanning tree for a KATA job

3.5.3 KATA Job Execution

Recall that a KATA job usually expresses some form of aggregation over the data elements and for that purpose, as discussed in Section 3.2.2, a spanning tree over the data elements (i.e., data graph) is identified. Given that the data graph is distinguished into a global level graph for the processing sites and a local level graph for the local data storage, the required spanning tree is identified in a similar manner.

At the global level of the overlay, bounded broadcast (refer to Section 2.1.4) is used to identify the global level spanning tree for the execution of the KATA job; this form of broadcasting is especially applicable for CM-DHTs, especially with the use of word metric. At the local level, the KATA function is applied on each data chunk with possible emitted values from the applications of the KATA functions on other data chunks; these other data chunks are possibly located on remote site according to the global spanning tree. The local level spanning tree is identified implicitly via the manner of applications of the KATA function on the data chunks.

At the local level, the KATA function is applied on each data chunk with several or no input from applications of the KATA functions on other data chunks; these other data chunks are possibly located on remote site according to the global spanning tree. The local level spanning tree is identified implicitly via the manner of applications of the KATA function on the data chunks. The flexibility of the identification of the spanning tree provides the opportunity to optimize the processing a little; this involves the *transformation* of the spanning tree (refer to Figure 3.5 for an example).

Figure 3.5 demonstrates a manner which is statically optimal for the execution of a KATA job. As each processing site is potentially awaiting the results from other processing sites, it will be optimal to perform as much local processing as possible prior to the wait. Out of all the data chunks, two chunks (i.e., the first two in the stored list) are arbitrarily selected: the *head chunk* (i.e., chunk k_4 in Figure 3.5) and the *subhead chunk* (i.e., chunk k_1 in Figure 3.5). For the other chunks, the KATA functions are applied as if they are leaves of the spanning tree and their results are used for the application of the KATA function on the subhead chunk. The result of subhead chunk will be used in the application of the KATA function on the head chunk. The application of the KATA function on the head chunk will also wait for the remote results (i.e., from site s_1 and site s_2 in Figure 3.5) and it will send its result to the parent site (i.e., site s_3 in Figure 3.5) according to the global spanning tree.

Note that as long as the applications form the required spanning tree as indicated in the programming model, there is no requirement (and should not have any requirement) that the actual spanning tree should follow the approach mentioned above, or any other pre-determined approach. For example, a certain degree of parallel local execution can be explored by

dynamically identifying several sets of head and subhead chunks for the execution. However, as of now, such local optimization scheme has not been implemented in the prototype used in the experiments; the prototype employs the static optimization approach displayed in Figure 3.5.

Recall that the result of a KATA job is not modelled by the data model and it is simply a list of arbitrary values (refer to Section 3.2.2); this is reflected in the fact that the final results will be shipped to the user after the execution of the KATA function on the head chunk at the root processing site as determined by the global spanning tree.

3.5.4 ANA Job Execution

Unlike the KATA job, the ANA job does not require causality in the execution; ideally, the executions of the ANA functions should be initiated simultaneously for all the processing sites. For this purpose, bounded broadcast is employed again to disseminate the ANA job. The spanning tree also serves as the manner in which the job managers may monitor the job progress in a decentralized manner (i.e., sites will monitor their child sites).

Upon receiving an ANA job via the bounded broadcast, the ANA function is applied sequentially on the data chunks and the emitted key/value pairs are stored together in a temporary flat file at the local site. The temporary file will be sorted and re-distributed to other processing sites according to the key with the locality information retrieved from the P2P engine. Therefore, each processing site will be receiving from all the other processing sites some data subsets. Given the concurrency of inter-site shipping of data elements, non-blocking network I/O becomes indispensable in this phase of operation. The processing site will merge the received data subsets until it has merged

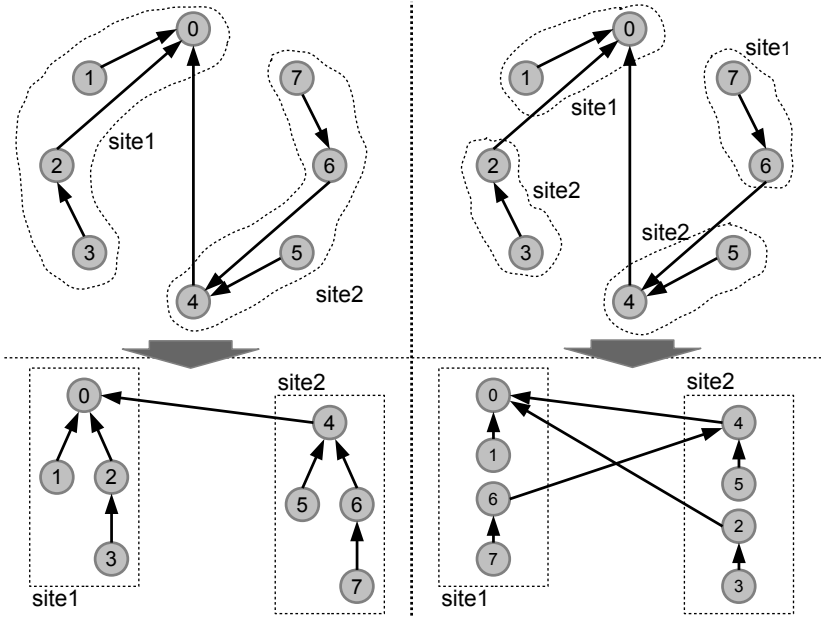


Figure 3.6: Effects of virtual sites on spanning tree of a KATA job

all the required data subsets into one. Since the data subsets are already sorted, therefore the merging process will not pose a problem. Notice that the resulting merged data subsets across the overlay necessarily form the required data graph in accordance to the type graph of its key type.

Note that the sorting and re-distribution of the resulting data set is an expensive operation and should be avoided if they are not necessary. Therefore, when storing the temporary data set, the Katana framework also take note of spillage (i.e., data elements that do not belong to the processing site) and out-of-order elements; sorting will be done if there is at least one out-of-order element and similarly, re-distribution will be done when spillage is detected.

3.5.5 Dual Overlays

While the use of virtual sites optimizes the load distribution of data elements, it disrupts the locality of the data elements. This is because each

physical site emulates a number of virtual sites, which may not be adjacent with one another. As a result, each physical site has to process *several* local spanning trees. Though this phenomenon does not affect the correctness of the processing model, it has negative effect on the execution of a KATA job; in particular, it increases communication cost due to larger global spanning tree built for the KATA job and it reduces degree of parallelism due to shorter local pipelining (refer to Figure 3.6). Note that the execution of an ANA job is not affected much since the spanning tree is only used for job dissemination.

The solution adopted is to deploy two distinct overlays over the same cluster of processing sites; they are named the *command* overlay and the *data* overlay. The command overlay does not utilize virtual sites and is used for the propagation of job packages, hence construction of global spanning tree. On the other hand, the data overlay deploys virtual sites as per normal and is used for the allocation and location of data elements. As such, the spanning tree constructed for a KATA job is built using the global spanning tree of the command overlay together with the chunks identified by the data overlay. On the other hand, for an ANA job, the job dissemination is done with the command overlay while the resulting data set is allocated according to the data overlay. In this way, the Katana framework enjoys the load balancing of the data overlay while having the simpler graph (hence, tree) of the command overlay for job executions.

3.6 Experimental Study

In order to gauge the performance of the Katana framework, a prototype[4] is implemented over the modified Chord (as mentioned in Section 3.5.2)

[4]Source code available at: https://bitbucket.org/xanec/projectkhloe

with Java 1.6 and *extensive* experiments against Apache Hadoop have been conducted. The experiments are conducted on a 65 machines cluster. Each machine is equipped with an Intel Xeon X3430 Quad Core CPU (2.4 Ghz), 8 GB memory, two 450 GB SCSI disks, and 1 Gbps Ethernet interface. The operating system used in each machine is CentOS Linux 5.6. The Java virtual machine installed is Java HotSpot™ 64-Bit Server VM (build 24.45-b08, mixed mode).

The experimental data is obtained via the Transaction Processing Performance Council Benchmark™ H (TPC-H) generator. The primary relations used are the LINEITEM relation and the ORDERS relation. The experiments will run on data sets of different scale factor (i.e., $SF \in \{32, 64, 128\}$); when $SF = 128$, the generator produces about 768 million records of LINEITEM (119 GB) and 192 million records of ORDERS (25 GB).

The performance evaluation consists of three experiments (i.e., *Document-Length*, *Equi-Join* and *Aggregation-Query*) over different cluster sizes (i.e., $N \in \{16, 32, 64\}$). For each experiment, the same or similar algorithm is implemented on different frameworks with different configurations to compare the running times taken on these frameworks to achieve the same result. Experiments have been conducted on the current revised version of the Katana framework (i.e., labelled as *Katana (revised)*) as well as the previous version[5] (Goh and Tan, 2013) (i.e., labelled as *Katana (previous)*). In addition, the MapReduce algorithm has been emulated in the manner as mentioned in Section 3.2.3 on the revised Katana framework (i.e., labelled as *Emulated MapReduce*). For each experiment, there will be a total of 45

[5]The previous version of the Katana framework comprises of the basic execution engine without some of the optimizations mentioned in Section 3.5; in particular, it does not employ the modified Chord (refer to Section 3.5.2) and the dual overlays (refer to Section 3.5.5).

experimental runs (i.e., 5 frameworks × 3 cluster sizes × 3 scale factors); the timing obtained for each experiment run is an average calculated from three distinct executions.

For the MapReduce framework, Apache Hadoop 2.2.0, which is equipped with YARN, is used. Based on experiences with MapReduce on YARN, the MapReduce jobs can be quite sensitive to the number of reducers used depending on the job profile, therefore the same job is executed using different number of reducers (i.e., labelled as *Hadoop (R=1)* and *Hadoop (R=2)*) where the R-value is the multiplier used to determine the number of reducers based on the number of sites (i.e., the N-values). For example, if $R = 2$ and $N = 32$, the number of reducers used is $R \times N = 64$. The choice of these R-values is based on empirical observations that the fastest timings tend to occur with these values. For the experiments on Hadoop, one additional machine is dedicated to perform the role of the master site (i.e., resource manager); for example, if $N = 32$, a total of 33 sites will be used. Most of the configurations on Hadoop are unchanged; the only major modification is the reduction of resources (i.e., memory) available per site from 8 GB to 4 GB as an effort to reduce external interference (i.e., experiments of other researchers using the same cluster); naturally, the Katana framework has been configured to match that of the Hadoop framework (e.g., sort buffer size and I/O buffer size).

3.6.1 *Document-Length*

The *Document-Length* experiment counts the number of words in the LINEITEM relation of the TPC-H data set; value of non-textual field is considered

Algorithm 5 MapReduce job for *Document-Length*

```
1: procedure MAP(K key, String value)
2:     int count ← num_words(value)
3:     emit (hash(key), count)
4: end procedure

1: procedure REDUCE(int hash, [int] counts)
2:     int total ← 0
3:     for each count in counts do
4:         total ← total + count
5:     end for
6:     emit (hash, total)
7: end procedure
```

to be counted as one word. The Katana algorithm used for the *Document-Length* experiment is mentioned in Section 3.2.3; the MapReduce algorithm used is as depicted in Algorithm 5.

As observed in Figure 3.7, it is evident that executions for *Document-Length* on Hadoop are faster with fewer reducers (i.e., $R = 1$); this is largely due to the fact that *Document-Length* has relatively light reduce tasks. In comparison to the faster Hadoop execution, the Katana algorithm on the revised Katana framework reduces the running times by 33.65% to 65.63% depending on the configuration of the experiment. In addition, when comparing to the previous version of the Katana framework, there is an improvement of 16.84% to 35.87%.

Generally, the implemented Katana algorithm is much more efficient than the MapReduce algorithm. This is directly due to the capability of the Katana programming model to express what is required with more precision; as a result, the processing task required on the Katana is lighter (i.e., a single KATA job versus a full MapReduce job) and without unnecessary internal operations (e.g., shuffle phase). This phenomenon is more explicit

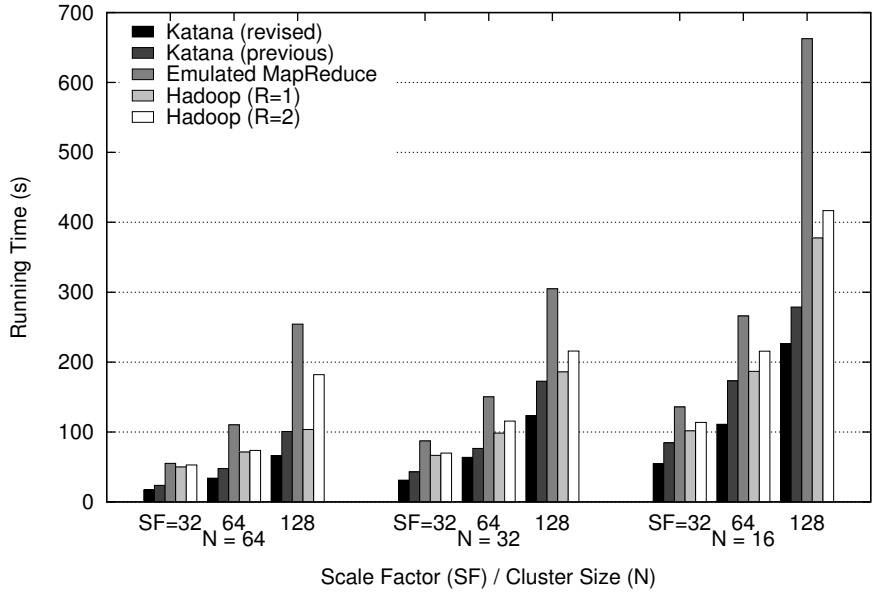


Figure 3.7: Running times of *Document-Length* ($N = cluster\ size$)

particularly when looking at the performance of the emulated MapReduce algorithm on the revised Katana framework.

Though the emulation still enjoys decent scalability, there is significant overhead incurred by the lack of pipelining between the emulated map phase and reduce phase; ultimately, the emulated MapReduce algorithm is still two distinct processing jobs as opposed to a single contiguous processing adopted by most MapReduce systems. Taking this overhead into consideration, the fact that the Katana algorithm is able to outperform the MapReduce algorithm on Hadoop is a clear indication that a better expressed processing such as that of the Katana programming model can improve the performance of generalized processing framework.

The running times on the revised Katana framework has improved quite significantly as compared to the previous version; this can be attributed to the new load balancing mechanism employed. Note that the performance of a KATA job is determined largely by the rate of disk reads because the output data is expected to be small and the internal processing is usually

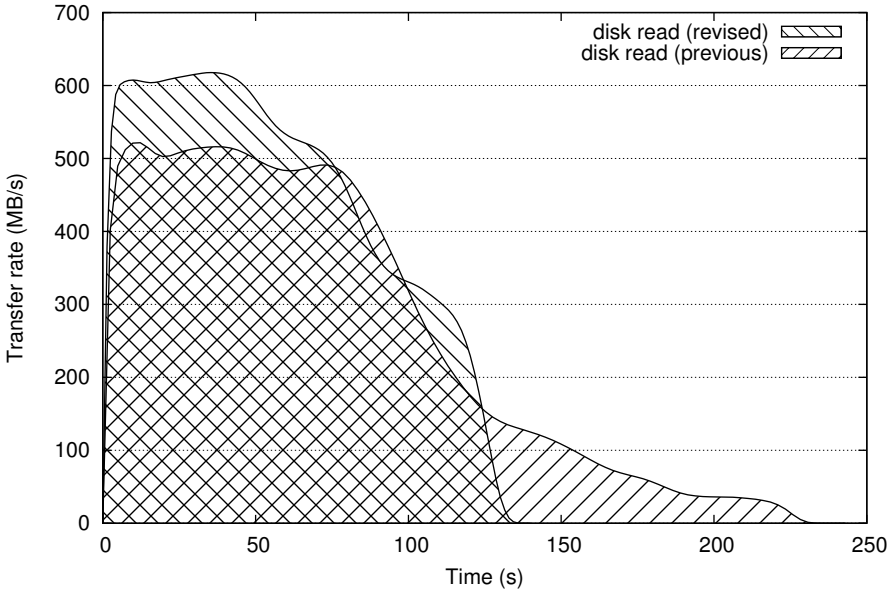


Figure 3.8: Data transfer rate of *Document-Length* ($N = 16$,$SF = 64$)

constrained due to its aggregation-like nature. Therefore, observed that in Figure 3.8, the aggregated throughput of the KATA job on the revised framework has faster tapering as compared to the previous Katana framework. This is an indication that the load is better balanced across the sites such that the overall performance is not hindered by disproportionately loaded sites.

3.6.2 *Equi-Join*

The *Equi-Join* experiment performs a foreign-key equi-join on the LINEITEM and ORDERS relations of the TPC-H data set. The equi-join algorithm used for the MapReduce is that of a reduce-side join (Blanas et al., 2010). The Katana algorithm used for *Equi-Join* experiment is mentioned in Section 3.2.3.

Observe that in contrast to the *Document-Length* experiment, from Figure 3.9, it can be seen that the executions for *Equi-Join* on Hadoop are

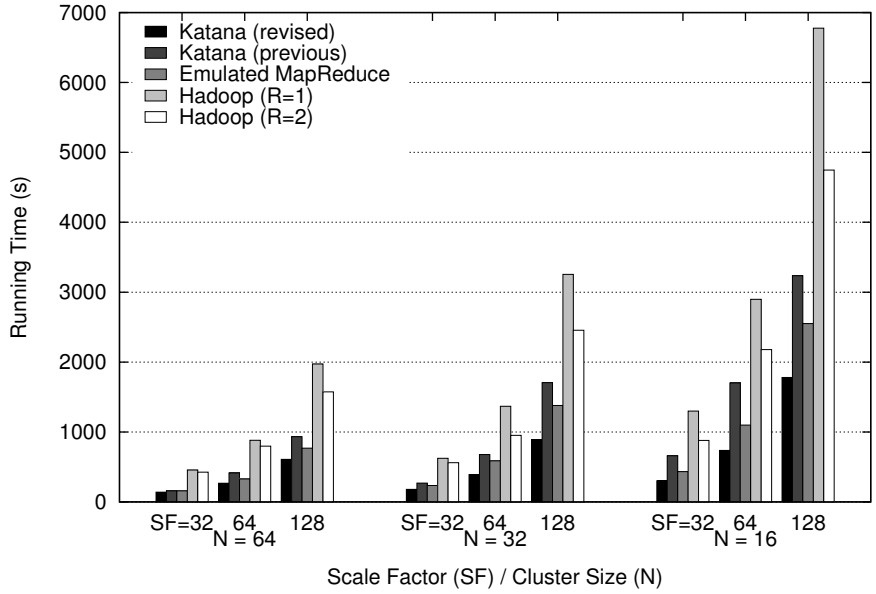


Figure 3.9: Running times of *Equi-Join* ($N = cluster\ size$)

faster with more reducers (i.e., $R = 2$) this time; this is largely due to the fact that *Equi-Join* has significantly heavier reduce tasks. Nevertheless, the Katana algorithm on the revised Katana framework reduces the running times by 59.03% to 68.09%. When compared to the previous version of the Katana framework, the revised framework improves the running times by 12.03% to 56.76%. Moreover, the emulated MapReduce algorithm also outperforms its Hadoop counterparts; in comparison to the faster execution on Hadoop, the emulation on the revised Katana framework reduces the running times by 38.31% to 62.63%.

The phenomenon that the MapReduce emulation for *Equi-Join* on the Katana framework actually outperforms the MapReduce algorithm on Hadoop is consistent with the previous version of the Katana framework though the improvement is much more significant. There are three main reasons for this phenomenon. Firstly, as opposed to the *Document-Length*, the internal processing of reduce-side join is much more significant than the inherent overhead of having two separate jobs. Secondly, due to the nature of an ANA

job, there is actually an effect of push-based data shipping involved in the emulated shuffle phase; as a result, there is less latency in commencing the emulated reduce phase. Lastly, the fact that the reduce phase is emulated by a complete ANA job actually helps the processing in that the emulated reducer task enjoys the use of all the available resources of the site (e.g., memory buffer) as opposed to the design of YARN within Hadoop where the reduce task is mandated to be executed with only a fraction (i.e., half[6] in the optimal configuration) of the available resources.

As compared to the MapReduce algorithm, the Katana algorithm for *Equi-Join* also offers significant improvement; when compared to the emulated MapReduce, the Katana algorithm reduces the running times by 12.15% to 35.51%. This improvement is a result of the capability of the data model. Not only the Katana algorithm does not require artificial tagging, as in reduce-side join, the execution also requires less data shipping. With the use of $\odot$ operator, the Katana implementation only requires one of the relations to the rehashed and re-distributed while reduce-side join requires *both* relations to be re-distributed. The revised Katana framework itself also improves the performance of the Katana algorithm. The rationale is the same as that of the *Document-Length* experiment; the new load balancing mechanism provides much better load distribution, which benefits the processing of both KATA jobs and ANA jobs. In fact, the improved load balancing even results in the emulation on the revised framework to outperform the Katana algorithm on the previous framework.

Figure 3.10 shows the aggregated transfer rates of the Katana algorithm for *Equi-Join*; note the distinction of the profiles of the two executed ANA jobs. In the first job, we observe the overlapping phases of the processing

[6]By default, each reduce task under YARN requires 1 GB of memory.

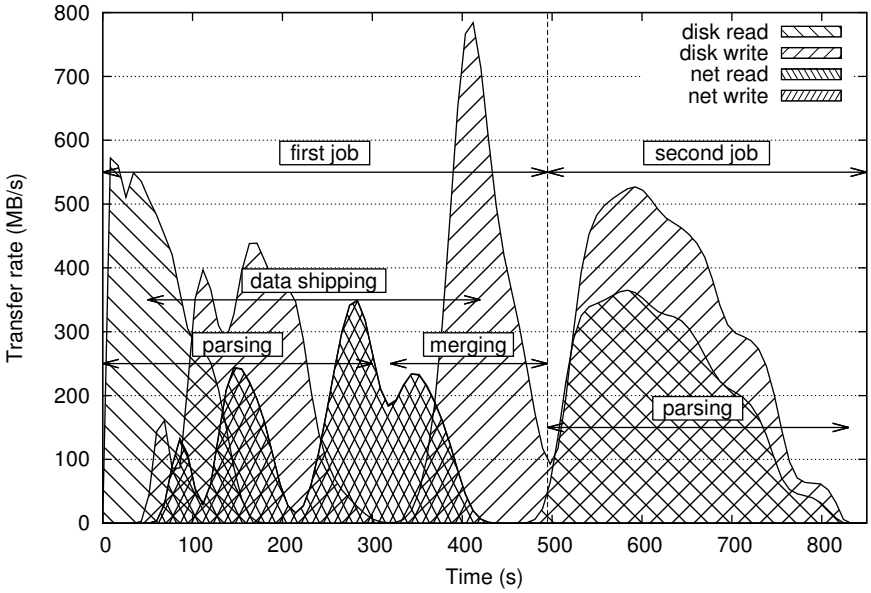


Figure 3.10: Data transfer rate of *Equi-Join* ($N = 16$,$SF = 64$)

of the ANA job; the overlaps come from the fact that the sites push data to the appropriate destination and that the sites will produce output once they received the required splits. Also in the first job, notice that the network read rate actually coincides exactly with the network write rate; this optimality in the latency can be attributed to the use of non-blocking network I/O (i.e., implemented with Java NIO). In the second job, there is actually no data shipping because this particular ANA job produces data elements that are guaranteed to be local to the processing sites.

3.6.3 *Aggregation-Query*

The *Aggregation-Query* experiment performs an aggregation query with the LINEITEM and ORDERS relations of the TPC-H data sets. The query can be expressed in SQL as depicted in Figure 6. The MapReduce algorithm is broken down into the following jobs:

Algorithm 6 SQL statement for *Aggregation-Query*

```
SELECT into temp linenumber, returnflag, linestatus,
                 SUM(totalprice * quantity) AS totalcost,
                 AVG(discount) AS avgdiscount
FROM     lineitem l, orders o
WHERE    l.orderkey = o.orderkey
AND      l.commitdate BETWEEN 1994-01-01 AND 1996-12-31
GROUP BY l.linenumber, l.returnflag, l.linestatus;

SELECT * FROM temp ORDER BY totalcost DESC LIMIT 1;
```

Job 1: The first MapReduce job is the canonical reduce-side join with filtering by *commitdate* during the map phase. The key of the result will be the fields *linenumber*, $returnflag$ and *linestatus* and the value will be the calculated fields *totalcost* ($= totalprice \times quantity$), *avgdiscount* ($= discount$) and *count* ($= 1$).

Job 2: The second MapReduce job will perform the aggregation. The map task is the identity map and the reduce task will sum up the fields of *totalcost*, *avgdiscount* and *count* before dividing *avgdiscount* by *count* to obtain the required average value for that group.

Job 3: The last MapReduce job will select the result with the largest *totalcost*.

Notice that the last job may be replaced with just a simple retrieval and selection; however, as the list of groups may be potentially large, it is not done in this way.

While the Katana algorithm used is also implemented into several jobs, the expression used in the Katana algorithm is more precise. Note that one MapReduce job is roughly equivalent to two ANA jobs. The Katana jobs are as follows:

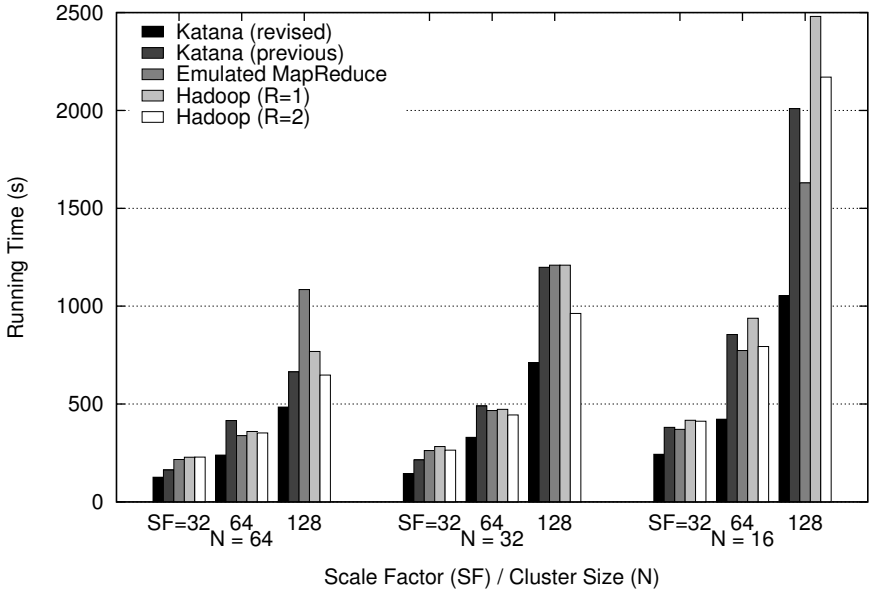


Figure 3.11: Running times of *Aggregation-Query* ($N = cluster\ size$)

Job 1: The first job is an ANA job on LINEITEM to "rehash" them according to the *orderkey* with filtering on *commitdate*.

Job 2: The second job is an ANA job on LINEITEM and ORDERS (i.e., a $\odot$ join); the result of the job the same as that of the Job 2 of the MapReduce algorithm.

Job 3: The last job is a KATA job that performs the required aggregation on the local data of the results from Job 2 and emits the result with the largest *totalcost*.

Since some heavier reduce tasks (i.e., the join operation) are required, it can be observed that, in Figure 3.11, the executions for *Aggregation-Query* on Hadoop are faster with more reducers (i.e., $R = 2$). In comparison to the MapReduce algorithm on the faster configuration of Hadoop, the Katana algorithm on the revised Katana framework reduces the running times by 25.2% to 51.45% depending on the configuration of the experiment. When

compared to the previous version of the Katana framework, the improvement is 12.03% to 56.76%.

As a mixture of job profiles is involved in this experiment, the emulated MapReduce algorithm on the revised Katana framework does not always outperform Hadoop. In general, when the load per site is light (e.g., $SF = 32$, $N = 16$) the emulation falls behind Hadoop; on the other hand, when the load is heavy (e.g., $SF = 128$, $N = 16$), the emulation will outperform. This phenomenon can be explained by the relative significance of the internal processing as compared to the inherent overhead; when the load is lighter, the inherent overhead of the Katana framework when emulating the MapReduce algorithm plays a more significant role.

In general, this experiment coalesces the effects previously described in Section 3.6.1 and Section 3.6.2. As such, the improvement produced by the Katana algorithm on the revised Katana framework can be explained by the reasonings indicated previously (i.e., expressiveness of the programming model and enhancement of the revised framework). The *Aggregation-Query* experiment indicates that the optimization mechanism offered by the Katana programming model and the revised framework transcends multiple and varied jobs.

3.7 Summary

The Katana framework serves as an exploratory work in adopting P2P-mentality in the design of generalized data processing systems; and the findings indeed indicate such a potential. In both its programming model and its system architecture, the Katana framework can be seen as a natural generalization of the MapReduce framework.

The example algorithms have demonstrated that the primitives (i.e., ANA and KATA functions) are actually more expressive than that of the MapReduce model; the fact that the Katana programming model encompasses the MapReduce programming model does say a lot in this regards. Furthermore, the coupling of the data model with the abstract structure of the overlay allows a natural and implicit access to data locality without overburdening the "language" of the processing model; one particular result of such a quality is the possibility of the $\odot$ operator.

Due to the model realization, the execution of the processing model is entirely decentralized, which allows the functioning of the Katana framework to conform to the P2P philosophy. As a result, the qualities of the execution architecture mirrors that of the system architecture. Since the system architecture is based on structured P2P overlays, the derived system inherits many of the systemic qualities. The Katana framework can exhibit better horizontal scalability due the lack of centralized control.

Going beyond the central theme of this chapter, there are reasons to believe that the Katana framework can provide more than just scalability (i.e., elasticity and robustness). Moreover, this work already opens up many research possibilities (e.g., distributed query optimization in the Katana framework) that has not been considered yet. For now, these research avenues shall be left to future works to explore.

Chapter 4

Robustness: Hardened Katana

4.1 Motivation

It has already been mentioned in Section 1.2 that as a distributed system grows in deployment scale, its capability to handle system-wide operations will come under increasing strain from augmented possibility of site failures. As a result, the implementation of a tolerance mechanism to site failures, or *fault-tolerance*, within a modern data processing system has become indispensable. In order to assume fault-tolerant operations, under the context of a modern data processing system, two system-wide architectural states have been identified in this thesis that require explicit assurance on their continual persistence: the *system* state and the *processing* state.

The fault-tolerance of the system state refers to the capability of the distributed architecture and system meta-data to persist in the event of singly site failures. For a master/workers architecture, the implementation of such an assurance is rather straight forward since the system state is usually maintained by the master site in its entirety with the workers announcing

its liveness through periodic updates sent to the master site; in other words, the master site acts as the resource manager whereby the worker sites are treated as deployable resources (refer to Section 2.2.3). Naturally, the challenge in this case lies with the assurance of the persistence of the master site. Therefore, this motivates various research works on the specific fortification of the master site (e.g., Myers, 2012; Ryan, 2012).

Long running processing jobs (e.g., deep data analytics) typically do not have the luxury of restarting upon exceptional failures; it is prohibitively costly and renders the system unusable given the inverse relationship between the cluster size and the mean time between failures (refer to Section 1.2.2). Therefore, the mechanism employed by the MapReduce framework to handle site failures during processing (i.e., fault-tolerance of processing state) gains recognitions. By defining a job to be a collection of relatively idempotent tasks processed across an array of the worker sites, the extent of failure can be kept minimal; the failures of the tasks need not forestall the progress of the entire job. Again, for the master/workers architecture, the maintenance of the processing state is done mostly by the master site; in particular, the master site determines if a task has to be restarted on surviving sites by keeping track of the progress of each task. For the fault-tolerance of processing state at the master site, a master/workers architecture typically resorts to relying on an element of higher availability to aid processing state reconstruction. For example in YARN, the restarting of the `ApplicationMaster` due to failure requires the `ResourceManager` to restore the state of the job progress while the availability of the `ResourceManager` in turn has to rely on some high availability service (e.g., Hunt et al., 2010).

As suggested in Section 1.3, the structured P2P overlay is a viable option for

web-scale system architecture in various aspects. To begin with, structured P2P overlays, by design, already assures the persistence of the system state; in fact, the main prowess of these architectures is their resilience against site failures, which invites the preference of structured P2P architectures over master/workers architectures. This chapter seeks to explore the feasibility of using structured P2P overlays to augment the robustness of a modern data processing system, particularly in the persistence of processing state. To be specific, an extension to the Katana framework is proposed: the *hardened* Katana framework.

The extension is intended to handle the fault-tolerance of the framework together with techniques that are already available to structured P2P overlays, as well as novel inventions specific to the Katana framework. Note that the Katana framework is used as the basis to explore the role of structured P2P overlay in assuring fault-tolerance in web-scale data processing system because the Katana programming model is unique in that it is entirely decentralized. Furthermore, the Katana data model is much akin to the MapReduce-style of job constitution (i.e., definition of a job as a summation of tasks), albeit with a tighter relationships between the "tasks" (i.e., key/values chunks). The following are some of the contributions of the hardened Katana framework:

- A generalized fault-tolerance scheme, called the COVER-CHARGE protocol, is proposed for system-wide operations in a distributed manner; the COVER-CHARGE protocol generalizes the MapReduce-style fault-tolerance.
- The COVER-CHARGE protocol is applicable to system-wide operations in a structured P2P overlay (e.g., bounded broadcast).

- Execution of ANA and KATA jobs can utilize the COVER-CHARGE protocol to render their processing fault-tolerant.

- Experimental studies indicate that the overhead incurred by the COVER-CHARGE protocol for the execution of ANA and KATA jobs is comparable to, if not lesser than, that of similar jobs under Hadoop Apache.

4.2 Model of Fault-Tolerance

It is not known whether the requirement for fault-tolerant processing has influenced the design of the MapReduce processing model or *vice versa* but it is noticeable that the handling of singly site failures by the MapReduce framework during processing is innately tied to the simplicity of its programming model; the expression of the MapReduce programming model translates into the tasks-based definition of a job while inversely, the horizontally-oriented definition of tasks allows for robust execution that constrains the extent of failures. Therefore, the MapReduce framework may not actually have a formal derivation for its provision of fault-tolerance. However, since the advent of the MapReduce framework, several similar generic processing systems have been developed (e.g., Isard et al., 2007; Malewicz et al., 2010); these systems seem to have indirectly adapted the mechanism of robust processing under the MapReduce framework, or what Yang et al. (2010) referred to as *MapReduce-style fault-tolerance*, into their individual execution architecture.

Note that the fact that its fault-tolerance model and its programming model are inherently related can be seen as a powerful revelation for the MapReduce framework, especially since the MapReduce programming model enjoys

recognition for its scope of algorithmic application (refer to Section 2.2.1). This is because the incorporation of the notion of fault-tolerance into its programming model greatly alleviates the considerations involved in the design of the execution architecture; some responsibility of fault-tolerance has been offloaded to the programming model. This is evident in the fact that allowing transparently fault-tolerant processing of generic programs in a large cluster (e.g., a supercomputer) is a heavily invested research area with on-going progress (e.g., Bosilca et al., 2009; Bougeret et al., 2011; Cappello et al., 2009; Elnozahy et al., 2002; Ferreira et al., 2011) but works on MapReduce frameworks often automatically assume fault-tolerance as an innate quality of the framework (e.g., Doulkeridis and Nørvåg, 2014; Lee et al., 2012; Li et al., 2014; Sakr et al., 2013); to be fair, the former considers problems beyond the scale typically assumed even for the latter (i.e., exabyte-scale processing) but the attitudes towards considerations of fault-tolerance hold true.

Given the legitimacy and general acceptance of the MapReduce-style fault-tolerance of processing state, it will perhaps be fruitful to "post-construct" a model of fault-tolerance based on this mechanism and seek to generalize it so as to further the goal of achieving robust processing over structured P2P overlays.

4.2.1 MapReduce-Style Fault-Tolerance

Without resolving the causality of influence, the MapReduce-style fault-tolerance for the maintenance of the processing state can be said to be built on top of several premises on the nature of processing in question:

Divisibility. Each processing job is divided into a (large) number of distinct "*sub*-jobs", or *task*s. Under the context of MapReduce, this is seen in the fact that each MapReduce job is divided into a collection of map tasks and reduce tasks.

Linearity. The completion of the processing job is wholly defined by the singular completion of each individual tasks. Note that a MapReduce job is considered completed if and only if all the map and reduce tasks have been completed.

Replicability. Each task can be repeated at another site, possibly with preference on locality, upon failure. Recall that a map task is defined by the sub-file, which is often replicated to several sites, and a reduce task can be relocated at a different site to restart by retrieving the already-existing intermediate map outputs.

Idempotence. Each task may be repeated without influencing the correctness of the result of the processing job. For most implementations of the MapReduce framework (e.g., Apache Hadoop), each map or reduce task manifests as one or more attempts and the task is considered completed if at least one of the attempts succeeds.

With the assumption on the aliveness of the master site, these premises allow the processing to persist even in the event of singly failures of the worker sites through a rather direct and intuitive mechanism. Given the *divisibility* and *linearity* of processing under MapReduce, the concern for fault-tolerant processing of the entire job is simplified to the assurance of completion of each individual task. Then, with the property of *replicability* and *idempotence*, failed attempts of a task can prompt the relocation of the task to a surviving site to resume the progress of the job without affecting its

correctness. Note that the notion of replicability implies the inheritance of the accessibility and availability of data input (i.e., replicas) from the fault-tolerance of system state. Under such a model, there are no presumptions made on the dependencies between the tasks; in the case of MapReduce, typically, the commencement of a reduce task is tied to the completion of the map tasks. It is assumed that the dependencies, if any, are resolved as part of the nature of the task. For example, under MapReduce, the reduce tasks are entirely oblivious to the global state of the processing; they are independent consumers of work, which is retrieved from the map tasks. Therefore, if the dependencies are not managed well, locking of resources may occur (refer to Figure 5.6 in Section 5.6.1).

4.2.2 Generalization of Notions

Even though the MapReduce-style fault-tolerance is tied to the framework in which it is deployed, based on the previously identified premises of processing, it is possible to extract framework-neutral notions so as to derive an abstract depiction of how persistence of processing state may be enforced under this style of processing. In this section, the derivations of these framework-neutral notions are introduced leading up to the description of the generalization of the said fault-tolerance scheme.

4.2.2.1 Cover

With the properties of divisibility, linearity and replicability, the MapReduce-style fault-tolerance suggests a notion of relatively-independent replicated units of work awaiting to be processed for the fulfillment of the job. A conceptual construct, called *cover*, is introduced to encapsulate this notion.

Definition 4.2.1: Cover

A cover is defined to be an *identifiable* unit of *idempotent* processable work that is *replicated* across the processing system and can be *reliably located* within the system[a].

[a]Note that if the fault-tolerance of the system state is established, the requirement of reliable location of covers is assumed to be satisfied.

With the references to "replication" and "location", it be inferred that the concept of cover is abstracted from the notion of the (local) data input to the processing task; the generalization also encompasses work that does not need data input (e.g., bounded broadcast as seen later in Section 4.3.1) and work that derives input at runtime (e.g., reduce task).

Despite their analogical similarities, the concept of a cover differs actually from that of a processing task (i.e., a map task or a reduce task). A processing task is assigned at runtime to a processing site to execute; in the case of a map task, the assigned site may not hold the required data input locally. On the other hand, a cover already exists conceptually on at least one of the processing sites (i.e., replication) and it is located and selected at runtime to contribute to the processing job; roughly speaking, a cover is always local.

4.2.2.2 Charge and Delegation

A processing job can be redefined as being made up of an union of disjoint covers (i.e., divisibility and linearity) instead of a collection of processing tasks; the covers can be assumed to be pre-identified according to the job definition. Similar to its tasks-based definition, the job is considered to be completed when all the identified covers, regardless of which replica

(i.e., replicability), have been processed. The selection of distinct COVERs to process is the determinant factor in achieving robust processing; for this, the notion of *CHARGE* is introduced.

Definition 4.2.2: Charge

A CHARGE is defined to be a sub-collection of COVER identities that requires processing. During the processing of a job, when a processing site receives a CHARGE, it is responsible for (i.e., in *charge* of) the completion of the processing of the COVERs identified within the CHARGE; in this case, the processing site is called the *site in charge* (of a particular CHARGE). When all the COVERs identified in a CHARGE has been processed, the CHARGE is said to be fully *covered*.

When a site receives a particular CHARGE, it may, or may not, possess the required COVERs as identified in the CHARGE. For the local COVERs that fall within the received CHARGE, the site will proceed to process them; however, for the COVERs that are within the CHARGE but not found locally, the site in charge have to forward a sub-collection of the CHARGE (i.e., a *sub*-CHARGE), consisting of the missing COVER identities, to the appropriate sites. This forwarding is referred to as *delegating*.

Definition 4.2.3: Delegation

When a site is requested to process a missing COVER, it will locate the required COVER via the system and send a sub-CHARGE consisting of the missing COVER to the correct remote site (i.e., delegation). The initial site in charge monitors the progress and aliveness of the said COVER periodically and re-delegates should the processing of the said COVER fails at the remote site. Upon completion, the delegated site will

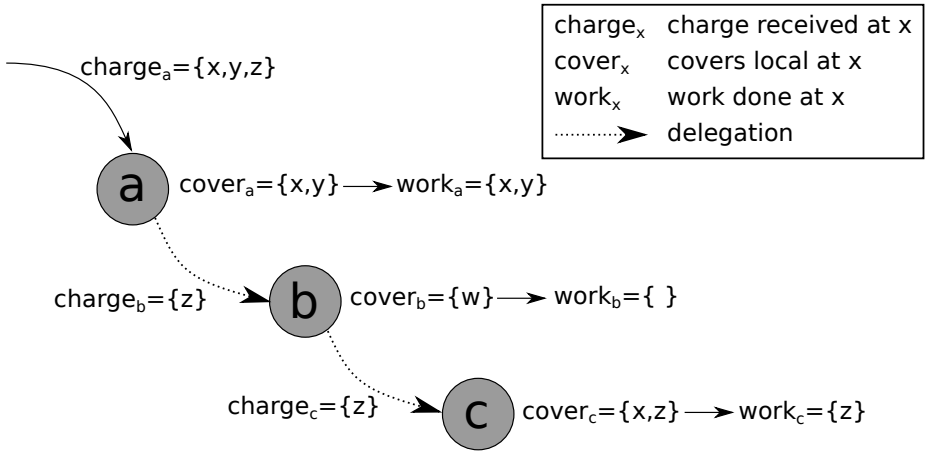


Figure 4.1: Example of COVER, CHARGE and delegation

inform the initial site and the initial site will mark this COVER as being covered under the initial CHARGE. The hierarchy of all the delegations naturally forms a (spanning) tree-shape structure, which is called the *delegation tree*.

Figure 4.1 shows an example of how the elements of COVER, CHARGE and delegation are related in action. The process is initiated when site a receives the CHARGE of $\{x, y, z\}$. Site a will have detected that the COVERs x and y are local while the processing of COVER z will have to be delegated. In the example, site a delegates the CHARGE $\{z\}$ to site b, which will have to further delegate it to site c. Notice that that at site c, only the COVER z is processed even though COVER x is one of the required COVERs; this is because the CHARGE received at site c contains only COVER z.

As depicted in Figure 4.1, the CHARGE can be delegated wrongly (i.e., delegation of CHARGE= $\{z\}$ to site b) and a further delegation is required. While the determination of the target of a delegation is subjected to implementation specificities, with reliable location of COVERs, as supported

by fault-tolerance of system state (i.e., via data replication), it can be assumed that the delegation will *eventually* reach the appropriate site and that "infinite delegation" will be avoided.

4.2.2.3 Cover-Charge Protocol

With the previously mentioned abstractions, the MapReduce-style fault-tolerance can be seen as simply a collection of callback functions based on the events generated regarding the COVERs and CHARGEs (refer to Algorithm 7). These functions are collectively referred to as the *COVER-CHARGE protocol* (CCP). The event-driven programming depiction allows simplistic state management, which augments the robustness of the processing.

The job processing is started by having one of the processing sites (e.g., the master site) receives the initial CHARGE of all the required COVERs. This will trigger the RECEIVECHARGE event. As depicted in Algorithm 7, the ONRECEIVECHARGE function will proceed to process all the local COVERs (Lines 4–9 in Algorithm 7) while delegating the remote COVER identities as sub-CHARGEs to other sites (Lines 11–15 in Algorithm 7); note that the delegation will in turn trigger the RECEIVECHARGE event at the respective remote sites. Should the processing site detect that one of its delegates have failed, it will reprocess the assigned charge under the failed delegates (Lines 21–29 in Algorithm 7); this is possible because idempotence is assumed on the nature of processing (refer to Section 4.2.1). Once the processing site has processed all the COVERs indicated by the assigned CHARGE (i.e., as detected on the WORKCOMPLETION event), it will proceed to notify the requester of the CHARGE (Lines 35–39 in Algorithm 7), triggering the WORKCOMPLETION event at the requester site. This notification is recursive and finally, the site that receives the initial CHARGE will have to be

Algorithm 7 Callback functions of the COVER-CHARGE protocol

```
 1: procedure onReceiveCharge(Site requester, Charge charge)
 2:     register charge under requester for completion notification
 3:
 4:     for each cover in local_covers do
 5:         if cover in charge then
 6:             enqueue cover in work_queue
 7:             remove cover from charge
 8:         end if
 9:     end for
10:
11:     for each cover in charge do
12:         Site delegate ← locate(cover)
13:         delegate.sendCharge(myself, cover.id)
14:         register delegate on cover for monitoring
15:     end for
16: end procedure
17:
18: procedure onDetectSiteFailure(Site failedSite)
19:     Cover[] covers ← covers under failedSite
20:
21:     for each cover in covers do
22:         if cover is local then
23:             enqueue cover in work_queue
24:         else
25:             Site site ← locate(cover)
26:             site.delegate(myself, cover.id)
27:             register site on cover for monitoring
28:         end if
29:     end for
30: end procedure
31:
32: procedure onWorkCompletion(Cover cover)
33:     mark cover as completed
34:
35:     for all the registered charge containing cover do
36:         if charge is fully covered then
37:             notify registered requester of the corresponding charge
38:         end if
39:     end for
40: end procedure
```

notified of the completion of the job (i.e., the initial CHARGE being fully covered by the processed COVERs).

The CCP can be said to assure robust processing because the site in charge will either process the required COVERs or delegate their processing to other sites, and will eventually cover its assigned CHARGE. This assurance is built on the assumption of the reliable location of the replicated COVERs, which is supported by the fault-tolerance of system state. Therefore, the fault-tolerance of system state is a critical ingredient in this protocol. In addition, the failure of the receiver of the initial CHARGE is considered as an exceptional situation that requires implementation-specific corrective measures (e.g., consider the failure of the master site under MapReduce).

4.2.3 MapReduce-Style Fault-Tolerance as CCP

Since the CCP is a generalization of the MapReduce-style fault-tolerance, naturally the operations of the MapReduce framework can be framed analogically under the context of the CCP.

The submission of a MapReduce job is equivalent of assigning the initial CHARGE to the master site. In this case, the initial CHARGE consists of COVER identities where each map or reduce task is represented by one unique cover. Note that as each worker site under MapReduce is capable of executing any of the assigned task, though in the case of map tasks, preferences on data locality are taken into account. Therefore, conceptually, this will mean that each worker site contains all the COVERs while the master site contains none of the COVERs. On a side note, this rightfully indicates that the concept of locality is sometimes lost under MapReduce since local processing is only a preference.

The MapReduce tasks assignment process is modelled by the behaviour that, upon receiving the initial CHARGE, the master site will have detected that none of the required COVERs are local; therefore, it will have to delegate the CHARGEs to the appropriate site. The specialization of the CCP protocol under MapReduce is that all the delegates of the master site will cover the assigned CHARGE because of the nature of the COVER under MapReduce. Upon site failures, the master site will re-delegate the CHARGE to an appropriate worker site, which is equivalent to the reattempting of the task under MapReduce. The rest of the MapReduce processing model (e.g., reading of input, shuffling and tasks assignment scheduling) is considered as implementation specificities as they do not affect the fundamental mechanisms of the CCP.

This thesis asserts that the MapReduce-style fault tolerance is a specialization of the CCP; in other words, the mechanism of the MapReduce-style fault tolerance is exactly the mechanism of the CCP with a few specificities. Similar to the CCP, the robustness of the MapReduce-style fault tolerance is built on top of the fault-tolerance of system state, which is manifested as the fault-tolerance of the HDFS (i.e., the replicated file splits) and of the resource manager (i.e., aliveness of resources). However, the caveat is that the master site will be left vulnerable and thus requires specific fortification and perhaps some higher availability service, as previously mentioned.

4.3 Robust Katana Operations

Given the generality of the CCP, this section seeks to apply the model of fault-tolerance on the operations within the Katana framework. Note that the CCP can be applied to a distributed processing job if and only if the

said processing job can be expressed as a CHARGE of COVERs; in particular, the definition of COVER must be adhered to.

4.3.1 Robust Bounded Broadcast

Since the implementations of most of the system-wide operations under the Katana framework are supported by the bounded broadcast operation (refer to Section 3.5), it will be fruitful to describe how the bounded broadcast can be enforced using CCP.

Note that the current Katana framework utilizes dual overlays (refer to Section 3.5.5). This introduces two types of COVERs under the Katana framework in accordance to the identities derived from the two overlays: the *command* COVERs and the *data* COVERs.

Definition 4.3.1: Command Cover

Given a processing site in the Katana framework, its command COVER is an unit of processable work identified by the local space bounded by its site identity and its *succeeding* neighbours (i.e., those which the site has fingers pointing to) according to the word metric based on the minimal Cayley graph as extracted from the *command overlay*.

For the Chord-variant of Katana, this will be the range $[i, j)$ where i is the identity value of the site in question and j is that of its successor. This local space is called the command COVER identity.

Definition 4.3.2: Data Cover

Given a processing site in the Katana framework, one of its data COVER is a unit of processable work identified by the local space bounded by

one of its virtual site identity and its *preceding* neighbours (i.e., those with fingers pointing to the site) according to the word metric based on the minimal Cayley graph as extracted from the *data overlay*.

For the Chord-variant of Katana, this will be the range $(i, j]$ where j is the virtual identity of the site and i is that of its predecessor. This local space is called the data COVER identity.

By the definition of the CM-DHT, both the command COVERs and the data COVERs are each necessarily non-overlapping and cover the entire overlay. Other than the fact that command COVERs are derived from the command overlay and data COVERs are derived from the data overlay, note that command COVERs are based on the successors while the data COVERs are based on the predecessors. This is because command COVERs represent commands/queries given to the processing sites, hence it follows the forwarding fingers. On the other hand, data COVERs represent processable work, which is defined by the location and allocation of data elements, hence it follows the predecessors. More shall be talked about this in Section 4.3.4.3. Given the nature of virtual sites, the framework will act as a natural surjective mapping from the data COVER identities to the command COVER identities; in other words, the collection of data COVERs is partitioned according to command COVERs.

Proposition 4.3.1

A command COVER can be seen as replicated across the Katana framework and can always be reliably located.

Proof. For the command overlay, when the COVER is lost (due to site failure), its preceding site will implicitly cover the lost COVER due to

Definition 4.3.1. Therefore, logically, any command cover can always be located (i.e., replication and reliable location) as long as at least one processing site exists. □

The nature of a command cover differs from that of a data cover. A command cover concerns with the processing site itself and from a "command" point of view, each processing site is indistinguishable from one another. Therefore, a site can always cover for any failed site in terms of command covers. However, a data cover is tied to the notion of data input, therefore each site is no longer indistinguishable from one another; if a site fails, the data that it holds may be lost if no replica is maintained by the covering site. Therefore, in order to assure the replicability of the data cover, replication of data elements, which is quintessential to all modern data processing systems, must be implemented.

Similar to the implicit covering of lost command covers as described in Proposition 4.3.1, the *succeeding* virtual site will implicitly cover a lost data cover; this is due to Definition 4.3.2. Therefore, it is natural to store each replica of data cover at the succeeding site. In fact, replicas can be stored at N succeeding sites for augmented assurance; the value $N+1$ (i.e., one for the original data) is commonly called the *replication factor*. Note that, the replication strategy adopted coincides with that of many structured P2P architectures (e.g, DeCandia et al., 2007; Ratnasamy et al., 2001; Stoica et al., 2001). In the implementation, the replication factor used is $\frac{1}{2}\log_2 N$, where N is the number of sites.

Corollary 4.3.2

As long as some replicas persist, a data cover can be reliably located.

Proof. This is analogous to Proposition 4.3.1 with the additional condition of the survivability of replicas. □

Corollary 4.3.3

If the processing in question is idempotent, a Katana cover (i.e., a command cover or data cover) is a cover in terms of the CCP.

Proof. This follows directly from Definition 4.3.1 and Proposition 4.3.1, and Definition 4.3.2 and Corollary 4.3.2. □

Proposition 4.3.4

Bounded broadcast is an execution of the CCP on command covers.

Proof. Given the definition of word metric, the command cover identities are non-overlapping and their union will cover the entire identity space (i.e., disjoint union). Therefore, the bounds in the determination of forwards under bounded broadcast conforms to the definition of a charge over the command covers while the forwards are the same as delegations. In this case, the execution of the onReceiveCharge function coincides to that of the forwarding mechanism under bounded broadcast.

The specificity of bounded broadcast as an application of the CCP is that each site has to keep track of the reception of the broadcast message so as to assure the idempotence of the processing (i.e., no double records of message). □

On top of expressing bounded broadcast as the CCP on command covers, Proposition 4.3.4 also indicates how reachability of the broadcast can be

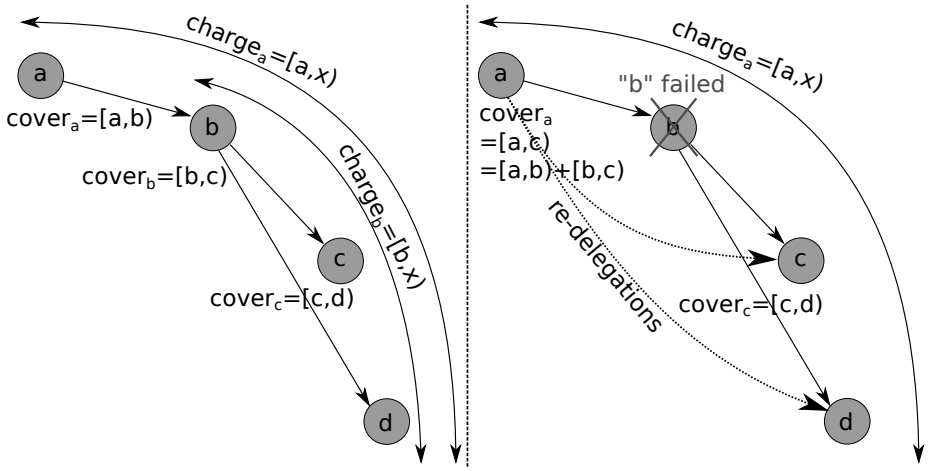


Figure 4.2: Rearrangement of the spanning tree of bounded broadcast

assured. In particular, with the CCP depiction, each site in charge has the responsibility for the broadcast under its assigned CHARGE and for the corrective measures if some COVERs under its CHARGE fail. Note that when corrective measure is effected, the spanning tree of the broadcast may be rearranged but this will not affect the correctness of the broadcast due to the nature of idempotence. For example, in Figure 4.2, the CHARGE of site a is $[a, x)$, which includes the COVERs $[a, b)$, $[b, c)$ and $[c, d)$. Other than its own COVER, site a will delegate the CHARGE $[b, x)$ to site b in accordance to the bounded broadcast. Recall that when a site detects a failed delegate, it will re-delegate the CHARGE to the appropriate site according to the bounded broadcast spanning tree extracted from the by-then-corrected finger tables. In Figure 4.2, when site b fails, site a will take over the CHARGE of site b since it happens to be the appropriate site and thus will also handle its COVER. When the "root" of the bounded broadcast, which is in charge of the entire span of COVERs, fails, some surviving site, preferably its successor[1], will have to take charge of the entire span; this will not incur a severe

[1]The preference of the successor to take over comes from the fact that there is a higher chance of the delegation tree resulting from the successor to coincide with the original tree, hence reducing possible extra work; in actuality, any site can cover for the failed "root" site.

performance penalty regardless of the nature of the COVER since most of the required COVERs (i.e., all except one) are already catered for. Note that this possibility of an "internal" processing site being able to take over the charge of the entire processing is a novel capability that comes from the decentralized nature of the Katana framework; more shall be talked about on this later in Section 4.3.4.2.

4.3.2 Robust KATA Job

Recall that for the processing of a KATA job, bounded broadcast provides the global spanning tree (refer to Section 3.5.3). Therefore, by allowing the (intermediate) results to piggyback on the completion notification of CCP, the bounded broadcast mechanism as implemented over the CCP essentially implements the robust processing of a KATA job. However, some complications arise from the utilization of dual overlays. Note that the bounded broadcast done for the processing of a KATA job (and a ANA job) is based on the command overlay; therefore, when a site fails, the re-delegated site that covers it may not possess the corresponding data COVERs. This is a noteworthy issue for both KATA and ANA jobs.

In order to overcome this problem of disassociation between the command COVERs and the data COVERs, a mechanism specific to the Katana framework, called *secondary delegation* is employed. Recall that the deployment of the dual overlays essentially creates a mapping between the command COVERs and the data COVERs (i.e., surjective from data COVERs to command COVERs). Therefore, through the recording of this mapping, the re-delegated site to cover a failed site can identify the required data COVERs that it is supposed to cover. It then seeks out these data COVERs and delegates their processing accordingly.

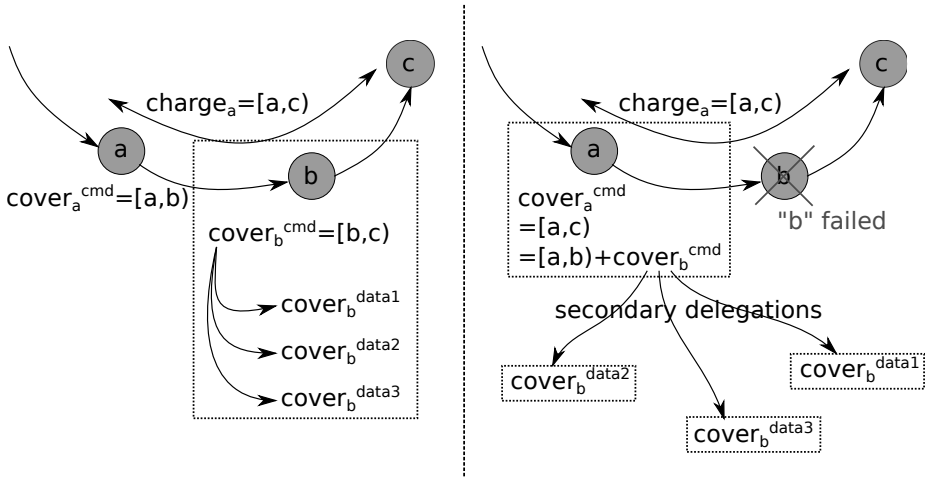


Figure 4.3: Example of a secondary delegation

Figure 4.3 shows an example how secondary delegation is conducted. In Figure 4.3, the CHARGE of site a is $[a, c)$, of which it delegates the sub-CHARGE $[b, c)$ to site b. When site b fails, site a is in charge of re-delegating the lost CHARGE, which it happens to be the correct covering site. However, site a may not possess the data COVERs of site b (i.e., $cover_b^{data1}$, $cover_b^{data2}$ and $cover_b^{data3}$) even though its corrected command COVER covers the command COVER $[b, c)$. In this case, site a will have to seek out these lost data COVERs, which are covered by their respective successors, via the data overlay to perform the secondary delegation.

As mentioned, the (intermediate) results from the application of the KATA function shall piggyback on the completion notification of the delegates, this includes the secondary delegates. Firstly, recall that by the construct of the processing model, the application of the KATA function on each data COVER is intended to be idempotent. And, note that the processing model of the KATA job does not dictate the structure of spanning tree (refer to Section 3.2.2), therefore the rearrangement of the spanning tree, due to the changes in delegation (i.e., both primary and secondary), will not affect the correctness of the processing.

4.3.3 Robust Ana Job

For the processing of an ANA job, bounded broadcast on the command overlay is used as the mean to disseminate the job package (refer to Section 3.5.4), therefore it faces the similar problem of the disassociation between the command COVERs and the data COVERs and thus the processing of an ANA job will also use secondary delegations to solve this problem as described in Section 4.3.2.

However, for the ANA job, the COVERs play not just the role of data input, they are also placeholders for the reception of the data shipment of the output from the applications of the ANA function by the other sites; recall that the purpose of an ANA job is to create a data graph out of a pre-existing data graph and under the processing of the ANA job, the soon-to-be created new data graph is, in a way, a type graph waiting to be filled up (i.e., a placeholder) by the output of the ANA function (refer to Section 3.5.4). Therefore, the re-delegated data COVERs for the ANA job hold the responsibility to broadcast their inclusion into the processing of the job so as to receive the output from the other COVERs; this is done through the robust bounded broadcast as mentioned in Section 4.3.1.

Finally, for the ANA job, as similar to the KATA function, the application of the ANA function on each data COVER is intended to be idempotent. Coupled with the fact that the data COVERs represent a subset of the vertices of a type graph (refer to Section 3.2.1), this means that re-delegation will not affect the correctness of the processing provided that each processor of the data COVER keeps track of the sender (i.e., data COVER identity) of the ANA output.

4.3.4 Discussion

The incorporation of CCP for the processing of Katana operations has some points worthy of further discussion; they are presented in the following.

4.3.4.1 Shape of Delegation Tree

Notice that both MapReduce-style fault-tolerance and "Katana-style" fault tolerance can be seen as specifications of the CCP but their corrective mechanisms seem to differ; under MapReduce, a single site controls the re-delegations while under Katana, the responsibility of re-delegations is distributed across to a number of sites. This is because of the nature of delegation, which is dependent on the framework-specific (reliable) location service. For MapReduce, all forms of location and re-location are directed by the master site, therefore it manifests into a strict one-level-only delegation tree. However for the Katana framework, location service is a decentralized service, thus the delegation tree coincides with the spanning tree as obtained via bounded broadcast.

4.3.4.2 Architectural Robustness

Due to the decentralized nature of the Katana framework, fault-tolerance is assured wholly by the participating sites and does not require the intervention of external availability services. The failure of the master site under MapReduce requires external recovery; recall that the recovery of the `ApplicationMaster` requires the intervention of the `ResourceManager` while the recovery of the `ResourceManager` requires some further external fortification. On the other hand, under Katana, the failure of the "root" of

the bounded broadcast just need any surviving site to take over the initial CHARGE; this applies for the processing of both KATA and ANA jobs. This essentially eliminates the single point of failure; in other words, the induced robustness of the hardened Katana framework is derived directly from its P2P architecture.

4.3.4.3 Definitions of Katana COVERs

A possible point of confusion about the definition of the Katana COVERs is the fact that the command COVER is defined by the successors while the data COVER is defined by the predecessors. The data COVER has to be defined by the predecessors because, as per the nature of CM-DHT, data elements are mapped to the bound between the site and its predecessors.

However, on the other hand, the command COVERs need not be defined by the successors; in fact, any partitioning of the processing sites would suffice since the CCP allows for wrong delegation. So as long as the COVER can be located, the CCP will work. Nevertheless, by defining the command COVER in accordance to the bounded broadcast[2], the CCP can work off the local information from the finger table and need not utilize the global service; in other words, the delegations and the associated sub-CHARGEs can be computed directly from the finger table without any remote communications.

4.3.4.4 Distribution of Correction Load

The Katana framework utilizes dual overlays so as to reduce communication cost. In coping with the disassociation between the command COVERs and

[2]Notice that a command COVER corresponds to the smallest bound under bounded broadcast.

data COVERs, secondary delegation has to be done upon site failure. However, the use of secondary delegation also brings about some unexpected gains. Notice that, upon a site failure, the re-delegated data CHARGEs are likely shared by V distinct surviving sites (i.e., several data CHARGEs), where V is the number of virtual sites employed by the data overlay. Therefore, the overhead incurred is theoretically only a fraction of the running time taken to complete the CHARGE in question. This accounts for the rapid recovery time (i.e., lower overhead) of the CCP in Katana as seen in the experiments (refer to Section 4.4).

4.4 Experimental Study

In order to gauge the performance of the hardened Katana framework, the COVER-CHARGE protocol is embedded into the prototype[3] developed for Chapter 3. Naturally, the implementation is developed with Java 1.6. The experiments are conducted on a 17 machines cluster (i.e., a subset of the configuration used in Section 3.6). Each machine is equipped with an Intel Xeon X3430 Quad Core CPU (2.4 Ghz), 8 GB memory, two 450 GB SCSI disks, and 1 Gbps Ethernet interface. The operating system used in each machine is CentOS Linux 5.6. The Java virtual machine installed is Java HotSpot™ 64-Bit Server VM (build 24.45-b08, mixed mode).

The experimental data is obtained via the Transaction Processing Performance Council Benchmark™ H (TPC-H) generator. The primary relations used are the LINEITEM relation and the ORDERS relation. The experiments will run on data sets generated specifically with the scale factor (SF) of 64. The performance evaluation consists of two of the three runs of the

[3]Source code available at: https://bitbucket.org/xanec/projectkhloe

previously used experiments (i.e., *Document-Length* and *Equi-Join*). The *Document-Length* experiment test the performance of a KATA job upon site failure while the *Equi-Join* experiment test the performance of ANA jobs upon site failure.

For the MapReduce framework, Apache Hadoop 2.2.0, which is equipped with YARN, is used. As mentioned in Section 3.6, the MapReduce jobs are quite sensitive to the number of reducers used. Therefore, according to the results obtained, the R-value selected is the one that produce the faster timings (i.e., $R = 1$ is used for *Document-Length* while $R = 2$ is used for *Equi-Join*). The rest of the configuration is the same as that used in Section 3.6. In particular, for the experiments on Hadoop, one additional machine is dedicated to perform the role of the master site (i.e., resource manager); experiments on Katana uses $N = 16$ sites while experiments on Hadoop uses $N + 1 = 17$ sites. Most of the configurations on Hadoop are unchanged; the only major modification is the reduction of resources (i.e., memory) available per site from 8 GB to 4 GB as an effort to reduce external interference (i.e., experiments of other researchers using the same cluster).

As inspired by the experiments conducted by Dinu and Ng (2012), in order to test the performance penalty incurred by site failure, the experiments are carried out with the killing of a randomly selected site at specific point of the execution. The running time measured is taken as a quotient over the average running time if no sites have failed to obtain the *normalized* running times; this normalized values will indicate fairly the overhead incurred due to the corrective measures taken by the respective fault-tolerance mechanism. Similarly, the time of fault injection is also normalized as a percentage of the average running time without failure. For each experiment, there will be a total of 20 experimental runs (i.e., 10 runs with varied fault injection

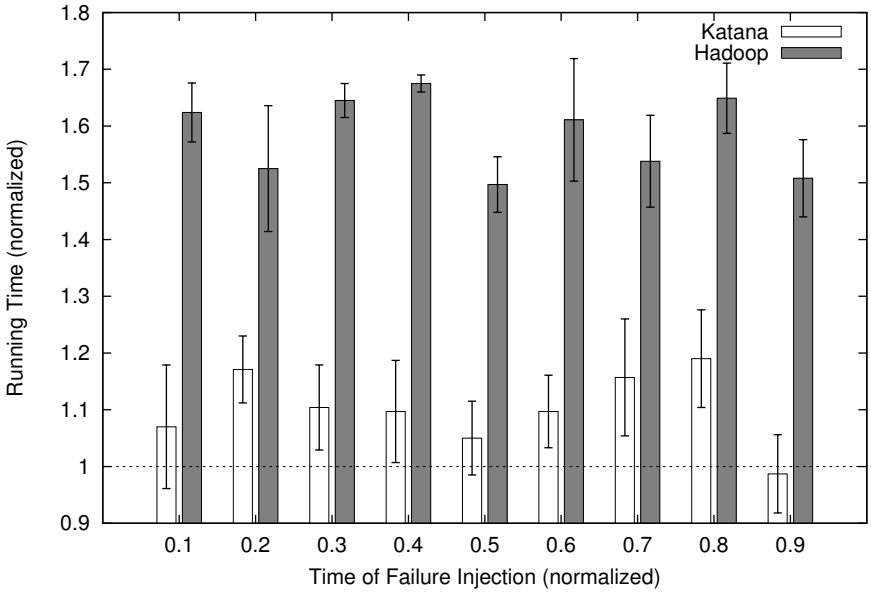


Figure 4.4: Normalized running times of *Document-Length* ($N = 16$, $SF = 64$) upon site failure

point × 2 frameworks); the timing obtained for each experiment run is an average calculated from three distinct executions.

4.4.1 Robust Kata Job: *Document-Length*

Figure 4.4 shows the normalized overhead caused with respect to the point of site failure. Note that Dinu and Ng (2012) has demonstrated that Hadoop suffers from excessive overheads upon site failures due to various conditions on the current execution phase when the failure occurs. However, the study was conducted on the previous version of Hadoop (i.e., not equipped with YARN); Figure 4.4 suggests that such a problem may have been eradicated via YARN. Similarly, the same uniformity in performance can be observed on Katana; moreover, the normalized overhead in Katana is also lesser.

For MapReduce, since *Document-Length* has a heavier map phase as compared to its reduce phase, a significant portion of the work will be lost when

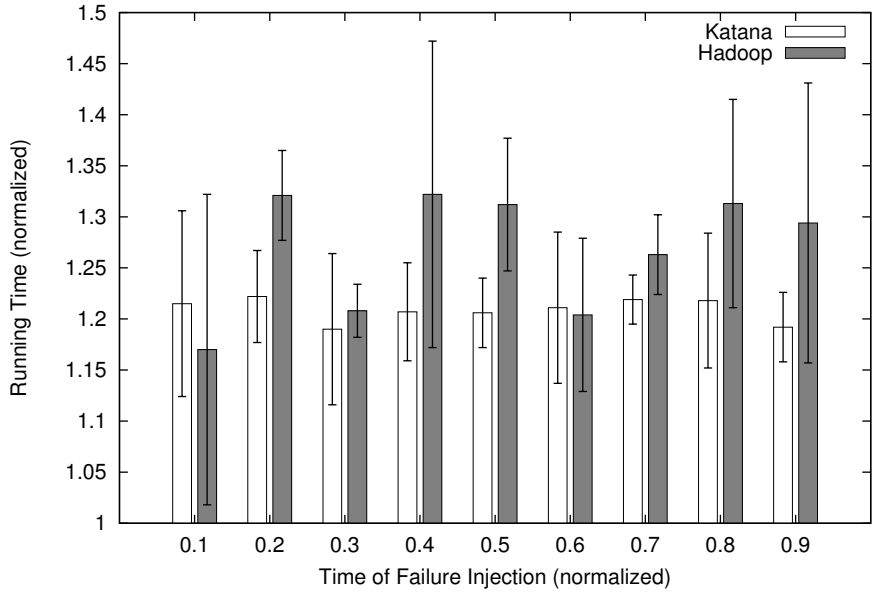


Figure 4.5: Normalized running times of *Equi-Join* ($N = 16$, $SF = 64$) upon site failure

a site fails. On the other hand, Katana outperforms Hadoop in this experiment mainly because the Katana algorithm utilizes only a single KATA job, which maintains minimal amount of internal states as compared to Map-Reduce due to the processing model; the liberty in identifying the spanning tree allows faster repetition of the lost work on a replica. An interesting phenomenon that occurs occasionally in Katana in this experiment is that the failure of a site may actually *improve* the running time (i.e., normalized running time of less than 1). This abnormality happens if by chance, the failed site is a heavily loaded one and the replicas of its data covers are distributed across lightly loaded sites; as a result, the overall running time is ironically reduced. This indicates an opportunity for some optimization schemes to be put in place with regards to query processing but for now, such exploration shall be left for future discussions.

4.4.2 Robust Ana Job: *Equi-Join*

Figure 4.5 shows the normalized overhead caused with respect to the point of site failure. The notable differences between this experiment and that in Section 4.4.1 are that the Katana algorithm uses two ana jobs and that the MapReduce algorithm used is heavier on the reduce tasks. The penalty incurred for Katana is slightly higher than *Document-Length* because ana jobs, in general, have to maintain a higher amount of intermediate data as compared to kata jobs; recall that for ana jobs, the covers are also placeholder to receive output from the applications of the ana function. Nevertheless, this overhead is lesser than that of Hadoop on the average. For Hadoop, the overhead incurred has large variance due to the fact that, under YARN, there is a larger variance in allocating reduce tasks; there is a higher chance that a site failure may not affect the overall progress (i.e., the selected site is not handling any reduce tasks).

Also, on a side note, notice that when failure is injected at 10% into the processing (i.e., normalized time of failure injection is 0.1), the running times of Hadoop is relatively low with high variance; this is because the MapReduce algorithm is still at the map phase implying that the cost of correction can be mild as compared to the overall processing depending on the map tasks assignment.

4.5 Summary

The proposed mechanism to maintain the persistence of processing state, the CCP, is a sound fault-tolerance scheme with abstract generality. The MapReduce-style fault-tolerance can be seen as a specialization of the CCP.

Taken into the context of the Katana framework, various system-wide operations (e.g., bounded broadcast) can be adapted to employ the CCP to ensure robust operations during processing. Furthermore, as the processing of KATA and ANA jobs employ bounded broadcast as their basic communication operator, they too can employ the CCP to render their operations as fault-tolerant. Moreover, due to the utilization of dual overlays, the dispersion of re-delegation of COVERs upon failure inadvertently help reduce the potential overhead of correction.

Experimental studies on the hardened Katana framework indicate that the overhead incurred by the deployed CCP on the processing of KATA and ANA jobs is comparable to, if not significantly lesser than, that of the MapReduce-style fault-tolerance under Hadoop Apache. This suggests that the structured P2P overlay, when using the CCP to assure the persistence of processing state, is a viable option for web-scale processing even in terms of fault-tolerance, especially since the robustness is derived entirely from the architecture and does not require external intervention.

Chapter 5

Elasticity: EMRE

5.1 Motivation

The externalization of resource managers, as mentioned in Section 2.2.3, provides many of the systemic qualities desired by large clusters; other than augmented reliability and scalability, resource managers also better manage multi-tenancy by means of more sophisticated resource schedulers. In a sense, one can say that resource managers play supplementing roles in the support of job executions. However, note that such abstraction exposes optimization opportunities that are not yet fully exploited by the current state-of-the-art MapReduce execution architecture. In particular, with the new environment of larger clusters and higher degree of tenancy, resource managers present potential elastic job execution that has arguably not yet been considered.

In the current architecture, elasticity in job execution is achieved through the observation that the ensemble of map and reduce tasks usually outnumbers the available processing sites and that tasks are executed greedily on

available resources. However, since the number of reduce tasks is a variable fixed by the job submitter, this effectively locks the degree of parallelism expressed during the shuffle phase and the reduce phase, thus limiting the elasticity of the entire job execution. Job submitters typically have to rely on some rule-of-thumb formula based on the amount of available resources to determine the number of reduce tasks. Such an approach is obsolete when the availability of resources is largely unknown and is subjected to runtime changes, which is the case when the degree of tenancy is high. Thus, it will be optimal if the number of reduce tasks can be adjusted dynamically with respect to allocated resources.

The challenge in having a dynamic number of reduce tasks is that the mappers (i.e., the sites executing the map tasks) currently need to know this number in advance so as to statically partition the sorted intermediate map output for the reducers (i.e., the sites executing the reduce tasks). Thus it can be observed that having a dynamically adjustable partitioning is the key to having a dynamic number of reducers. Should one re-imagine this requirement as wanting to dynamically allocate and locate data elements over an undetermined number of sites, notice that this is precisely the problem solved by consistent hashing (refer to Section 2.1) which means one can leverage on structured peer-to-peer (P2P) overlays as potential solutions.

This chapter presents an enhanced execution architecture to run MapReduce job, which is highly elastic to the amount of resources allocated; it is called *Elastic MapReduce Execution* (EMRE). The work is developed with YARN as the target resource manager and there are reasons to believe that the ideas presented can be easily adapted for other resource management frameworks. The following are some of the contributions of EMRE:

- EMRE is the first development to leverage on embedding a structured P2P overlay so as to incorporate elasticity into MapReduce job execution.
- In addition to elasticity, the distributed index of the structured P2P overlay allows EMRE to push intermediate map output to the reducers, thus allowing pipelined execution of jobs.
- The structured P2P overlay allows the worker sites to share some of the maintenance of execution state so as to alleviate the load on the master site.
- From the experiments, the elasticity and pipelined execution of EMRE greatly improves the performance of MapReduce jobs.
- EMRE maintains fault-tolerance despite the improvements to the internal execution.
- EMRE preserves compatibility to the original MapReduce job definition; the experiments run MapReduce jobs *with no modifications* at all.

5.2 Differences in Execution Environment

Picking up from the discussion in Section 2.2.3, even though the MapReduce execution under YARN deviates from the original design of `JobTracker` and slot-based `TaskTracker`s, it assumes similar discrete work-oriented manner of processing. This probably stems from the desire to maintain backward compatibility. However, there are several significant contextual changes with the executions under YARN where the current MapReduce execution can be rather maladjusted to:

- With the potential number and variety of application executions, the availability of resources (i.e., `Containers`) can be quite volatile; other than singly execution, it is difficult to divine the optimal number of reducers, if one even exists under such dynamic environment, to configure for the submission.

- More resources may be requested and allocated at any point of time throughout the execution; under the current MapReduce execution architecture, expansion of resources is only catered for during the map phase.

- The main commodities under YARN are the processing resources but the current MapReduce execution architecture still works with input workload (i.e., tasks) as the main units of execution; such mismatch causes the MapReduce to behave somewhat sub-optimally, such as releasing `Containers` after the completion of each task even when the execution is still ongoing.

This thesis holds the position that it is possible to accommodate these new changes in the MapReduce execution without compromising on the compatibility; the key, as previously mentioned, is to incorporate a dynamically adjustable partitioning on the intermediate map output.

5.3 Observations

The original design of the MapReduce processing model identifies *horizontal* partitioning to the execution architecture, thus defining the different phases of execution (refer to Figure 5.1). Notice that under the horizontal partitioning, the MapReduce processing can be described as a process of *data*

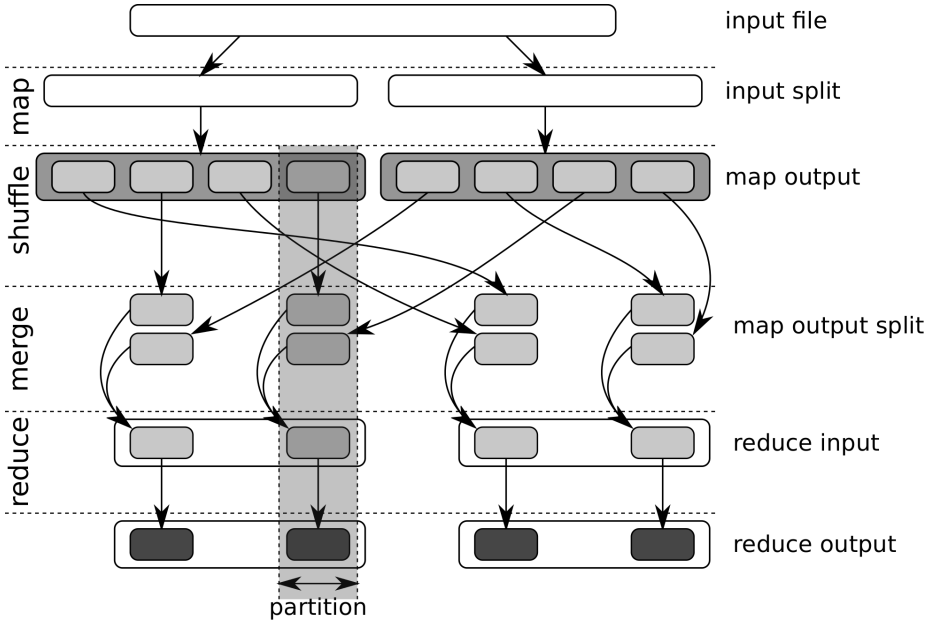


Figure 5.1: Data transformation of MapReduce processing model

transformation through the phases, from the input to the intermediate data and finally to the output.

Now, focusing only on one specific range of hash values, one will essentially create a *vertical* slice on data elements of the map output and of the reduce input. The point of interest is that this vertical slice is a single pipelined, relatively independent unit of work in that, as long as the required data is available at any point, it can proceed with the next transformation without any other synchronizations with others. Therefore, it can be argued that the MapReduce processing model is, by nature, a "push-based" model. However, most MapReduce systems adopt a more "pull-based" model simply because the mappers do not know where to push its output to if the required reducers are not present.

In addition, it is trivial to note that none of the merging process can commence without the completion of *all* the map tasks, and a reduce task cannot begin without some output from the merging process. These *wait conditions* that are embedded in the processing model indicate that the concept

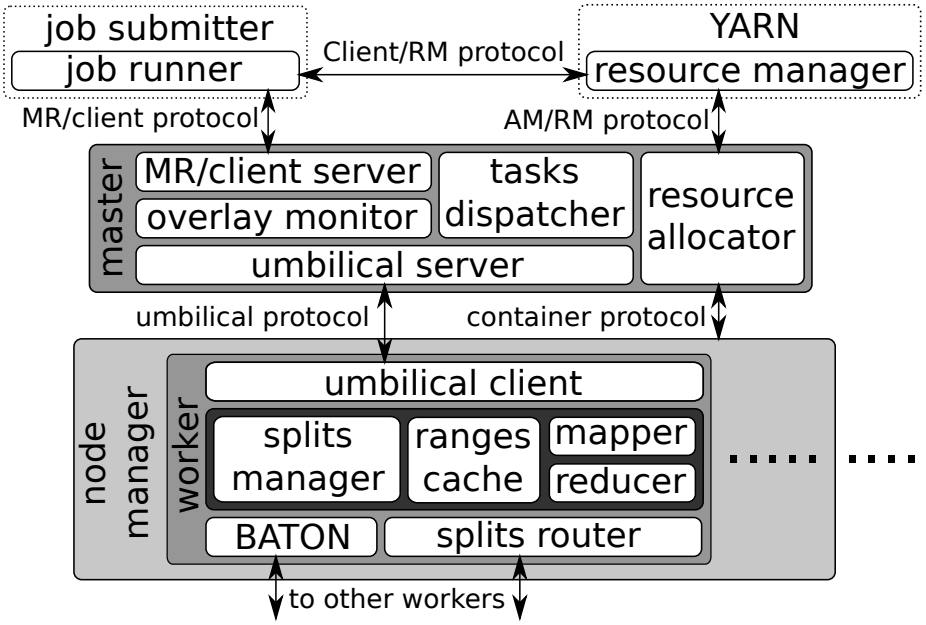


Figure 5.2: EMRE system components

of "phases" is already implicit in the data transformation itself; therefore, there is really no need for an external entity for the explicit execution of the phases.

The design philosophy of EMRE differs from that of the current execution architecture, which considers a MapReduce job as simply a collection of map and reduce tasks. Instead, EMRE views a MapReduce job as a collection of vertical data transformation pipelines. In this way, these pipelines can be easily collated together or be distributed across multiple sites (i.e., elastic job execution); a structured P2P overlay (i.e., BATON) provides a dynamic and robust manner in which the pipelines can be distributed.

5.4 System Design

As EMRE is meant to be deployed within YARN (and other resource managers), it inherits the master/workers architecture. However, as opposed to

the current execution architecture where the master site plays the main coordinator and the worker sites are mere processors of tasks (i.e., oblivious of the one another), the worker sites under EMRE play a bigger role partly as an effort to alleviate the load on the master site. Figure 5.2 displays a rough outline of the relationships between the system components in EMRE.

5.4.1 Master Site

The master site of EMRE, running the `ApplicationMaster` process, plays four main roles in the execution architecture:

- requesting and allocating resources from the `ResourceManager`,
- monitoring the P2P overlay,
- dispatching the map tasks, and
- determining the completion of job.

Notably, the master site no longer coordinates the execution phases of MapReduce; the worker sites will coordinate the execution phases among themselves in a distributed manner.

5.4.1.1 Resource Allocator

As mentioned in Section 2.2.3, the current MapReduce execution architecture requests for resources in accordance to the global phase that the execution is in, under the coordination of the `ApplicationMaster`. This can be seen as a direct effect from the more work-oriented mentality adopted in that architecture. As a result of such mentality, a MapReduce job is

defined as a direct summation of all the map and reduce tasks; and to process the job, the `ApplicationMaster` requests and allocates `Container` to run a processor for each task in a bijective manner. We can say that the current MapReduce execution architecture treats assigned resources as a manifestation of the work.

On the other hand, EMRE adopts a more resource-oriented mentality whereby it is the resources that are the commodities; such is the case for YARN. Upon submission, the `ApplicationMaster` will request for as many `Containers` as possible with preference on the locality of the input sub-files. For each allocated `Container`, the `ApplicationMaster` will launch a worker site in that `Container` indiscriminately. Note that the launched worker sites will persist until the end of the job. This means that each allocated `Container` will only be voluntarily released at the end; it is believed that this is a more appropriate manner in handling allocated resources since there is no reason to release `Containers` when the application still requires more `Containers`. Even if there are no assignable map tasks for a particular worker site, it can still potentially contribute to the job as a reducer. As it shall be revealed later, during the course of execution, it is the tasks that are assigned to the worker sites; while such an approach resembles the original slot-based design of MapReduce, there is significant difference in the role the worker sites play in the architecture of EMRE.

5.4.1.2 Overlay Monitor

For EMRE, BATON (Jagadish et al., 2005) has been chosen as the underlying structured P2P overlay. Even though BATON offers many systemic advantages particularly in the handling range queries, it has been chosen for reasons that are quite different from what the authors intended. In fact,

in EMRE, its support for range queries has been discarded by distributing according to the hash values of the keys, rather than the keys themselves; more will be discussed about this in Section 5.4.2.

Compared to many DHTs, the functioning structure of BATON is relatively independent to the number of participating sites. Note that, for example given N number of participating sites, Chord (Stoica et al., 2001) requires $\log_2 N$ virtual sites to be well-balanced and CAN (Ratnasamy et al., 2001) suggests using a dimension of $\frac{\log_2 N}{2}$; such variable structure is problematic because under the new execution environment, we do not know how many sites are or will be participating. Under BATON, load share at each site may be adjusted with negotiations between its adjacent sites (i.e., without perturbation to the overall structure of the overlay). Such confined disruption facilitates the implementation of some form of work stealing mechanism in the execution architecture. Understand that the *Achillies' heel* in using P2P structures for processing is the inherent load imbalance, therefore having a work stealing mechanism is critical to optimal processing. Even if the load balancing is perfect, the heterogeneity in processing capabilities, due to either heterogeneous hardware or imbalance in resource scheduling, makes work stealing a useful feature to have.

Strictly speaking, since a P2P overlay is adopted, there should not be a need for an external centralized monitor to maintain the state of the system. This is because the worker sites should be able to represent the global state by themselves in a distributed manner, which is really the whole point of adopting a P2P overlay. However, there are actually three reasons that the `ApplicationMaster` still has to play a monitoring role over the worker sites in EMRE.

Firstly, the stability of BATON requires a "virtual site" to monitor the root site of the tree (Lupu et al., 2008). Note that this "virtual site" neither stores any data elements nor perform any processing; its only purpose is to restructure the overlay if the root site fails. Thus, it does not take much effort for the `ApplicationMaster` to perform such a role.

Secondly, other than for reporting purposes, there is a need to manage the status of all the tasks and to recover them in times of failures so as to maintain correctness of execution. For this, the `ApplicationMaster` has to also keep track of the assignments of tasks to the worker sites, which is similar to what the current execution architecture does for the locations of the intermediate outputs.

Finally, the `ApplicationMaster` determines the completion of the MapReduce job execution. Note that, the determination of job completion can be done individually by the worker sites in a distributed manner. However, for a more robust execution, it has been decided to rely on the `ApplicationMaster` to monitor the completion. Actually, since the `ApplicationMaster` has to monitor the job progress as part of the design of MapReduce, it is trivial to extend this monitoring to cover the determination of job completion.

5.4.1.3 Tasks Dispatcher

Once a worker site has been successfully launched in a `Container`, it will commence in sending periodic *heartbeat messages* to the `ApplicationMaster` via the umbilical protocol. Similar to the mapper and reducer processes in the current execution architecture, these heartbeat messages carry the status reports of the tasks assigned to the particular site. In addition, the

response messages for these heartbeat messages also carry the assignments of map tasks.

The mapper component of a worker site will indicate in the heartbeat messages whether it is currently performing any work. If the mapper component reports that it is available for tasks, the tasks dispatcher will check for any unassigned map task with preference on data locality to assign to that worker site. If there are, this assignment will be registered with the overlay monitor for purposes mentioned previously and the information (e.g., input sub-file location) of an unassigned map task will be included in the response message for that mapper to execute. Note that such an assignment mechanism will also enjoy the balancing effect that the faster sites will be assigned more map tasks.

5.4.2 Worker Sites

As evident in Figure 5.2, the components within the worker site are much more complex than just a mapper or a reducer; in fact, each worker site consists of *both* a mapper component *and* a reducer component. Each worker site is also equipped with a BATON component, which has also been slightly modified to suit the execution context. At the core of each worker site is a splits[1] manager that manages the storage, distribution and processing of all the intermediate data produced during the execution.

[1]Given that the data elements (i.e., key/value pairs and key/values chunks) are sorted and grouped according to their hash values, the term *split* shall be used to refer to a contiguous block of data elements over a range of hash values.

5.4.2.1 Intrinsic Synchronization

Before delving into the designing of the worker sites under EMRE, it is perhaps pertinent at this point to introduce a rather intriguing phenomenon experienced by the worker sites during the execution: *intrinsic synchronization*.

Definition 5.4.1: Intrinsic Synchronization

Due to a number of reasons, such as the wait conditions as discussed in Section 5.3, there is often an illusion of coordinations between the worker sites during the course of execution even though no such coordination is put in place nor is it required. As such, the worker sites end up doing the same thing at the same time (or at least within a short time frame). Such a phenomenon is defined as intrinsic synchronization.

For example, through empirical observations, all the mapper components seem to complete their tasks at almost the same time; this is likely due to the fact that map functions are usually quite similar in execution. Also, all the worker sites will commence the merging process simultaneously; this is caused by the completion of the last map task being the "starting pistol." Though not particularly a problem by itself, such intrinsic synchronization aggravates the load of many of the processing works done in EMRE if not handled properly. Take the merging process for example, as all the worker sites start to read from hard disk at the same time, the `Containers` that co-locate in the same machine will affect the disk seeks and page caching of each other such that overall throughput will be degraded significantly. In the course of the discussion, readers shall come across several such examples and how EMRE cater for them. On a side note, under the current execution

architecture, disk I/O for merging is reduced by holding as much of the fetched data in memory as possible and merging them directly.

5.4.2.2 Mapper

As previously mentioned, the mapper component will indicate its availability through the heartbeat messages sent via the umbilical protocol. Upon the assignment of a map task, it will proceed with applying the map function on each key/value pair similar to that of the current execution architecture. The main difference between the mapping process in EMRE and that in the current architecture is the partitioning of the intermediate map output.

In the current execution architecture, the intermediate output will be sorted and partitioned according to the number of reducers as defined by the job submitter; and the file offset and length of each partition will be recorded into an index file. The indexing of the intermediate output is essential to facilitate the reducers to fetch the appropriate partition. In EMRE, such indexing is also required for similar reason but there are some additional deliberations in determining the granularity of the partition size. This decision is very critical because it directly affects the unit of work adopted in EMRE, more about this shall be discussed in Section 5.5.

Recall that with the paradigm of consistent hashing, the data elements will first be grouped according a set of buckets, which is achieved via some hash function, and then they are distributed in buckets across the participating sites. If the bucket size is too small (i.e., large set of buckets), the index file will bloat up; consider the case where there are 2^{32} buckets (i.e., hashing to 32-bit integers), then with an index entry of an file offset (e.g., 64-bit unsigned integer) and a partition length (e.g., 64-bit unsigned integer), the

index file of each intermediate output will be 64 gigabytes. On the other hand, if the bucket size is too big (i.e., small set of buckets), the architecture quickly loses the elasticity aspect of the execution; note that this is precisely the case for the current execution architecture, where the number of buckets is equal to the number of reducers. An additional consideration is that even if the granularity for the intermediate map output is optimally determined, the same granularity may be too limiting for the reducer because the reducer will be dealing with only a subset of the buckets and should be able to handle a finer granularity.

With these considerations, two different granularities for the index files on the intermediate files is used depending on the phase of execution; they are called *major-partitioning* and *minor-partitioning*.

Definition 5.4.2: Major- and Minor- Partitions

The major-partitioning (i.e., the one with coarser granularity) is used for the immediate output from the mapper component. The minor-partitioning (i.e., the one with finer granularity) is used for the output produced from merging all the splits; this output is used as input for the reducer.

For the experiments, the implementation uses a major-partitioning of 2^{10} $(= 1,024)$ buckets and a minor-partitioning of 2^{12} $(= 4,096)$ buckets. If the hash values of the keys are also considered, which can be seen as buckets, this will essentially create a three-levels hierarchy of buckets; in the implementations, the keys are hashed to a 30-bit integer, thus a major-partition will contain 2^2 $(= 4)$ minor-partitions and a minor-partition will contain 2^{18} $(= 262,144)$ hash values.

Recall that, unlike most DHTs, BATON is unique in the way it distributes data elements according to the sorted order of the keys so as to allow efficient processing of range queries (refer to Section 2.1.3); this suggests that the hashing of keys might actually be unnecessary, or even counter-productive. In fact, as mentioned previously, by hashing the keys and dealing with the hash values directly, this speciality of BATON is lost. Moreover, one could have designed EMRE *without* the use of hashing while preserving the overall functioning as described in this chapter. However, there are three main reasons as to why an additional hashing is done in the deployed BATON other than the fact that range queries is not of concern to EMRE:

- In most situations, the hash value is significantly smaller than the actual key used. Thus, this will alleviate the memory usage in a lot of the internal operations, such as the management of the splits and the maintenance of the ranges cache. Furthermore, the serialization of meta-data involving the sub-ranges will also be faster due to the reduced size.

- Handling a system primitive type such as an integer is much faster than handling a generic key type as defined by the user. In particular, the sorting operation has much to gain from the faster (i.e., native speed) integer comparison.

- From a development perspective, handling integers facilitates the implementation of a lot of the internal operations such as estimation of split size, load distribution, as well as various interval (i.e., range) operations (e.g., intersection and union). These operations will be more complicated to implement if the generic key types were to be used.

5.4.2.3 Reducer

The reducer component of the worker site is concerned with only applying the reduce function on the key/values chunks when they are available. Unlike its counterpart in the current execution architecture, the reducer component in EMRE does not perform the fetching or merging of input data; this is handled by the splits manager because this process is much more sophisticated under EMRE. In the case for a map-only MapReduce job, the reducer component will not be activated.

Note that for most MapReduce implementations (e.g., Apache Hadoop), each reduce task will produce a distinct sub-file as a side effect of the processing model; this is also the case for EMRE. Therefore, the number of reducers determines informally the number of output sub-files. There may be very rare situations where the job submitter requires a particular number of output sub-files from the MapReduce job for purposes other than control of parallelism, though no practical examples can come to mind. Thus in these cases, EMRE will not meet the expectation of the job submitter and may not be compatible to the usage of the original job since each worker site is equipped with one reducer component implying that each worker site corresponds to one reduce task in a full MapReduce job. However, there is actually no specification in the MapReduce programming model that dictates the number of output files. Nevertheless, this is probably the only occasion where EMRE may have compatibility issues with the original intent of the MapReduce job definition.

5.4.2.4 Splits Management

In EMRE, the intermediate outputs from the mapper component are transferred directly to the splits manager and the splits manager is also in charged

of dispatching input data to the reducer to process. In order to incorporate elasticity, processing in EMRE is done in relatively small chunks, as compared to the current execution architecture; in fact, map output splits are merged in major-partitions while input to reducer is transferred in minor-partitions. Therefore, one important role of the splits manager is to maintain the status of the splits (e.g., "available", "merging" or "merged" for the map output splits) to ensure correctness in the execution. Other than processing, the re-distribution of the splits (i.e., shuffling and work stealing) and the transformation of the intermediate map output splits into reduce input are also coordinated by the splits manager. The role that the splits manager plays in the overall job execution shall be described more in details later in Section 5.5.

One important aspect of EMRE, and P2P architectures in general, is its distributed nature with minimal or no central coordinations. Such a quality will further augment the scalability of the framework because the global functioning of the entire system depends only on the individual correct functioning of each local element. As previously mentioned, the incorporation of the BATON overlay allows the maintenance of the system state; with regards to the processing, on the other hand, the splits manager plays the orthogonal role of maintaining the global processing state of the framework.

Note that in the case of a map-only MapReduce job, the execution is relatively straightforward in that there is no requirement for the re-distribution of splits; therefore the splits manager, just like the reducer component, will not be activated for this kind of jobs.

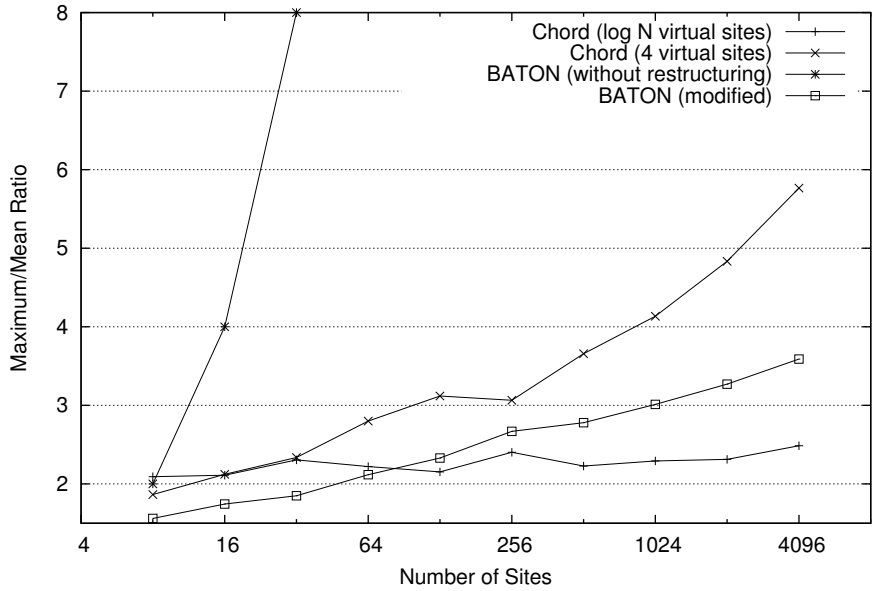


Figure 5.3: Maximum/Mean ratios of some structured P2P overlays

5.4.2.5 Modified BATON

The original design of BATON considers restructuring as a rather common phenomenon. After all, most structured P2P overlays are designed for a much harsher environment whereby site participations are very dynamic. As such, one critical ingredient for load balancing mechanism in BATON is the forced restructuring of the overlay by an artificial departure of a lightly loaded site. Such a mechanism will not be feasible in EMRE because a departure of a site means that its load has to be migrated entirely to its adjacent sites and with the potential amount of data held, this can be prohibitively costly and, not to mention inefficient too.

Without restructuring, the load imbalance in BATON can quickly go out of hand with just the sharing on join. Figure 5.3 shows that a simulated BATON overlay without restructuring has load imbalance that is exponential to the number of participating sites. Note that the ratio of maximum load over mean load is used as an measurement of the load imbalance; this

should be quite a fair metric since the overall performance of MapReduce is tied to the slowest site.

To handle the conflicting problems of load balancing and data migration, two simple modifications are applied to the BATON overlay used in EMRE. Firstly, the adjacent fingers of the modified BATON are circular; this means that the "left most" site will have a left adjacent finger pointing to the right most site, and vice versa on the right adjacent finger of the "right most" site. Secondly, rather than sharing half the load of its parent, the new joining site will share the load of both its adjacent sites in a well-distributed manner. For example, if the left adjacent site has N_l data elements and right adjacent site has N_r data elements, the new site will get about $N = \frac{N_l+N_r}{3}$ data elements by retrieving the upper $\max(N_l - N, 0)$ data elements from the left adjacent site and the lower $\max(N_r - N, 0)$ data elements from the right adjacent site. In this way, each join will seek to re-balance the load, at least locally.

Note that with just the two mentioned modifications, the load of the modified BATON becomes better balanced than Chord with constant number (i.e., four) of virtual sites (refer to Figure 5.3); though it is not as well balanced as Chord with $\log_2 N$ virtual sites, which has issues implementing in the execution context as previously mentioned. Nevertheless, having such manageable load imbalance, the shortfall can be further covered relatively well by a work stealing mechanism, which shall be described in Section 5.5.

The trade-off for incorporating these two modification is a more involved joining algorithm; in particular, the sharing of loads from both adjacent sites may cause a reduction in robustness of the original design. However, as previously noted, even though the conceptualization of MapReduce considers site failures as a regular phenomenon, a deployed MapReduce cluster

still has a relatively benign operating environment as compared to the ones structured P2P overlays are typically designed for. Therefore, such a compromise on the robustness for a much better load balancing is a fair deal.

5.4.2.6 Ranges Cache

Once the mapper component has completed a map task, its output will have to be shipped directly to the corresponding worker sites. This requires the discovery of the sub-ranges of all the participating sites. The underlying problem is that, with intrinsic synchronization, this becomes a situation whereby *everyone has something to send to everyone else*, meaning the system has to process $O(N^2)$ messages instantaneously. Therefore, the problem itself is not scalable.

Under EMRE, each site maintains individually a directory of other sites and their sub-ranges (i.e., the ranges cache). When joined, each newly-joined site will inherit its parent's directory as an initial copy. Recall that the BATON overlay has been modified to be circular. So, to propagate the sub-ranges information, each worker site will periodically announce its sub-range to its two adjacent sites, who will update their own ranges cache and forward the information to their corresponding left or right adjacent sites.

Notice that the propagation mechanism only requires the system to process $O(N)$ messages, which is much more manageable than a broadcast. However, the trade-off is that mis-sends may occur due to outdated cache; the recipient then has to re-send the split to its correct location. As the sites that are closest in terms of adjacency to a particular site will first receive the updated sub-range, we are guaranteed that the splits will eventually reach the correct site due to the increasing precision of the directory as the

splits are re-sent. Naturally, each mis-send incurs penalty of additional I/O, though our experience suggests that such propagation method works relatively well even with larger clusters. This is because outdated ranges cache result from *changes* in cluster size (i.e., resource allocation/deallocation) rather than its actual size; note that the current architecture does not cater to changes in cluster size and, as evident in the experiments, suffers much more severely.

5.5 Elastic Job Execution

This section shall describe the MapReduce job execution under EMRE. Note that even though the different aspects of the job execution are described linearly, these different processes may be (and often are) running *concurrently*. For example, there may be new launches of worker sites (i.e., due to delayed allocation) while some of the sites are still merging the map output splits and the others are already applying the reduce function on the reduce input splits. Recall that such dynamic processing can be performed because the MapReduce execution can be partitioned "vertically" (refer to Section 5.3). Essentially, with the vertical partitioning, EMRE breaks down the concept of global phases in the current MapReduce execution architecture. It is precisely such asynchronous work processing that allows very elastic participation of the sites.

5.5.1 Worker Site Launch

A MapReduce job definition usually consists of some meta information regarding the data input; in particular, if a distributed file system (e.g.,

HDFS) is used, it will indicate the locations of all the distributed input sub-files. Upon job submission, the `ApplicationMaster` of the current execution architecture will typically request for three `Containers` for each sub-file:

- one with the host as the split (i.e., data-local),
- one within the same rack as the split (i.e., rack-local), and
- an arbitrary one as the last option.

The `ApplicationMaster` of EMRE performs the same request with the input defined by the job definition. However, the difference is that with each allocated `Container` during the startup (i.e., the map phase), the `ApplicationMaster` of the current execution architecture will assign the map task to the `Container` with preference to the data locality but in EMRE, there are no phases and the `ApplicationMaster` will launch a worker site process in the `Container` regardless of the execution progress.

When launched, each worker site will start as a singular (i.e., not joined) BATON site. In the periodic heartbeat messages to the `ApplicationMaster`, the worker site will indicate its singular status as long as it is not joined. The `ApplicationMaster`, upon the reception of such a heartbeat message, will then identify a joined site (or the first site) and include its location in the response message. Once the worker site receives the location of a joined site, it will proceed with the BATON joining procedure to finally locate the actual site to attach to as its child site. The difference here, as noted in Section 5.4, is that this new site will negotiate with both its adjacent sites for the actual range of hash values that it will be governing (i.e., its sub-range). As mentioned previously, the new site will seek to re-balance

the load among its adjacent sites but, as we shall see later, this is subjected to some statistical estimation. The new site will update its adjacent sites of their new sub-ranges to finish the joining procedure and thus completing the launch of the worker site. It is noteworthy to highlight that at this point, only the sub-ranges are updated, the data elements will not yet be migrated.

5.5.2 Map Task Execution

The tasks dispatcher of the EMRE `ApplicationMaster` is in charge of dispatching map tasks to the worker sites when they indicate in the heartbeat messages that they are available. In addition, in order not to burden the execution with too many concurrent activities, the tasks dispatcher will only dispatch map tasks when a certain threshold of the currently launched worker sites are already joined; in the implementation for the experiments, this threshold is set at 80%. The map tasks assignment is done with preference to the proximity of the work site; that is to say that if there is an unassigned data-local map task, it will be assigned first, followed by an unassigned rack-local map task, and then finally an arbitrary one.

When the worker site receives the assignment of a map task, it will commence the application of the user-defined map function on the key/value pairs of the input data. At this point, the execution is exactly the same as that of the current execution architecture, including the use of *combiner* function if such optimization mechanism is defined. The output from the map function (or the combine function) differs from the current execution architecture in that it will be partitioned and indexed according to the major-partitioning, as mentioned in Section 5.4. The intermediate output

will be kept in its entirety with the local splits manager for recovery purposes if required. To proceed with the execution, the output will have to be copied, split and distributed to all the other worker sites. The split ranges used to partition the output and the corresponding location to send the splits are retrieved from the ranges cache (refer to Section 5.4.2.6) for the most part. Due to anticipated dynamism in available resources (e.g., addition of sites from `Scheduler` or removal of failed sites), the worker site may, at times, have an incomplete view of the overlay (i.e., gaps in the cache); in this case, discovery via bounded broadcast may be employed.

As mentioned in Section 2.1, bounded broadcast is a technique employed in some DHTs to efficiently broadcast messages without resorting to flooding. In order to broadcast, each message will be tagged with a particular bound on the site identification; the recipient will forward the message to the fingers that fall within the bound and each forwarded message will be tagged with an appropriate partition of this bound. Thus, a requirement for efficient bounded broadcast is a way to enforce *total order* on the participating sites; for example, bounded broadcast can be done easily on a Chord overlay with bounds on the site IDs because its site IDs constitute a factor ring. A direct way of implementing bounded broadcast on BATON is via the sub-ranges since these ranges definitely form a total order. However, the range of a site is such a fickle state, therefore the decision is made to use instead the site ID of BATON as the bound; this is slightly more complicated because the ID of a BATON site consists of a level/number pair. In order to enforce the total order, suppose (l, n) represents the site ID where the level is l and the number is n, $(l, n) \in \mathbb{B}$ where

$$\mathbb{B} \triangleq \left\{(l, n) \mid (l, n) \in \mathbb{N} \times \mathbb{Z}^+, n \leq 2^l\right\}$$

the following binary relation $\preceq$ on the site IDs can be defined:

Definition 5.5.1

$$(l_1, n_1) \preceq (l_2, n_2) \triangleq f(l_1, n_1) \leq f(l_2, n_2)$$

where

$$\forall (l, n) \in \mathbb{B},\ f(l, n) \triangleq (2n-1)\left(\frac{1}{2}\right)^{l+1}$$

Lemma 5.5.1

f is injective.

Proof. Given $(l_1, n_1), (l_2, n_2) \in \mathbb{B}$, consider the following cases:

$l_1 = l_2 \wedge n_1 \neq n_2$**:**

$$\begin{aligned} n_1 \neq n_2 &\Rightarrow (2n_1 - 1) \neq (2n_2 - 1) \\ &\Rightarrow (2n_1 - 1)\left(\frac{1}{2}\right)^{l_1+1} \neq (2n_2 - 1)\left(\frac{1}{2}\right)^{l_2+1} \quad \text{by } l_1 = l_2 \\ &\Rightarrow f(l_1, n_1) \neq f(l_2, n_2) \end{aligned}$$

$l_1 \neq l_2$**:** Suppose $f(l_1, n_1) = f(l_2, n_2)$, without loss of generality, assuming $l_1 > l_2$,

$$\begin{aligned} & (2n_1 - 1)\left(\frac{1}{2}\right)^{l_1+1} = (2n_2 - 1)\left(\frac{1}{2}\right)^{l_2+1} \\ \Rightarrow\ & \frac{2n_1 - 1}{2n_2 - 1} = \left(\frac{1}{2}\right)^{l_2 - l_1} \\ \Rightarrow\ & \frac{2n_1 - 1}{2n_2 - 1} = 2^{l_1 - l_2} \end{aligned}$$

$$\Rightarrow 2n_1 - 1 = 2^{l_1 - l_2}(2n_2 - 1)$$

which is a contradiction since LHS (i.e., $2n_1 - 1$) is odd but RHS (i.e., $2^{l_1 - l_2}(2n_2 - 1)$) is even, meaning $f(l_1, n_1) \neq f(l_2, n_2)$.

Taking both cases together, this means that

$$\forall (l_1, n_1), (l_2, n_2) \in \mathbb{B},\ (l_1, n_1) \neq (l_2, n_2) \Rightarrow f(l_1, n_1) \neq f(l_2, n_2)$$

Therefore, f is injective. □

Theorem 5.5.2

$\preceq$ establishes a total order on $\mathbb{B}$.

Proof. Consider the following qualities:

totality:

This follows directly from the fact that f is a well defined function. Therefore, $\forall (l_1, n_1), (l_2, n_2) \in \mathbb{B}$, $(l_1, n_1) \preceq (l_2, n_2)$ or $(l_2, n_2) \preceq (l_1, n_1)$.

reflexivity:

This follows directly from the fact that $f : \mathbb{B} \to \mathbb{R}$. Therefore, $\forall (l, n) \in \mathbb{B}$, $(l, n) \preceq (l, n)$.

anti-symmetry:

Given $(l_1, n_1), (l_2, n_2) \in \mathbb{B}$, if $(l_1, n_1) \preceq (l_2, n_2)$ and $(l_1, n_1) \neq (l_2, n_2)$, then

- $f(l_1, n_1) \leq f(l_2, n_2)$ by Definition 5.5.1, and

- $f(l_1, n_1) \neq f(l_2, n_2)$ because f is injective.

This means that

$$f(l_1, n_1) < f(l_2, n_2) \Rightarrow \neg\big(f(l_2, n_2) \leq f(l_1, n_1)\big)$$
$$\Rightarrow \neg\big((l_2, n_2) \preceq (l_1, n_1)\big)$$

transitivity:

Given $(l_1, n_1), (l_2, n_2), (l_3, n_3) \in \mathbb{B}$,

$$(l_1, n_1) \preceq (l_2, n_2) \wedge (l_2, n_2) \preceq (l_3, n_3)$$
$$\Rightarrow f(l_1, n_1) \leq f(l_2, n_2) \wedge f(l_2, n_2) \leq f(l_3, n_3)$$
$$\Rightarrow f(l_1, n_1) \leq f(l_3, n_3)$$
$$\Rightarrow (l_1, n_1) \preceq (l_3, n_3)$$

Since $\preceq$ is total, reflexive, anti-symmetric and transitive, therefore $\preceq$ establishes a total order on $\mathbb{B}$. □

Having such a relation, total ordering can be enforced on the sites with just binary comparisons; in fact, the order obtained with this binary relation will be similar to that of the order of sub-ranges by the worker sites as maintained by BATON.

Via bounded broadcast, the worker site with incomplete view can make a request for the missing ranges to all the worker sites in the execution architecture. However, this is potentially a costly operation, especially with the possible occurrence of intrinsic synchronization; therefore this will be performed only when the incompleteness is deemed heuristically as severe (e.g., a gap of more than twice of either the ranges adjacent to the gap).

Otherwise, estimated correction will be performed (e.g., missing range is sent to least loaded site adjacent to the gap).

With the ranges, the mapper component from the worker site will select the appropriate major-partitions out from the intermediate output using the index file and send them to their corresponding locations. Note that this is a true pipelined execution model for MapReduce; and it is only possible precisely because the index is distributed and each worker site also functions as a reducer such that it can know *a priori* where to send the splits to.

5.5.3 Splits Routing

The splits management is one of the most important element of the entire EMRE architecture because it is the component that coordinates the MapReduce processing in a distributed manner.

At any point of time during the execution, the splits manager may receive splits from any of the worker sites (including from itself). There are two types of splits that it may receive: map output split and reduce input split. Map output splits are major-partitioned splits of intermediate output from the mapper components; the splits manager can receive map output splits directly from the mapper components or as part of the work stealing mechanism. When the splits manager receives all the required map output splits, it will proceed to merge them to produce a minor-partitioned reduce input split, which will be transferred eventually to a reducer component to process. Similarly, as part of the work stealing mechanism, the splits manager may also receive yet-to-be-processed reduce input splits from other worker sites.

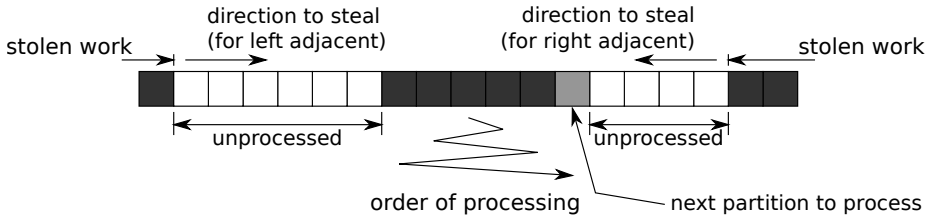


Figure 5.4: Order of processing of the partitions

In order to have a more pipelined execution and also to allow finer-grained work sharing, the processing of the received splits is done in chunks. When the splits manager receives all the map output splits to cover a particular major-partition range (i.e., a "slice" of the actual sub-range of the site), it may proceed to merge these splits over this partition range to produce a reduce input split. Similarly for reduce input split, either produced from the merge or received from other sites, the data is transferred to the reducer component in minor-partitions when they are available. The split manager will select the partition, be it a major-partition of a map output split or a minor-partition of a reduce input split, that is closest to the center of the sub-range of that worker site to begin processing; subsequent selections of the partitions will branch out left and right in a zig-zag manner from the center (refer to Figure 5.4). Essentially, such a way of choosing partition creates a double-ended work stealing queue of splits that allow further work sharing for the joining of new worker sites or for the work stealing from adjacent worker sites.

As seen in the launch of the worker sites, updates to the sub-ranges will not reflect immediately on the splits actually held at a particular site. The splits manager will instead run a periodic thread to adjust the splits held locally with respect to the current sub-range. There are two reasons for such a delayed reaction. Firstly, in order to have a more robust execution, it is preferable that state updates (i.e., sub-ranges and fingers) be kept

as brief as possible. Secondly, the sub-ranges of the worker sites will be altered throughout the execution either due to the joining of new worker sites or as part of the work stealing mechanism. Therefore, it is a common phenomenon for a splits manager to receive splits that have ranges that do not coincide exactly with its sub-range; there will be too much overhead to immediately adjust the out-of-bound splits upon reception.

5.5.4 Reduce Task Execution

As the reducer component receives the minor-partitions, it will commence the application of the user-defined reduce function on the key/values chunks within each minor-partition of the reducer input. Again, this part of the execution is exactly the same as that of the current MapReduce execution architecture.

Given that the splits manager will transfer the minor-partitioned reduce input splits to the reducer component for processing in a zig-zag manner. This means that there will not be any assurance on the order in which the reduce input is processed. Note that in the current execution architecture, there is an implicit and unofficial assurance that the order of execution of the key/values chunks is also sorted according to the key. However, under EMRE, such assurance will be complicated, especially with the three levels of buckets hierarchy. Therefore, if the ordering of execution is required (e.g., for MapReduce sorting algorithm), the job submitter has to indicate this desire so that the reducer will re-order the output accordingly before committing to the final output. Otherwise, the reducer will simply write the output as they are produced, which does not pose a problem for mass majority of jobs and is much faster in operation.

Once the range of the processed reduce input covers the entire sub-range of the worker site, the reduce task corresponding to that worker site can be deemed to be completed. However, as we shall see later, such a status may not be absolute because its sub-range may be enlarged due to various reasons. Therefore, the `ApplicationMaster` has to play the final coordinating role in determining the actual completion; the job is completed only when all the map tasks have been dispatched *and* all the worker sites have reported their completion. At that point of time, the `ApplicationMaster` will indicate the completion on the responses to the periodic heartbeat messages from the worker sites so that the reducer component of the worker sites can proceed to commit the output into the file system.

5.5.5 Work Stealing

As previously mentioned, due to the inherent load imbalance of BATON and possible heterogeneous processing capabilities, the processing of work will not be uniform across the worker sites. Therefore, work stealing is a good feature to have in EMRE to soften the load imbalance. Note that the work of a site can only be stolen by its adjacent sites and the left (right) adjacent site can only steal the lower (upper) data elements; such a restriction is enforced to maintain the structural integrity of the execution architecture.

The work stealing mechanism is achieved with a mere adjustment to the sub-ranges of the thief (i.e., the worker site that is stealing) and its adjacent sites; recall that with the splits management, the splits will eventually be redistributed accordingly with respect with the updated sub-ranges. To initiate the stealing process, the thief will notify its adjacent sites of its intention to steal and which direction it is stealing from. The target of

the theft will decide on the amount of work to be stolen based on some statistical heuristics on its performance and the thief's performance, as well as the number of partitions available for stealing (i.e., *one* of the unprocessed parts in Figure 5.4). Roughly speaking, the number of partitions to be stolen is calculated so as to balance the time taken to send the stolen work and to process them locally.

Note that in order not to thrash the system with disk I/Os, it is preferable that data shipping and processing (including merging) are *not* concurrent, Therefore, in EMRE, the worker site alternates the checking of sub-ranges and processing of tasks; as a result, the time consumed to complete a task is the direct sum of the time taken to transfer (due to theft) and the time taken to process. As such, temporary time statistics are collected with regards to these two events. These statistics are collected and updated throughout the processing of the job:

- $T_{receive}$, time taken to receive a partition,
- T_{send}, time taken to send a partition, and
- $T_{process}$, time taken to process a partition

Each statistical value is calculated as a weighted average of collected values based on their freshness. The most recent N values are stored together with their time stamp where N is a pre-configured value: $(T_1, t_1), \ldots, (T_N, t_N)$. For the experiments, the implementation uses $N = 16$. $\forall i = 1 \ldots N$, T_i is the i^{th} value, t_i is the i^{th} time stamp and $t_1 > \cdots > t_N$. Suppose t_{curr} is the current time stamp, the weighted average is calculated as follows:

$$T_{avg} \triangleq \frac{\sum_{i=1}^{N} \frac{T_i}{t_{curr} - t_i}}{\sum_{i=1}^{N} \frac{1}{t_{curr} - t_i}}$$

Due to the nature of the work and the granularity of the partition, each worker site has to maintain two independent sets of statistics (i.e., one for map tasks and another for reduce tasks). For the brevity of discussion, the specificity of the task will not be assumed as the concept applies similarly for either map or reduce task.

Suppose site a intends to steal from site b and without loss of generality, site a is on the left of site b it will send its $T_{receive}$ value and $T_{process}$ value to site b, labelled $T^a_{receive}$ and $T^a_{process}$ respectively. Upon receiving the intention to steal, site b will calculate the number of partitions to be stolen. Firstly, the time taken to transfer a partition, $T_{transfer}$, is calculated as $\max(T^a_{receive}, T^b_{send})$. Let L be the number of partitions available on the left and R be the number of partitions available on the right. Furthermore, let $T^b_{receive}$ and $T^b_{process}$ be respectively the $T_{receive}$ and $T_{process}$ values of site b. The number to be stolen, S, is calculated as

$$S = \begin{cases} \min\left(\left\lfloor \frac{(L+R)\times T^b_{process}}{T^a_{process}+T^b_{process}} \right\rfloor, L\right) & \text{if } T_{transfer} \leq T^b_{process}, \\ 0 & \text{otherwise} \end{cases}$$

Firstly, if $T_{transfer} > T^b_{process}$, then there is no incentives to ship any work to the theft; it is optimal to process all remaining work locally. However, if that is not the case, then the optimization lies in splitting the work such that both site a and site b will ideally complete at the same time after the shipping.

Typically, the work stealing mechanism is activated when a worker site has completed its assigned workload so that it may share the some of the unprocessed partitions of its adjacent sites. Once a worker site begins stealing

due to the completion of its work, it will continue stealing until both its adjacent sites report that nothing is to be stolen.

Also, understand that worker sites may be joining at any point of time, including when the site to join is already merging or reducing; thus, this join should be also be seen as a form of stealing. Therefore, each join will also use the same stealing mechanism to negotiate the sub-ranges. Special cases are made when there are insufficient statistics or when the worker site has not received all the map output splits; during these cases, the sharing is done like the modified BATON as mentioned in Section 5.4.

5.5.6 Fault-Tolerance

As mentioned in Section 2.2, the popularity of using the MapReduce framework for Big Data analytics arises not just because of its simple programming model that happens to be able to express a wide array of distributed computation, its widespread usage is also due to its robustness in execution, particularly its tolerance in handling site failures; in processing Big Data workload, it is particularly important that partial failures should not halt the entire execution.

For robust execution, a data processing framework is generally interested in

- the structural integrity of the execution architecture,
- the availability of input data, and
- the persistence of processing.

Fortunately, the first two out the three aspects of fault-tolerance are already accounted for by just the design of the EMRE architecture. Firstly, the incorporation of the BATON overlay, together with the `ApplicationMaster` as the "virtual site", assures that in the occasion of site failures, the tree structure of the worker sites can persist by some logical rearrangement. The original design has been modified slightly in that the replacing site has to be one of adjacent sites of the failed site so that the rearrangement will not disrupt the distribution of the sub-ranges, thus ensuring minimal data migration. Secondly, MapReduce systems typically rely on the fault-tolerance of the underlying file system to assure data availability; if a distributed file system is used, this aspect of the fault-tolerance manifests as the replication of the sub-files, which is also what EMRE relies on to assure the availability of input data.

As for the persistence of processing, a more sophisticated recovery mechanism, as compared to the current execution architecture, has to be put in place. It can be seen that the adoption of the work-oriented mentality by the current execution architecture is mainly to cater for a simpler work recovery; this is why each task is relatively independent and is associated with a rather idempotent processing such that the processing can always be redone. Note that this is, by no means, trying to undermine the importance of simplicity in the design of systems. However, this thesis holds the position that the recovery mechanism implemented in EMRE, albeit slightly more involved than the current execution architecture, suffers no loss in the robustness of its execution.

The key of the recovery mechanism is to reconstruct the intermediate data generated and received by the failed worker site. Note that at any point

of time, there are four main types of data that the splits manager may be holding:

- map output from the mapper component,
- received map output splits,
- reduce input splits, and
- processed reduce input splits.

Fortunately, all these types of data can be reconstructed with minimal effort.

To begin with, when a particular worker site fails, its adjacent sites will have to cover its sub-ranges equally; this is after the restructuring of the BATON overlay. These adjacent sites will announce their updated sub-ranges via bounded broadcast. All the participating worker sites, upon the reception of such announcement, will locate in its own collection of map output (i.e., the ones kept for recovery purposes) and extract out the appropriate major-partitions to send to these adjacent sites. Since the adjacent sites simply experience an update in their sub-ranges, normal activity of the worker site is sufficient to assure the correct continuity of the processing. The `ApplicationMaster` will have had independently detected the failure of the worker site due to the timeout of the heartbeat messages, so it will proceed to set the map tasks associated with that failed worker site back to unassigned status so that other worker sites may redo the processing of these tasks. The worker sites will receive these failed map tasks and process them as per normal (i.e., applying map function, sort and distribute the resulting output); a slight optimization implemented is that the splits of the output from these map tasks will not be sent to the worker sites if they already

possess them. The robustness of such a recovery mechanism lies in the fact that other than the restructuring of the worker sites, all forms of activity remains practically the same. Notice that there is no requirement to request for more `Container`s, though it may be helpful if the `ApplicationMaster` does; in this case, special attention will be put on the joining of the newly allocated `Container` such that it will join in between the two adjacent sites of the failed worker site.

Now, taking into account potential site failures together with out-of-bound splits, it is possible that some map output splits are lost even with the above mentioned recovery mechanism because they were stored at the failed site while not being under its sub-range; note that reduce input splits will not suffer this problem because they can always be reconstructed from the map output splits. Therefore, an additional timeout is put in place for the case when a worker site persistently lacks some map output splits. On timeout, the worker site will announce its need for these splits via bounded broadcast. If the worker sites that originally processed those splits have not failed, they will respond to this exceptional request by extracting out the appropriate major-partitions from its collection of map output and send them to the requester directly. Even if in the situation where there is no site failure, such a special request mechanism can also act as a mean to optimize the throughput of the execution architecture (i.e., to minimize the number of occasions of having idling sites).

5.5.7 Optimizations

Not only the restructuring of the execution architecture under EMRE permits an enhanced manner of processing MapReduce jobs, it also exposes

several optimization opportunities that EMRE can exploit. In the following, some of the implemented optimization is discussed.

5.5.7.1 Buffer Reuse

Under the current execution architecture, a rather large memory buffer (i.e., 100 MB by default) is instantiated per map task to sort the intermediate map output; it is subsequently discarded after the completion of the task. For EMRE, there presents the opportunity for the worker sites to re-use this buffer for all the assigned map tasks. Moreover, this buffer is also useful in reducing the number of disk accesses in other aspects of the execution such as for data shipping and for merging. The same buffer can be reused in these areas because they are actually mutually exclusive.

5.5.7.2 Merging Specificity

In general, the merging of intermediate map output of MapReduce jobs should warrant additional attention because it is actually quite unlike traditional merging (e.g., the merging part of an external sort in a relational database system). In particular, the merging process for MapReduce has very large fan-in (i.e., proportional to input size) and relatively small individual split size (i.e., inversely proportional to the number of reducers). Should the intermediate map output splits be stored as different files under the local file system, a single pass within the merge process would be bogged down by excessive disk seeks (i.e., at least one per split); this is especially aggravated with the issue of intrinsic synchronization. Therefore, for EMRE, it makes more sense to store all the splits of the same sub-range within the same file and to load as much as possible with the reused memory buffer to merge so as to reduce disk seeks.

5.5.7.3 Pre-fetching

Other than excessive disk seeks, high rates of page cache miss while reading from disk is also problematic; this is an effect of the co-location of `Containers` within the same machine (i.e., same hard disk) and the fact that all worker sites will commence merging at about the same time. The result is that regardless of the replacement strategy, disk pages will be replaced rapidly and temporal locality is lost. Fortunately, under EMRE, we know the sequence of execution on the splits (i.e., the zig-zag order); this allows EMRE to implement a pre-fetching mechanism such that the next partition is pre-fetched while the current partition is being processed.

5.5.7.4 Restricted Concurrency

As an effort to reduce disk contention within a single machine in the light of the phenomenon of intrinsic synchronization, there is an intentional restriction on the concurrency of the processes within each worker site. Several distinct periodic and sequential activities can be identified:

- Polling for network I/O (with non-blocking socket)
- Checking for readiness of map output splits
  - If yes, perform merging and supply reduce input
- Adjusting splits to current sub-range
- Checking for completeness of reduce task
  - If yes, initiate work-stealing

It would be very elegant academically if all these activities run within a single thread but our experience shows that having a separate thread to handle network I/O would be more responsive to work-stealing activities at no cost to the local performance. Therefore, the EMRE splits management consists of two threads running periodically: one thread that is dedicated to the network I/O and another that handles the rest of the activities in sequence.

5.5.8 Discussion: Autonomy of Execution

Notice that, other than the periodic heartbeat messages, the worker sites do not communicate with the master site at all throughout the execution; this includes the sharing of work, and data location and allocation. In fact, the entire job execution does not require the intervention of the master site; the master site is only required to dispatch tasks upon announcement of availability through the heartbeat messages. By independently obeying a set of locally-scoped rules (e.g., execution order and work stealing mechanism), the summation of the efforts of the worker sites defines implicitly the global progression of the job execution. Such autonomy of execution is only possible because EMRE embeds a P2P overlay into its execution architecture resulting in the control element involved in a MapReduce job being essentially distributed.

5.6 Experimental Study

An EMRE prototype[2] have been implemented with Java 1.6. The version of Hadoop used is 0.23.1 configured with most of the default settings.

[2]Source code available at: https://bitbucket.org/xanec/projectemily

The experiments are conducted on a 65 machines cluster. Each machine is equipped with an Intel Xeon X3430 Quad Core CPU (2.4 Ghz), 8 GB memory, two 450 GB SCSI disks, and 1 Gbps Ethernet interface. The operating system is CentOS 5.6 (Linux 2.6.148).

As the cluster used is shared by other researchers, the amount of memory resources available to each `NodeManager` is purposely reduced from 8 GB to 4 GB so as to reduce possible external interference. Note that on the current execution architecture, the `ApplicationMaster` requires a 2 GB `Container` and each map or reduce task requires a 1 GB `Container`. For EMRE, a worker site requires a 1 GB `Container`.

The experimental data is obtained from a subset of the Purdue MapReduce Benchmarks Suite (PUMA) (Ahmad et al., 2012); the selected benchmarks are namely *Word-Count*, *Inverted-Index*, *Self-Join* and *Adjacency-List*. For the first two jobs, the data set used is 150 GB of web documents in text/xml format downloaded from `http://dumps.wikimedia.org/enwiki/` while *Self-Join* and *Adjacency-List* use 80 GB and 30 GB of synthetic data respectively.

For each benchmark, the same job definition is executed on the current execution architecture (i.e., a completely unmodified Hadoop) and on EMRE. In addition, the program is also run on different cluster sizes (i.e., 16, 32, 64 sites of `NodeManagers`). For the current execution architecture, different number of reducers have also been experimented with as indicated on the x-axis as the r-values; for example, when the cluster size is 16 and the r-value is 2, the corresponding number of reducers used is $16 \times 2 = 32$. The experiment is also repeated with a simulated dynamic environment to explore the elasticity of the architectures. This is achieved via a customized

`Scheduler` that increases linearly the amount of resources available at regular intervals (i.e., arbitrarily chosen to be 50s and 100s); for example, only 1 `NodeManager` will be available for the job at the beginning, after 1 interval (e.g., 50s), 2 `NodeManager`s will be available. This growth continues until the entire cluster is available or until the end of the job, whichever is earlier. The running times under different execution environments are labelled *static*, *interval=50s* and *interval=100s* where *static* indicates that all the `NodeManager`s are available immediately upon submission. For each experiment, there will be a total of 45 experimental runs (i.e., 3 execution environments $\times$ 3 cluster sizes $\times$ (4 r-values + 1 EMRE run)); the timing obtained for each experiment run is an average calculated from three distinct executions.

5.6.1 *Word-Count*

The *Word-Count* experiment is the "hello world" program of the MapReduce framework; this simple job counts the occurrences of each word in a large collection of documents. The input key is arbitrary (e.g., the line number) and the value is each line in the document. For each application of the map function, the line is separated into its words and emitted as $\langle word, 1 \rangle$ pairs. The reducer simply sums the values of each word.

Figure 5.5 shows the running times. Under the static environment, the current execution architecture performs the best when $r = 1$. Comparing to this best case, EMRE is able to provide an average of 21.76% reduction in the running times; this is largely due to the pipelining of executions. Under the dynamic environment, it is noticeable that there is no single r-value that provides the best running times across the different cluster size;

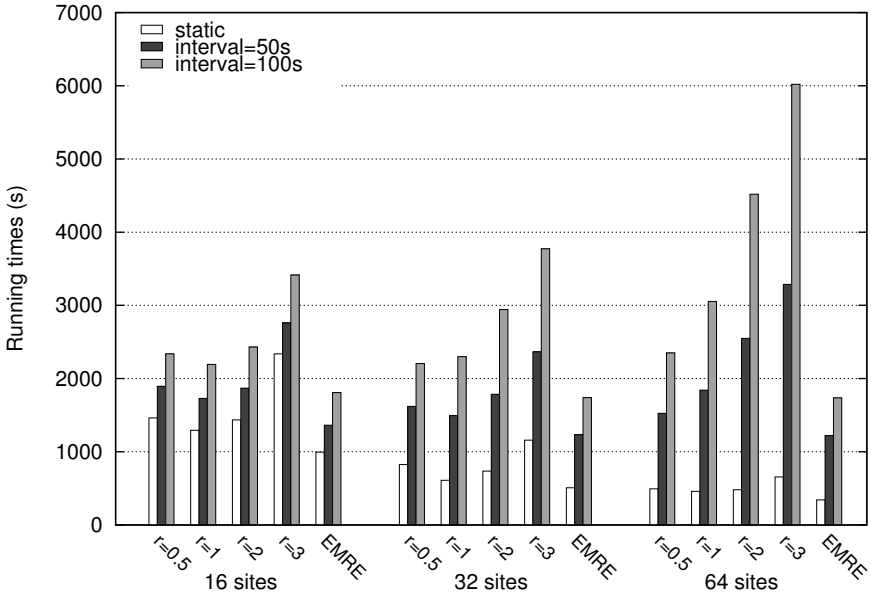


Figure 5.5: Running times for *Word-Count*

in particular, note that for 32 sites, when the interval is 50s, the optimal r-value is 0.5 but $r = 1$ is optimal when the interval is 100s. Regardless of the r-value, EMRE persistently provides the better running times. When the interval is 50s, EMRE provides, on the average, 23.96% to 53.77% reductions in running times depending on the r-values. When the interval is 100s, the reductions ranges from 23.27% to 57.35% depending on the r-values.

Note that execution under EMRE has as many reducers as there are `Containers` for the worker sites; therefore, it is natural to question how the current execution architecture performs with that many reducers. A further study is conducted on the current execution architecture on the effects of the number of reducers and the results are shown in Figure 5.6.

Firstly, the performance of having the maximum number of reducers is abysmal; this is because the architecture progressively degrades to having only one mapper while having many incomplete map tasks. Secondly, just as also evident in Figure 5.5, there is actually no "optimal" number of reducers which supports the argument that we actually cannot determine the best

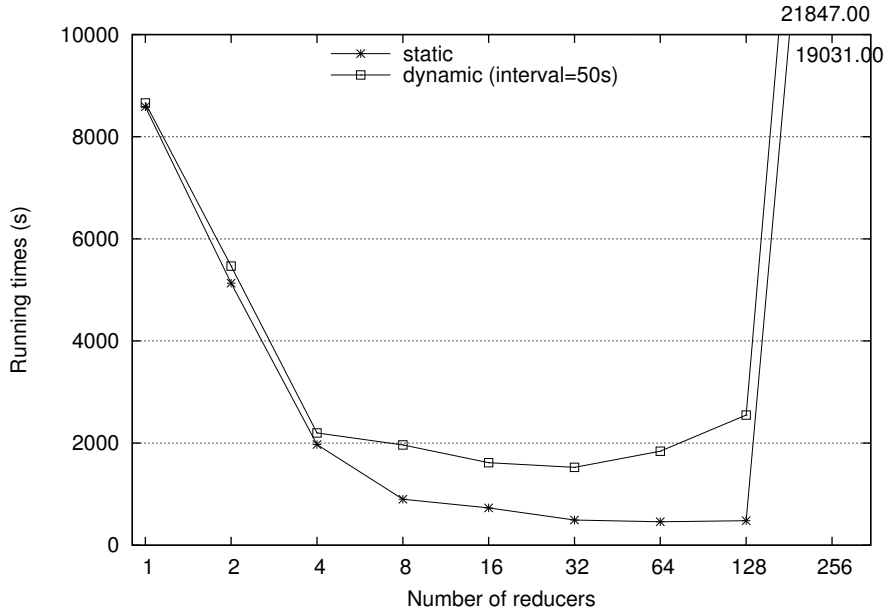


Figure 5.6: Effects of number of reducers for *Word-Count*

number of reducers to set because it is very sensitive to various external factors.

5.6.2 *Inverted-Index*

The *Inverted-Index* experiment takes a list of documents as input and generates word-to-document indexing. The input key is arbitrary (e.g., the line number) and the value is each line in the document. For each application of the map function, the line is separated into its words and emitted as $\langle word, docID \rangle$ pairs where $docID$ is the identity of the document (e.g., filename of the document). The reducer aggregates the $docID$s into a list for each word.

Figure 5.7 shows the running times. Due to the fact that the job profile (i.e., the processing load of the map and reduce tasks) of the *Inverted-Index* experiment is identical to that of the *Word-Count* experiment, their results are very similar and the causes leading to the results are the same

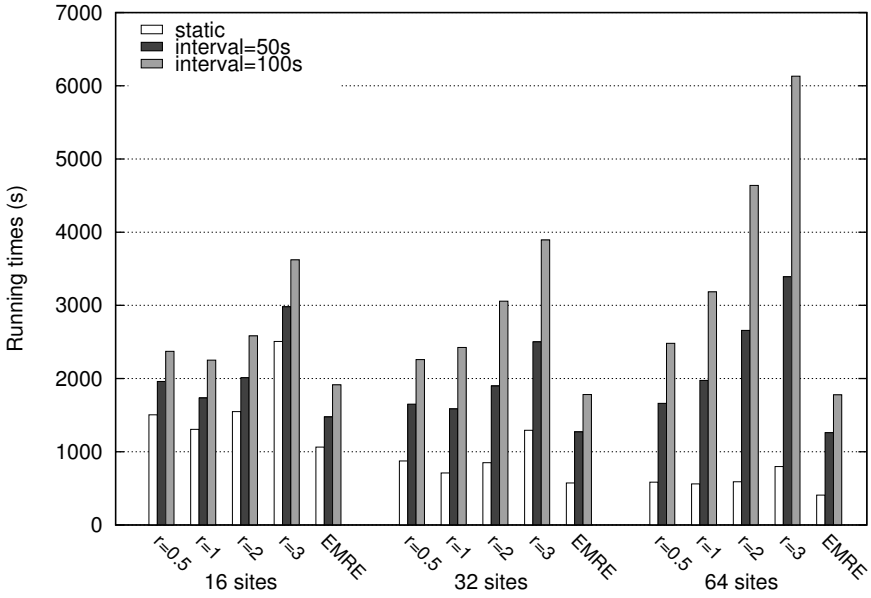


Figure 5.7: Running times for *Inverted-Index*

(refer to Section 5.6.1). Under the static environment, the current execution architecture performs the best when $r = 1$. Comparing to this best case, EMRE is able to provide an average of 21.62% reduction in running times. When the interval is 50s, EMRE provides, on the average, 23.58% to 54.14% reductions in running times depending on the r-values. When the interval is 100s, the reductions ranges from 22.96% to 57.47% depending on the r-values.

5.6.3 *Self-Join*

Despite its name, the *Self-Join* experiment is a syntactic job with not much significance in its output. The input key is arbitrary (i.e., the line number) and the value consists of k items tuple: $\{i_1, \ldots, i_k\}$. The output of the map function is the key/value pair of $\langle\{i_1, \ldots, i_{k-1}\}, i_k\rangle$. The reducer will emit consecutive pairs of the received values; for example for an input of $\langle\{i_1, \ldots, i_{k-1}\}, \{v_1, \ldots, v_n\}\rangle$, the reducer will output value pairs of (v_i, v_{i+1})

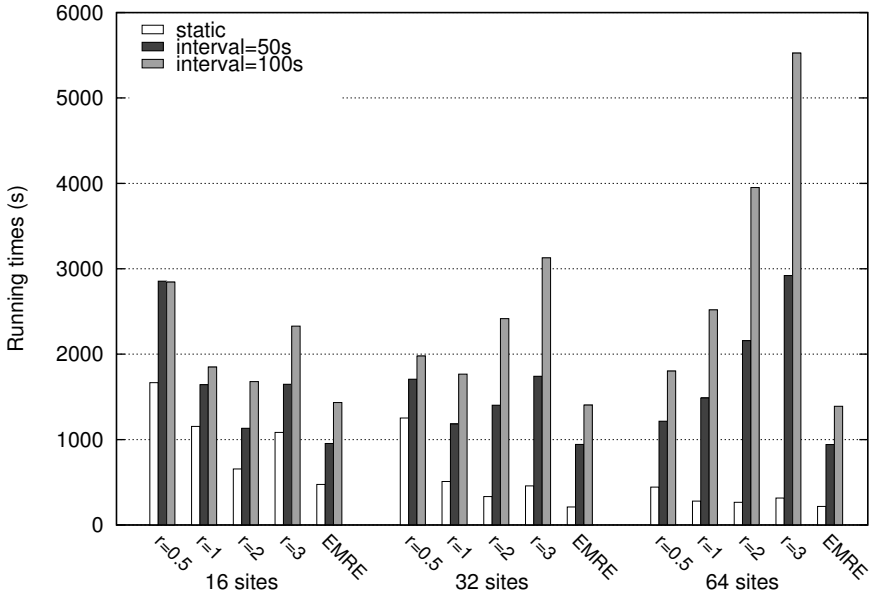


Figure 5.8: Running times of *Self-Join*

for $i = 1, \ldots, n-1$. The *Self-Join* experiment is said to be similar to the candidate generation step of the *Apriori* algorithm (Agrawal and Srikant, 1994).

Figure 5.8 shows the running times. Under the static environment, the current execution architecture actually performs best when $r = 2$. This is probably due to the significantly heavier reduce work that the time gained from having more reducers outweighs the penalty of having lesser mappers. Nevertheless, comparing to this best case, EMRE is able to provide an average of 27.71% reduction in running times. Under the dynamic environment, EMRE provides 32.90% to 51.87% reductions in running times when the interval is 50s and 29.24% to 56.11% reductions when it is 100s.

For the case of 16 sites when $r = 0.5$, it can be seen that the running times of the current architecture are about the same regardless of the interval. This is likely because there are so few reducers that the time taken to reduce dominates. It can also be seen this effect in Figure 5.6 for the case when the number of reducers are few. Given the experience of achieving optimality

when having few reducers under dynamic environment (e.g., 64 sites for all the benchmarks), one might be tempted to purposely restrict the number of reducers in expectation of a turbulent environment. However, the case of 16 sites and $r = 0.5$ in Figure 5.8 is a clear indication that one may run into the problem of being overzealous and become penalized instead.

Overall, EMRE provides even better improvement as compared to *Word-Count*. This phenomenon can be attributed to the fact that the reduce task of *Self-Join* is much more involved algorithmically (i.e., more processing per data size), therefore the work-stealing mechanism can bring about better work distribution. In the case of running in dynamic environment, the elasticity of EMRE helps tremendously because the current execution does not cater for resource augmentation during the reduce phase but EMRE does.

5.6.4 *Adjacency-List*

The *Adjacency-List* benchmark generates the adjacency and reverse-adjacency lists of vertices of a graph. The input key is arbitrary and the value is an edge of a directed graph. For each $\langle p, q \rangle$, the map function produces the vertex-to-dual-lists pair of $\langle p, (\{\}, \{q\}) \rangle$ and $\langle q, (\{p\}, \{\}) \rangle$ as output. The reducer will union two lists according to the key.

Figure 5.9 shows the running times. Under the static environment, the current execution architecture performs the best when $r = 2$, This is again due to the significantly heavier reduce work. Comparing to this best case, EMRE is able to provide an average of 21.54% reduction in running times.

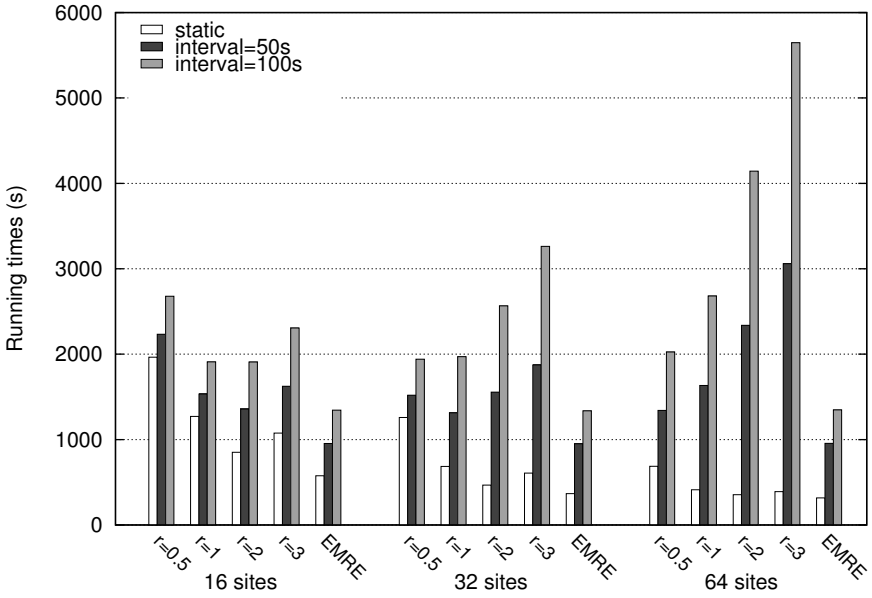


Figure 5.9: Running times for *Adjacency-List*

Under the dynamic environment, EMRE provides 35.64% to 53.08% reductions when the interval is 50s and 37.16% to 58.69% reductions when it is 100s.

Notice that the input data size used is relatively small (i.e., 30 GB) as compared to the previous two benchmarks while the running times remain comparable. This means that actually a large amount of intermediate data is generated for this benchmark resulting to much heavier reduce work than the previous two benchmarks. This difference accounts for even better improvement by EMRE as compared to *Self-Join*.

5.6.5 Discussion: Number of Reducers

As evident in the experimental results, the rule-of-thumb approach in determining the number of reducers does not work well; this is because the

number is rather sensitive to the work profile and the execution environment. Furthermore, now that YARN does not adopt the archetypal dedicated mappers or reducers slots, setting the wrong number of reducers can bring about much penalty.

The point to note is that under the current execution architecture, mappers last momentarily (i.e., about 45s) while reducers persist until the end of the job. Even though the assignment of reduce tasks is dependent on the map progress, the number set by the job submitter indicates the *eventual* proportion of the `Containers` "locked" by the reducers. Thus, the number of reducers becomes a trade-off between the parallelism exposed to the mappers and reducers. This is why a mapper-to-reducer ratio of $3:1$ works the best for some jobs while the equi-ratio of $1:1$ is better for others. When the execution environment is dynamic, the considerations become even more complex. Therefore, in our opinion, adopting EMRE provides precisely the solution to this problem.

5.7 Summary

Through the introduction of external resource managers into the MapReduce framework, resource scheduling is decoupled from the execution architecture. As a result of such decoupling, the execution architecture experiences dynamic resource allocation that is not unlike the typical environment immersing structured P2P overlays, albeit less harsh.

Therefore, EMRE seeks to exploit such externalization of resource managers by embedding a structured P2P overlay into the execution architecture in order to tap into the said dynamism. BATON is chosen for such a structured P2P overlay because of its independence from the cluster size and its

relatively limited inter-site dependencies. Note that due to the heavier cost of data shipping, the BATON overlay used for EMRE has been modified to assure that load sharing is as optimal as possible upon join.

With the worker sites deployed as a structured P2P overlay, work sharing can be done dynamically as and when sites join through the allocation of resources by the resource manager. Furthermore, staying true to the P2P philosophy, each worker site represents a single pipeline of the MapReduce execution that is identical from one another. Therefore, work sharing can be done at any point of the processing; elasticity is vastly increased as a result. A positive side-effect of such an approach is that the MapReduce execution architecture is transformed from a pull-based model to a push-based one; the result of such a transformation is improved performance in running times. On the whole, much of the control of the execution has been decentralized and shifted to the worker sites such that the master site presents mostly as a monitor; arguably, due to this offload of control on the master site, horizontal scalability has been extended.

EMRE demonstrates that even if the global architecture is not that of a structured P2P overlay, by incorporating a structured P2P overlay at some level, the system can benefit from some of the systemic qualities of the overlay. For the case of EMRE, the execution elasticity is vastly augmented due to the structural elasticity of the overlay. Furthermore, pipelining can be done because of the distributed index provided by the overlay.

Chapter 6

Conclusion

Of late, the database community has been experiencing unprecedented waves of paradigms-challenging trends. Even without taking sides in this war of conservatism versus progressivism conducted on the battleground of data mongering, by intersecting the domains of interests of these new trends, one can single out specific qualities that have catalyzed the blooming of the said trends. Three dimensions are highlighted in this thesis as the primary medium to view these novel developments: scalability, robustness and elasticity. It is now the mainstream consideration to adopt massive horizontal scaling as the mean to cope with overwhelming workload. While a web-scale distributed system has much to gain from horizontal scaling, it will be inhibited by its own size due paradoxical augmented probability of failures; therefore, the robustness of the system in the events of singly failures is now an indispensable factor to take into consideration. With expanded scale of computer clusters and multi-tenancy of processing jobs, elasticity of both the system *and* the processing has become a powerful architectural quality.

While the predominant architecture adopted by modern data processing systems is the simplistic master/workers architecture, this thesis pushes

the proposition that alternatives can be considered; particularly, structured P2P overlays are attractive candidates to be considered. The crux of this thesis is dedicated to the investigations, conducted or proposed, into various scenarios under the scalability-elasticity-robustness axes whereby structured P2P overlays can be demonstrated as a performance boon. The Katana framework demonstrates a manner in which a generalized programming model may be fused with the topology of the overlay such that expressiveness is augmented; with better expressiveness, the queries are executed closer to what is intended and thus provides much better running times. Hardened Katana is an extension to the original Katana framework that seeks to induced decentralized fault-tolerance into Katana's novel distributed programming model via a generalized fault-tolerance model called the COVER-CHARGE protocol (CCP). The Elastic MapReduce Execution (EMRE) shows that by embedding a structured P2P overlay into a master/workers architecture (i.e., YARN), an even more elastic execution can be adopted.

The presentation of this thesis is not the absolute extent of the works developed; there are many areas in which these works may be extended through further investigations:

- The Katana framework is designed to be built on a class of structured P2P overlays (i.e., CM-DHT) but as an introductory work, only the Chord-variant has explored; it will be interesting to investigate how other CM-DHTs fare in operation. In addition, with Katana as the foundation, other frameworks can be developed. Higher-level language that interprets into ANA and KATA jobs can be devised as a wrapper over the Katana framework for better user-side API. Other popular

modes of operation, such as iterative job execution and main-memory processing, may be included in future works.

- With the depiction by the CCP, formal model of operations may now be devised to study the processing in the framework in an abstract form. While being a fault-tolerant implementation, hardened Katana also highlights how the present Katana processing may have some oversights that require further study; in particular, job processing, especially under heterogeneous or load imbalanced environment, can be further improved with some optimization scheme.

- EMRE, on the other hand, has a lot of optimization opportunities that have yet to be explored. A way of achieving better load balancing, even in a heterogeneous environment, will be a beneficial addition to the execution architecture. An intelligent manner of deciding work load sharing, rather than a heuristic one, will deterministically reduce over-zealous work stealing. In addition, it will be interesting to see how EMRE fares when deployed in other resource management frameworks (e.g., Mesos and Corona).

While structured P2P overlays are not exactly extinct from the design of modern data processing systems, most works that utilize structured P2P overlays in one way or another have often neglected their individual characteristics and have failed to exploit them. This thesis holds the position that structured P2P overlays can and should have a bigger role in these changing times. The findings from the experimented works, as well as the proposed future developments seek to evince the feasibility of using structured P2P overlays in modern data processing systems and the benefits that these architectures can bring.

Bibliography

Aggarwal, C. C., N. Ashish, and A. Sheth (2013). The Internet of Things: A Survey from the Data-Centric Perspective. In C. C. Aggarwal (Ed.), *Managing and Mining Sensor Data*, pp. 383–428. Springer US. 3

Agrawal, R. and R. Srikant (1994). Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 487–499. Morgan Kaufmann Publishers Inc. 172

Ahmad, F., S. Lee, M. Thottethodi, and T. N. Vijaykumar (2012). PUMA: Purdue MapReduce Benchmarks Suite. Technical Report TR-ECE-12-11, Purdue University School of Electrical and Computer Engineering. 167

Ali-Eldin, A., M. Kihl, J. Tordsson, and E. Elmroth (2012). Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control. In *ScienceCloud '12: Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, New York, NY, USA, pp. 31–40. ACM. 20

Ananthanarayanan, G., A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica (2012). PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI'12: Proceedings of the 9th USENIX conference*

on Networked Systems Design and Implementation, Berkeley, CA, USA, pp. 20–20. USENIX Association. 16

Anderson, E. and J. Tucek (2010, March). Efficiency Matters! *SIGOPS Operating Systems Review 44*(1), 40–45. 14

Androutsellis-Theotokis, S. and D. Spinellis (2004, December). A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys 36*, 335–371. 23, 24

Apache (2012, February). Apache Hadoop NextGen MapReduce (YARN). Accessed on April 2014, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. 22, 27, 47

Appuswamy, R., C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron (2013, January). Nobody ever got fired for buying a cluster. Technical Report MSR-TR-2013-2, Microsoft Research. 14, 16

Atzori, L., A. Iera, and G. Morabito (2010, October). The Internet of Things: A survey. *Computer Networks: The International Journal of Computer and Telecommunications Networking 54*(15), 2787–2805. 3

Aviziens, A. (1976). Fault-Tolerant Systems. *IEEE Transactions on Computers C-25*(12), 1304–1312. 19

Battré, D., S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke (2010). Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SOCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, New York, NY, USA, pp. 119–130. ACM. 52

Bernstein, P. A. and N. Goodman (1984, December). An Algorithm for

Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems 9*(4), 596–615. 12

Bias, R. (2010, November). Elasticity is NOT #Cloud Computing ... Just Ask Google. Accessed on April 2014, http://cloudscaling.com/blog/cloud-computing/elasticity-is-not-cloud-computing-just-ask-google/. 20

Bird, R. S. and L. Meertens (1998, jun). Nested Datatypes. In J. Jeuring (Ed.), *LNCS 1422: Proceedings of Mathematics of Program Construction*, Marstrand, Sweden, pp. 52–67. Springer-Verlag. 64

Blanas, S., J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian (2010). A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, New York, NY, USA, pp. 975–986. ACM. 63, 89

Borthakur, D. (2010, May). Facebook has the world's largest Hadoop cluster! Accessed on April 2014, http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html. 3

Bosilca, G., R. Delmas, J. Dongarra, and J. Langou (2009, April). Algorithm-based Fault Tolerance Applied to High Performance Computing. *Journal of Parallel and Distributed Computing 69*(4), 410–416. 101

Bougeret, M., H. Casanova, M. Rabie, Y. Robert, and F. Vivien (2011). Checkpointing Strategies for Parallel Jobs. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, pp. 33:1–33:11. ACM. 101

Brebner, P. C. (2012). Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applica-

tions. In *ICPE '12: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, New York, NY, USA, pp. 263–266. ACM. 20

Brewer, E. (2012). CAP Twelve Years Later: How the "Rules" Have Changed. *Computer 45*(2), 23–29. 2, 10

Brewer, E. A. (2000). Towards Robust Distributed Systems. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, pp. 7–. ACM. 2

Cappello, F., A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir (2009, November). Toward Exascale Resilience. *International Journal of High Performance Computing Applications 23*(4), 374–388. 101

Cattell, R. (2011, May). Scalable SQL and NoSQL Data Stores. *SIGMOD Record 39*, 12–27. 2, 10

Chen, J., C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears (2012). Walnut: A Unified Cloud Object Store. In *SIGMOD '12: Proceedings of the 2012 international conference on Management of Data*, New York, NY, USA, pp. 743–754. ACM. 21

Ching, A., R. Murthy, D. Molkov, R. Vadali, and P. Yang (2012, November). Under the Hood: Scheduling MapReduce jobs more efficiently with Corona. Accessed on April 2014, https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920. 27, 47, 52

Codd, E. F. (1983, January). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM 26*(1), 64–69. 5

Cohen, J., B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton (2009, August). MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment 2*(2), 1481–1492. 6

Das, S., D. Agrawal, and A. El Abbadi (2013, April). ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *ACM Transactions on Database Systems 38*(1), 5:1–5:45. 21

Dean, J. and S. Ghemawat (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*(1), 107–113. 13, 21

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels (2007). Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review 41*(6), 205–220. 12, 24, 113

Devlin, B. (2011, November). The Seven Faces of Data - Rethinking data's basic characteristics. Technical report, 9sight Consulting. 1

Dijkstra, E. W. (1982). *Selected Writings on Computing: A Personal Perspective*, Chapter EWD447: On the Role of Scientific Thought, pp. 60–66. Springer-Verlag New York, Inc. 8

Dinu, F. and T. E. Ng (2012). Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *HPDC '12: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, New York, NY, USA, pp. 187–198. ACM. 19, 122, 123

Douglas, L. (2012, June). The Importance of 'Big Data': A Definition. Technical report, Gartner, Inc Survey. 6

Doulkeridis, C. and K. Nørvåg (2014, June). A Survey of Large-Scale Analytical Query Processing in MapReduce. *The VLDB Journal 23*(3), 355–380. 101

Duan, Q., Y. Yan, and A. Vasilakos (2012, December). A Survey on Service-Oriented Network Virtualization Toward Convergence of Networking and Cloud Computing. *IEEE Transactions on Network and Service Management 9*(4), 373–392. 10

Elnozahy, E. N. M., L. Alvisi, Y.-M. Wang, and D. B. Johnson (2002, September). A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys 34*(3), 375–408. 101

Fang, Z. (2012). Storage Space Market Brief. Technical Report 12, IHS iSuppli. 5

Fardone, G. (2012, May). Cloud elasticity and cloud scalability are not the same thing. Accessed on April 2014, http://blog.evolveip.net/index.php/2012/05/24/cloud-elasticity-and-cloud-scalability-are-not-the-same-thing-2/. 20

Fernando, N., S. W. Loke, and W. Rahayu (2013, January). Mobile cloud computing: A survey. *Future Generation Computer Systems 29*(1), 84–106. 3

Ferreira, K., J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold (2011). Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, pp. 44:1–44:12. ACM. 101

Fox, A., S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier (1997). Cluster-Based Scalable Network Services. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 78–91. ACM. 2

Friedman, E., P. Pawlowski, and J. Cieslewicz (2009). SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment 2*(2), 1402–1413. 43, 52

Gantz, J. and D. Reinsel (2012, December). The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Technical report, International Data Corporation. 1

Gantz, J., D. Reinsel, C. Chute, W. Schlichting, J. McArthur, M. Stephen, X. Irida, A. Toncheva, and A. Manfrediz (2007, March). The Expanding Digital Universe: A Forecast of Worldwide Information Growth Through 2010. Technical report, International Data Corporation. 1

Garfinkel, S. (1999). *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. MIT Press. 8

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). The Google File System. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 29–43. ACM. 41, 43

Gilbert, S. and N. Lynch (2002, June). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News 33*, 51–59. 2

Goh, W. X. and K.-L. Tan (2013). Katana: Generalized Data Processing on Peer-to-Peer Overlays. In *IC2E '13: Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, Washington, DC, USA, pp. 318–327. IEEE Computer Society. 25, 78, 85

Goh, W. X. and K.-L. Tan (2014). Elastic MapReduce Execution. In *CCGrid '12: Proceedings of the 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Washington, DC, USA. IEEE Computer Society. 25

Gray, J., P. Helland, P. O'Neil, and D. Shasha (1996). The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, New York, NY, USA, pp. 173–182. ACM. 12

Gustafson, J. L. (1988, May). Reevaluating Amdahl's law. *Communications of the ACM 31*, 532–533. 43

Hecht, R. and S. Jablonski (2011). NoSQL evaluation: A use case oriented survey. In *CSC '11: Proceedings of the 2011 International Conference on Cloud and Service Computing*, Washington, DC, USA, pp. 336–341. IEEE Computer Society. 11

Heintz, B., C. Wang, A. Chandra, and J. Weissman (2013). Cross-Phase Optimization in MapReduce. In *IC2E '13: Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, Washington, DC, USA, pp. 338–347. IEEE Computer Society. 44

Herodotou, H., H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu (2011). Starfish: A Self-tuning System for Big Data Analytics. In *CIDR '11: Proceedings of the fifth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, pp. 261–272. 15

Hindman, B., A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI '11: Proceedings of the 8th USENIX conference on Networked systems design and implementation*, Berkeley, CA, USA, pp. 22–22. USENIX Association. 22, 28, 47, 52

Hunt, P., M. Konar, F. P. Junqueira, and B. Reed (2010). ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, Berkeley, CA, USA, pp. 11–11. USENIX Association. 98

Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming 9*(4), 355–372. 65

Isard, M., M. Budiu, Y. Yu, A. Birrell, and D. Fetterly (2007, March). Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Operating Systems Review 41*, 59–72. 21, 100

Jagadish, H. V., B. C. Ooi, and Q. H. Vu (2005). BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pp. 661–672. VLDB Endowment. 29, 38, 134

Jiang, D., B. C. Ooi, L. Shi, and S. Wu (2010, September). The Performance of MapReduce: An In-depth Study. *Proceedings of the VLDB Endowment 3*, 472–483. 14, 20

Jiang, D., A. K. H. Tung, and G. Chen (2011, September). MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters. *IEEE Transactions on Knowledge and Data Engineering 23*(9), 1299–1311. 51

Johnson, P. R. and R. Thomas (1975, January). Maintenance of Duplicate Databases. RFC 677, RFC Editor. 2

Kachris, C. and I. Tomkos (2012, Fourth). A Survey on Optical Interconnects for Data Centers. *IEEE Communications Surveys Tutorials 14*(4), 1021–1036. 10

Kallman, R., H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi (2008, August). H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment 1*(2), 1496–1499. 24

Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin (1997). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, New York, NY, USA, pp. 654–663. ACM. 32, 34

King, I., J. Li, and K. T. Chan (2009). A Brief Survey of Computational Approaches in Social Computing. In *IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks*, Piscataway, NJ, USA, pp. 2699–2706. IEEE Press. 3

Konstantinou, I., E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris (2011). On the Elasticity of NoSQL Databases over Cloud Management Platforms. In *CIKM '11: Proceedings of the 20th ACM international conference on Information and knowledge management*, New York, NY, USA, pp. 2385–2388. ACM. 21

Lämmel, R. (2007). Google's MapReduce Programming Model — Revisited. *Science of Computer Programming 68*(3), 208–237. 43, 64

Lee, K., T. W. Choi, A. Ganguly, D. I. Wolinsky, O. Boykin, and R. Figueiredo (2011). Parallel Processing Framework on a P2P System Using Map and Reduce Primitives. In *In the 8th International Workshop on Hot Topics in Peer-to-Peer Systems in Conjunction with IPDPS 2011*. 14, 35

Lee, K.-H., Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon (2012). Parallel Data Processing with MapReduce: A Survey. *SIGMOD Record 40*(4), 11–20. 101

Leong, B., B. Liskov, and E. D. Demaine (2006, May). EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. *Computer Communications 29*, 1243–1259. 35

Li, F., B. C. Ooi, M. T. Özsu, and S. Wu (2014, January). Distributed Data Management Using MapReduce. *ACM Computing Surveys 46*(3), 31:1–31:42. 101

Lupu, M., B. C. Ooi, and Y. C. Tay (2008). Paths to Stardom: Calibrating the Potential of a Peer-based Data Management System. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 265–278. ACM. 39, 53, 68, 69, 136

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2010). Pregel: A System for Large-Scale Graph Processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, New York, NY, USA, pp. 135–146. ACM. 100

Malkhi, D., M. Naor, and D. Ratajczak (2002). Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *PODC '02: Proceedings of the*

twenty-first annual symposium on Principles of distributed computing, New York, NY, USA, pp. 183–192. ACM. 69

Manohar, N. (2013). A Survey of Virtualization Techniques in Cloud Computing. In V. S. Chakravarthi, Y. J. M. Shirur, and R. Prasad (Eds.), *Proceedings of International Conference on VLSI, Communication, Advanced Devices, Signals & Systems and Networking (VCASAN-2013)*, Volume 258 of *Lecture Notes in Electrical Engineering*, pp. 461–470. Springer India. 9

McKendrick, J. (2012, September). Big Data, Big Challenges, Big Opportunities: 2012 IOUG Big Data Strategies Survey. Technical report, Unisphere Research, A Division of Information Today, Inc. 1, 3, 4

Meijer, E., M. Fokkinga, and R. Paterson (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA '91: Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, New York, NY, USA, pp. 124–144. Springer-Verlag New York, Inc. 53

Mohan, C. (2013). History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *EDBT '13: Proceedings of the 16th International Conference on Extending Database Technology*, New York, NY, USA, pp. 11–16. ACM. 11

Monash, C. (2011, July). Petabyte-scale Hadoop clusters (dozens of them). Accessed on April 2014, http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/. 3, 52

Mundkur, P., V. Tuulos, and J. Flatow (2011). Disco: A Computing Platform for Large-Scale Data Analytics. In *SIGPLAN '11: Proceedings of*

the 10th ACM SIGPLAN workshop on Erlang, Erlang '11, New York, NY, USA, pp. 84–89. ACM. 41

Murthy, A. C., C. Douglas, M. Konar, O. O'Malley, S. Radia, S. Agarwal, and K. V. Vinod (2011). Architecture of Next Generation Apache Hadoop MapReduce Framework. Technical report, Apache Hadoop. 52

Myers, A. (2012, March). High Availability for the Hadoop Distributed File System (HDFS). Accessed on April 2014, https://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/. 22, 46, 52, 98

Nadkarni, A. and L. DuBois (2013, April). Storage for Big Data: Insight into Usage Patterns. Technical Report 240372, International Data Corporation. 5

Olston, C., B. R Reed, U. Srivastava, R. Kumar, and A. Tomkins (2008). Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 1099–1110. ACM. 43, 52

Ordonez, C., I.-Y. Song, and C. Garcia-Alvarado (2010). Relational versus Non-Relational Database Systems for Data Warehousing. In *DOLAP '10: Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*, New York, NY, USA, pp. 67–68. ACM. 11

Özsu, M. T. and P. Valduriez (1999). *Principles of Distributed Database Systems*. Prentice Hill. 12

Park, G., S. Kim, Y. Cho, J. Kook, and J. Hong (2010). Chordet: An Efficient and Transparent Replication for Improving Availability of Peer-to-Peer Networked Systems. In *SAC '10: Proceedings of the 2010 ACM*

Symposium on Applied Computing, New York, NY, USA, pp. 221–225. ACM. 35

Pavlo, A., E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker (2009). A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, New York, NY, USA, pp. 165–178. ACM. 13, 14, 43

Pike, R., S. Dorward, R. Griesemer, and S. Quinlan (2005). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*(4), 277–298. 43, 52

Rasti, A., D. Stutzbach, and R. Rejaie (2006). On the Long-term Evolution of the Two-Tier Gnutella Overlay. In *INFOCOM 2006: Proceedings of the 25th IEEE International Conference on Computer Communications*, pp. 1–6. 23

Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Shenker (2001). A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, pp. 161–172. ACM. 23, 24, 34, 50, 113, 135

Rowstron, A., D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas (2012). Nobody ever got fired for using Hadoop on a cluster. In *HotCDP '12: Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, New York, NY, USA, pp. 2:1–2:5. ACM. 14, 16

Rowstron, A. I. T. and P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In

Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, London, UK, pp. 329–350. Springer-Verlag. 23, 34

Ryan, A. (2012, June). Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode. Accessed on April 2014, https://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920. 98

Saito, Y. and M. Shapiro (2005, March). Optimistic Replication. *ACM Computing Surveys 37*(1), 42–81. 2

Sakr, S., A. Liu, D. Batista, and M. Alomari (2011, March). A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys Tutorials 13*(3), 311–336. 8, 9

Sakr, S., A. Liu, and A. G. Fayoumi (2013, July). The Family of MapReduce and Large-Scale Data Processing Systems. *ACM Computing Surveys 46*(1), 11:1–11:44. 13, 14, 101

Sánchez-Artigas, M. and P. García López (2010, February). Echo: A peer-to-peer clustering framework for improving communication in DHTs. *Journal of Parallel and Distributed Computing 70*, 126–143. 35

Shen, H., C.-Z. Xu, and G. Chen (2006, March). Cycloid: A Constant-Degree and Lookup-Efficient P2P Overlay Network. *Performance Evaluation 63*, 195–216. 69

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 conference on*

Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, pp. 149–160. ACM. 23, 24, 29, 34, 113, 135

Stonebraker, M., S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland (2007). The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pp. 1150–1160. VLDB Endowment. 11

Suleiman, B., S. Sakr, R. Jeffery, and A. Liu (2012). On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Journal of Internet Services and Applications 3*(2), 173–193. 20

Tao, L. (2001, October). Shifting Paradigms with the Application Service Provider Model. *Computer 34*(10), 32–39. 8

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy (2009, August). Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment 2*, 1626–1629. 43, 52

Vishnevsky, V., A. Safonov, M. Yakimov, E. Shim, and A. D. Gelman (2008, February). Scalable blind search and broadcasting over Distributed Hash Tables. *Computer Communications 31*, 292–303. 40

Weibull, W. (1951). A statistical distribution function of wide applicability. *Journal of Applied Mechanics 18*, 293–297. 18

Wong, J. (2013, January). Which Big Data Company has the World's Biggest Hadoop Cluster? Accessed on April

2014, http://www.hadoopwizard.com/which-big-data-company-has-the-worlds-biggest-hadoop-cluster/. 3

Yang, C., C. Yen, C. Tan, and S. Madden (2010). Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE '10: Proceedings of the 26th International Conference on Data Engineering*, pp. 657–668. 14, 100

Yang, H.-C., A. Dasdan, R.-L. Hsiao, and D. S. Parker (2007). Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 1029–1040. ACM. 51

Zhao, B. Y., L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz (2004). Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications 22*(1), 41–53. 23, 34

Appendix A

Group Theory

Definition A.1: Group

A *binary operation* on a set G is a function $\oplus$ that maps each element $(a, b) \in G \times G$ to an unique element $(a \oplus b) \in G$, or $\oplus^a : G \times G \to G$. A *group* is a set G together with a binary operation that satisfies the following axioms:

- The binary operation is *associative* such that

$$\forall a, b, c \in G,\ (a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{A.1}$$

- There exists an *identity element* $e \in G$ such that

$$\forall a \in G,\ e \oplus a = a \oplus e = a \tag{A.2}$$

- For each element $a \in G$, there exists an *inverse element* $a^{-1} \in G$

such that

$$a \oplus a^{-1} = a^{-1} \oplus a = e \tag{A.3}$$

[a] For the ease of reading, an *infix* notation is used to depict the binary operation.

Remark A.2

Trivially, the group is *finite* if its corresponding set G has finite elements; otherwise it is *infinite*. In our discussion, we shall primarily deal with finite groups because of its practical implication in reality. A group $(G, \oplus)$ is called *abelian* if $\oplus$ is commutative; that is to say $\forall a, b \in G,\ a \oplus b = b \oplus a$. In addition, a group $(H, \oplus)$ is called a *subgroup* of another group $(G, \oplus)$ if H is a subset of G and both groups share the same binary operation $\oplus$.

Definition A.3: Integer Group

An *integer group* $(\mathbb{Z}_n, +_n)$ is a group with $\mathbb{Z}_n \triangleq \{\overline{0}_n, \overline{1}_n, \overline{2}_n, \ldots, \overline{(n-1)}_n\}$ as its set where $\forall a \in \mathbb{Z}$, $\overline{a}_n$ is the *congruence class* of a *modulo* n, that is

$$\forall a \in \mathbb{Z},\ \overline{a}_n = \{\ldots, a - 2n, a - n, a, a + n, a + 2n, \ldots\} \tag{A.4}$$

and the modular addition $+_n$ as its binary operation, where

$$\forall \overline{a}_n, \overline{b}_n \in \mathbb{Z}_n,\ (\overline{a}_n +_n \overline{b}_n) = \overline{(a+b)}_n \tag{A.5}$$

Proof. Firstly, $\forall a, b \in \mathbb{Z}$, $\overline{(a+b)}_n \in \mathbb{Z}_n$ by definition, therefore $+_n$ is a binary operation (i.e., $+_n : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$).

Associativity. Given $\overline{a}_n, \overline{b}_n, \overline{c}_n \in \mathbb{Z}_n$,

$$\begin{aligned}
(\overline{a}_n +_n \overline{b}_n) +_n \overline{c}_n &= \overline{(a+b)}_n +_n \overline{c}_n && \text{by (A.5)} \\
&= \overline{\big((a+b)+c\big)}_n && \text{by (A.5)} \\
&= \overline{\big(a+(b+c)\big)}_n && \text{by (A.1)} \\
&= \overline{a}_n +_n \overline{(b+c)}_n && \text{by (A.5)} \\
&= \overline{a}_n +_n (\overline{b}_n +_n \overline{c}_n) && \text{by (A.5)}
\end{aligned}$$

Identity. $\overline{0}_n$ is the identity element in $(\mathbb{Z}_n, +_n)$; $\forall \overline{a}_n \in \mathbb{Z}_n$,

$$\begin{aligned}
\overline{0}_n +_n \overline{a}_n &= \overline{(0+a)}_n && \text{by (A.5)} \\
&= \overline{a}_n && \text{by (A.4)} \\
&= \overline{(a+0)}_n && \text{by (A.4)} \\
&= \overline{a}_n +_n \overline{0}_n && \text{by (A.5)}
\end{aligned}$$

Inverse. $\forall \overline{a}_n \in \mathbb{Z}_n$, $(\overline{a}_n)^{-1} = \overline{(-a)}_n$; by definition, $\forall k \in \mathbb{Z}$, $\overline{k}_n \in \mathbb{Z}_n$, therefore, $\forall \overline{a}_n \in \mathbb{Z}_n$, $\exists \overline{(-a)}_n \in \mathbb{Z}_n$ such that

$$\begin{aligned}
\overline{a}_n +_n \overline{(-a)}_n &= \overline{\big(a+(-a)\big)}_n && \text{by (A.5)} \\
&= \overline{0}_n && \text{by inverse of } \mathbb{Z} \\
&= \overline{\big((-a)+a\big)}_n && \text{by inverse of } \mathbb{Z} \\
&= \overline{(-a)}_n +_n \overline{a}_n && \text{by (A.5)}
\end{aligned}$$

Therefore, $(\mathbb{Z}_n, +_n)$ is a group. □

Definition A.4: Symmetric Group

Given a set S, a bijection $\pi : S \to S$ is called a *permutation* on S. Suppose S contains n elements, let S_n denote the set of all the permutations on S, $(S_n, \circ)$ is a group called the *symmetric group* where $\circ$ is function composition:

$$\forall \pi_1, \pi_2 \in S_n,\ \forall s \in S,\ (\pi_1 \circ \pi_2)(s) = \pi_1(\pi_2(s)) \tag{A.6}$$

Proof. Note that a composition of bijective functions is bijective, therefore, $\circ$ is a binary operation (i.e., $\circ : S_n \times S_n \to S_n$). Furthermore, function composition is, by definition, associative. The identity element is the identity function $i : S \to S$, where $\forall s \in S$, $i(s) = s$. Given $\pi \in S_n$, the corresponding inverse element is trivially the inverse function π^{-1}, which is well-defined since π is bijective (i.e., its inverse exists) and π^{-1} is also bijective (i.e., $\pi^{-1} \in S_n$). Therefore, $(S_n, \circ)$ is a group. □

Definition A.5: Permutation Group

A subgroup of a symmetric group $(S_n, \circ)$ is called a *permutation group*.

Definition A.6: Direct Product

Given two groups $(G, \oplus)$ and $(H, \otimes)$, the direct product of $(G, \oplus)$ and $(H, \otimes)$ is a group with $G \times H$ as its set and $\odot$ as the binary operation, where $\odot$ is defined as follows:

$$\forall g_1, g_2 \in G, h_1, h_2 \in H,\ (g_1, h_1) \odot (g_2, h_2) \triangleq (g_1 \oplus g_2, h_1 \otimes h_2) \tag{A.7}$$

Proof. Since

$$\oplus : G \times G \to G \text{ and } \otimes : H \times H \to H$$

then

$$\odot : (G \times H) \times (G \times H) \to (G \times H)$$

Associativity. Given $g_1, g_2, g_3 \in G$, $h_1, h_2, h_3 \in H$,

$$\begin{aligned}
&\big((g_1, h_1) \odot (g_2, h_2)\big) \odot (g_3, h_3) \\
&= (g_1 \oplus g_2, h_1 \otimes h_2) \odot (g_3, h_3) && \text{by (A.7)} \\
&= \big((g_1 \oplus g_2) \oplus g_3, (h_1 \otimes h_2) \otimes h_3\big) && \text{by (A.7)} \\
&= \big(g_1 \oplus (g_2 \oplus g_3), h_1 \otimes (h_2 \otimes h_3)\big) && \text{by (A.1)} \\
&= (g_1, h_1) \odot (g_2 \oplus g_3, h_2 \otimes h_3) && \text{by (A.7)} \\
&= (g_1, h_1) \odot \big((g_2, h_2) \odot (g_3, h_3)\big) && \text{by (A.7)}
\end{aligned}$$

Identity. Given e_G identity element of $(G, \oplus)$ and e_H identity element of $(H, \otimes)$, (e_G, e_H) is the identity element of the direct product of $(G, \oplus)$ and $(H, \otimes)$. $\forall g \in G, h \in H$,

$$\begin{aligned}
(e_G, e_H) \odot (g, h) &= (e_G \oplus g, e_H \otimes h) && \text{by (A.7)} \\
&= (g, h) && \text{by (A.2)} \\
&= (g \oplus e_G, h \oplus e_H) && \text{by (A.2)} \\
&= (g, h) \odot (e_G, e_H) && \text{by (A.7)}
\end{aligned}$$

Inverse. Given $(g, h) \in G \times H$, suppose g^{-1} is the inverse of g in $(G, \oplus)$ and h^{-1} is the inverse of h in $(H, \otimes)$, (g^{-1}, h^{-1}) is the inverse of

(g, h) in $(G \times H, \odot)$.

$$\begin{aligned}
(g, h) \odot (g^{-1}, h^{-1}) &= (g \oplus g^{-1}, h \otimes h^{-1}) && \text{by (A.7)} \\
&= (e_G, e_H) && \text{by (A.3)} \\
&= (g^{-1} \oplus g, h^{-1} \otimes h) && \text{by (A.3)} \\
&= (g^{-1}, h^{-1}) \odot (g, h) && \text{by (A.7)}
\end{aligned}$$

Therefore, $(G \times H, \odot)$ is a group. □

Definition A.7: Homomorphism

In general, *homomorphism* is a "structure-preserving" function that maps from one algebraic structure to another; the exact definition of the preservation in question depends on the axioms of the algebraic structure in question.

A *group homomorphism* mapping from a group $(G, \oplus)$ to another group $(H, \otimes)$ is a function $h : G \to H$ such that

$$\forall a, b \in G,\ h(a \oplus b) = h(a) \otimes h(b) \tag{A.8}$$

A *graph homomorphism* mapping from a graph $G = (V, E)$ to another graph $G' = (V', E')$ is a function $h : V \to V'$ such that

$$\forall a, b \in V,\ (a, b) \in E \Rightarrow \big(h(a), h(b)\big) \in E' \tag{A.9}$$

The algebraic prefix (i.e., "group" and "graph") will be dropped and the function will be denoted as just homomorphism when it is clear which

kind of algebraic structure is referred to.

A homomorphism that is *bijective* is called an *isomorphism*. An isomorphism whose domain and codomain are equivalent is called an *automorphism*.

Theorem A.1: Cayley's Theorem

Every group is isomorphic to some permutation group.

Proof. Given a group $(G, \oplus)$, for any $g \in G$, define a function $\lambda_g : G \to G$ such that $\forall a \in G$, $\lambda_g(a) = g \oplus a$. λ_g is a bijective, hence permutation, on G:

Injective. $\forall a, b \in G$, suppose $\lambda_g(a) = \lambda_g(b)$,

$$\begin{aligned}
\lambda_g(a) = \lambda_g(b) &\Rightarrow g \oplus a = g \oplus b && \text{by def}^{\text{n}} \text{ of } \lambda \\
&\Rightarrow g^{-1} \oplus (g \oplus a) = g^{-1} \oplus (g \oplus b) \\
&\Rightarrow (g^{-1} \oplus g) \oplus a = (g^{-1} \oplus g) \oplus b && \text{by (A.1)} \\
&\Rightarrow e \oplus a = e \oplus b && \text{by (A.3)} \\
&\Rightarrow a = b && \text{by (A.2)}
\end{aligned}$$

Surjective. $\forall a \in G$, define $b = g^{-1} \oplus a$. Therefore,

$$\begin{aligned}
\lambda_g(b) &= g \oplus b && \text{by def}^{\text{n}} \text{ of } \lambda \\
&= g \oplus (g^{-1} \oplus a) && \text{by def}^{\text{n}} \text{ of } b \\
&= (g \oplus g^{-1}) \oplus a && \text{by (A.1)} \\
&= e \oplus a && \text{by (A.3)}
\end{aligned}$$

$$= a \qquad \text{by (A.2)}$$

Therefore, the set $\overline{G} \triangleq \{\lambda_g \mid g \in G\}$ forms a group together with the function composition: $(\overline{G}, \circ)$.

Define the function $\phi : G \to \overline{G}$ such that $\forall g \in G$, $\phi(g) = \lambda_g$. ϕ is a homomorphism from $(G, \oplus)$ to $(\overline{G}, \circ)$; $\forall a, b \in \overline{G}$, $\forall x \in G$,

$$\begin{aligned}
\phi(a \oplus b)(x) &= \lambda_{a \oplus b}(x) && \text{by def}^{\text{n}} \text{ of } \phi \\
&= (a \oplus b) \oplus x && \text{by def}^{\text{n}} \text{ of } \lambda \\
&= a \oplus (b \oplus x) && \text{by (A.1)} \\
&= a \oplus \lambda_b(x) && \text{by def}^{\text{n}} \text{ of } \lambda \\
&= \lambda_a\big(\lambda_b(x)\big) && \text{by def}^{\text{n}} \text{ of } \lambda \\
&= (\lambda_a \circ \lambda_b)(x) && \text{by (A.6)}
\end{aligned}$$

ϕ is also bijective:

Injective. $\forall a, b \in G$, suppose $\phi(a) = \phi(b)$, then $\forall x \in G$,

$$\begin{aligned}
\phi(a)(x) = \phi(b)(x) &\Rightarrow \lambda_a(x) = \lambda_b(x) && \text{by def}^{\text{n}} \text{ of } \phi \\
&\Rightarrow a \oplus x = b \oplus x && \text{by def}^{\text{n}} \text{ of } \lambda \\
&\Rightarrow (a \oplus x) \oplus x^{-1} = (b \oplus x) \oplus x^{-1} && \\
&\Rightarrow a \oplus (x \oplus x^{-1}) = b \oplus (x \oplus x^{-1}) && \text{by (A.1)} \\
&\Rightarrow a \oplus e = b \oplus e && \text{by (A.3)} \\
&\Rightarrow a = b && \text{by (A.2)}
\end{aligned}$$

Surjective. $\forall a \in G$, $\exists \lambda_a \in \overline{G}$ such that $\phi(a) = \lambda_a$ by def$^{\text{n}}$.

Therefore, ϕ is an isomorphism from G to $\overline{G}$. □

Definition A.8: Generating Set

A *generating set* of a group $(G, \oplus)$ is a subset $S \subseteq G$ such that $\forall a \in G$, $\exists s_1, \ldots, s_n \in S$ for some $n \in \mathbb{Z}^+$, $a = s_1 \oplus \cdots \oplus s_n$. The elements of S are called the *generators* of $(G, \oplus)$.

Definition A.9: Cayley Graph

Given a group $(G, \oplus)$ with a generating set S, the *Cayley graph*, denoted as $Cay(G, \oplus, S)$, of $(G, \oplus)$ is a graph representation that encodes the abstract structure of $(G, \oplus)$. The vertices are the elements of G and $\forall a, b \in G$, there exists an edge (a, b) in $Cay(G, \oplus, S)$ if and only $\exists s \in S$, such that $b = a \oplus s$.

Theorem A.2

Every Cayley graph is vertex transitive.

Proof. Recall that a graph $G = (V, E)$ is said to be vertex transitive if $\forall v_1, v_2 \in V$, there is some automorphism of graph that maps v_1 to v_2.

Given a Cayley graph $Cay(G, \oplus, S)$, let a and b be two arbitrary elements of G, consider the function $h : G \to G$ where $\forall x \in G$, $h(x) = (b \oplus a^{-1}) \oplus x$.

Firstly, by Cayley's Theorem, any group is isomorphic to a permutation group, therefore this implies that h is bijective.

Secondly, note that$h$ maps a to b,

$$\begin{aligned} h(a) &= (b \oplus a^{-1}) \oplus a && \text{by def}^{\text{n}} \text{ of } h \\ &= b \oplus (a^{-1} \oplus a) && \text{by (A.1)} \\ &= b \oplus e && \text{by (A.3)} \\ &= b && \text{by (A.2)} \end{aligned}$$

Thirdly, h is an homomorphism, suppose there is an edge from x to y, this means $\exists s \in S$, such that

$$\begin{aligned} y = x \oplus s &\Rightarrow (b \oplus a^{-1}) \oplus y = (b \oplus a^{-1}) \oplus (x \oplus s) \\ &\Rightarrow (b \oplus a^{-1}) \oplus y = ((b \oplus a^{-1}) \oplus x) \oplus s && \text{by (A.1)} \\ &\Rightarrow h(y) = h(x) \oplus s && \text{by def}^{\text{n}} \text{ of } h \end{aligned}$$

which implies there is an edge from $h(x)$ to $h(y)$.

Therefore, h is an graph automorphism on $Cay(G, \oplus, S)$ that maps a to b. □

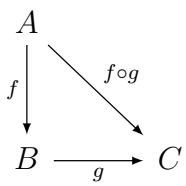
Appendix B

Category Theory

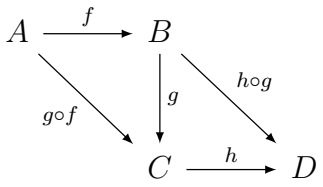
Definition B.1: Category

A *category* consists of a collection of *objects* and a collection of *arrows* subjected to the following axioms:

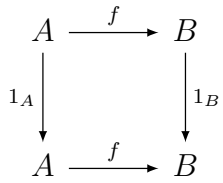
- For each arrow f, there are two associated objects in the object collection: $source(f)$ and $target(f)$. These are respectively the source and target of f. The notation $f : A \to B$ is used to indicate that $source(f) = A$ and $target(f) = B$.
- Given two arrows $f : A \to B$ and $g : B \to C$, there is an arrow $f \circ g : A \to C$ in the arrow collection. The arrow $f \circ g$ is called the *composite* of f and g. This is often expressed with the following *commutative diagram*[a]:



The composition of arrows is associative; that is to say, for all $f : A \to B$, $g : B \to C$, $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$:



- For each object A, there is an *identity arrow* $1_A : A \to A$ in the arrow collection such that for all $f : A \to B$, $f \circ 1_A = f = 1_B \circ f$:



[a] In a *commutative diagram*, all directed paths with the same start and endpoints lead to the same result by composition; *typically*, identity arrows are not portrayed unless required.

Remark B.2

Note that the definition of a category is entirely abstract; anything that satisfies this definition qualifies as a category. Typically, uppercase and bold characters (e.g., $\mathbf{C}$ and $\mathbf{D}$) are used to represent categories; for a category $\mathbf{C}$, $\mathbf{C}_0$ denotes the corresponding collection of objects and $\mathbf{C}_1$ denotes the corresponding collection of arrows. Also, in many ways, a categorical arrow generalizes the notion of function or morphism; therefore it also generalizes some of their properties.

Definition B.3: Isomorphism

An arrow $f : A \to B$ is called an *isomorphism* (in a categorical manner)

if there exists an arrow $g : B \to A$ such that $f \circ g = 1_B$ and $g \circ f = 1_A$. In this case, the arrow g is called the *inverse* of f. Naturally, inverses are also isomorphism.

Definition B.4: Functor

A *functor* $F : \mathbf{C} \to \mathbf{D}$ is a mapping from a category $\mathbf{C}$ to another category $\mathbf{D}$ such that objects are mapped to objects and arrows are mapped to arrows subjected to the following axioms:

- $F(f : A \to B) = F(f) : F(A) \to F(B)$
- $F(f \circ g) = F(f) \circ F(g)$
- $F(1_A) = 1_{F(A)}$

Remark B.5

Notice that a functor is used as both a mapping of objects and a mapping of arrows; thus, in order to clarify the notation, lowercase characters (e.g., f and g) are used to represent arrows and uppercase characters (e.g., A and B) are used to represent objects.

Remark B.6

As categories are entirely abstract entities, there is a problem in dealing with categories that are potentially "too big". For example, observe that functors composes in a similar manner as arrows and we can define a trivial *identity functor* $1_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ that maps a category $\mathbf{C}$ to itself by mapping objects and arrows to themselves. Therefore, there

can be possibly a category of categories: a category with all categories as objects and all functors as arrows. Such a notion is particularly problematic as much akin to the Russell's paradox: will such a category be an object of itself?

In category theory, categories can be distinguished according to the "size": a category is called *small* if both its collections are sets, otherwise it is called *large*. In this manner, a category of all small categories, **Cat**, can be defined instead; the category **Cat** is naturally large, thus avoiding paradox. While small categories allow set-theoretic notions to be employed, this class categories is too restrictive in application. Therefore, in our discussion, we shall assume that all categories are *locally small* instead.

Definition B.7: Locally Small Category

A category $\mathbf{C}$ is called *locally small* if for any two arbitrary objects A and B in $\mathbf{C}$, the collection of all the arrows with A as the source and B as the target, denoted as $Hom_{\mathbf{C}}(A, B)$.

Definition B.8: Dual Statement

Given any sentence Σ in the language of category theory, a *dual statement* Σ^* by

- interchanging the occurrence of "source" and "target" and
- interchanging the order of the arrow composition.

Such a statement is a well-formed sentence since only the symbols are changed.

Proposition B.1: Formal Duality

For any sentence Σ in the language of category theory, if Σ follows from the axioms for categories, then so does its dual statement Σ^*.

Proof. Firstly, notice that applying dual on the axioms of the categories does not change the axioms at all. Secondly, dual statements preserve the entailment since the substituted terms are treated as mere undefined constants. □

Proposition B.2: Conceptual Duality

For any statement Σ about categories, if Σ holds for all categories, then so does the dual statement Σ^*.

Proof. For any category $\mathbf{C}$, a *dual category* $\mathbf{C}^*$ can be defined by interchanging the source and target of each arrows. Naturally, $(\mathbf{C}^*)^* = \mathbf{C}$.

Firstly, notice that is a statement Σ holds for a category $\mathbf{C}$, then Σ^* holds for its dual category $\mathbf{C}^*$. Now if Σ holds for all categories, naturally, it holds for all dual categories. Thus, Σ^* will hold for all dual "dual categories" (i.e., all categories). □

Remark B.9

Propositions B.1 and B.2 together form the duality principle of categories. The idea is that any categorical notion will have an equally valid

dual notion, which is formed by reversing the order of composition and the words "source" and "target".

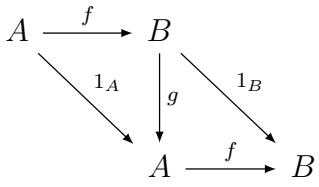
Definition B.10: Initial and Terminal Objects

Given a category $\mathbf{C}$, an object A is *initial* if for each object B, there is an unique arrow from A to B. Or equivalently, by skolemization, an object A is initial if there exists a mapping $(\!|\cdot|\!)$ from objects to arrows such that for any object B, $(\!|B|\!) : A \to B$. Dually, given a category $\mathbf{C}$, an object A is *terminal* if for each object B, there is an unique arrow from B to A. Similarly, by skolemization, an object A is terminal if there exists a mapping $|\!) \cdot (\!|$ from objects to arrows such that for any object B, $|\!) A (\!| : B \to A$.

Proposition B.3

Initial (terminal) objects are unique up to isomorphism.

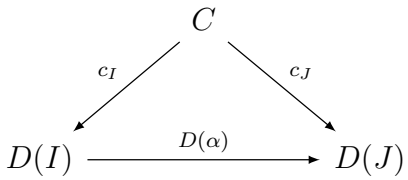
Proof. Given a category $\mathbf{C}$ with two initial objects, A and B, by definition, there exists an unique arrow $f : A \to B$ and another unique arrow $g : A \to B$. Then, the following commutative diagram proves that f and g are isomorphisms:



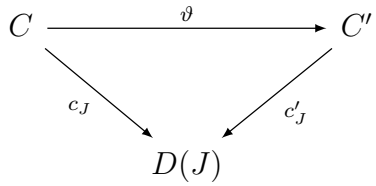
The uniqueness of terminal objects is proven dually. □

Definition B.11: Category of Cones

Given two categories, $\mathbf{J}$ and $\mathbf{C}$, and let the functor $D : \mathbf{J} \to \mathbf{C}$ be called a *diagram* of *type* $\mathbf{J}$ in $\mathbf{C}$. A *cone* to a diagram D is created with an identified object C in $\mathbf{C}$ together with a collection of arrows in $\mathbf{C}$, $c : C \to D(J)$, for each object J in $\mathbf{J}$ such that for each arrow $\alpha : I \to J$ in $\mathbf{J}$, the following triangle commutes:



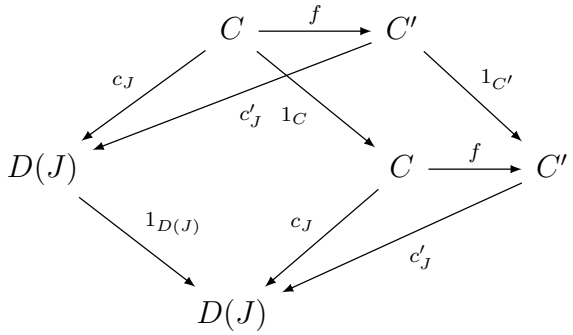
A morphism of cones $\vartheta : (C, c_J) \to (C', c'_J)$ is an arrow ϑ in $\mathbf{C}$ such that the following triangle commutes for all objects J in $\mathbf{J}$:



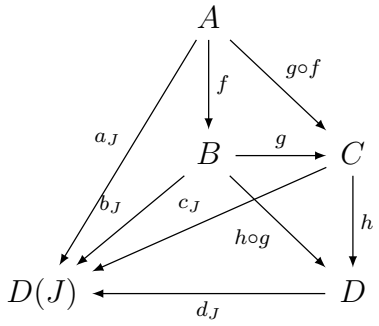
Therefore, by the construction of cones and their morphisms, a category can be identified with the cones as objects and the morphisms as arrows, this category is called the category of cones: $\mathbf{Cone}(D)$.

Proof. Given a diagram $D : \mathbf{J} \to \mathbf{C}$, for a cone on the object C in $\mathbf{C}$, the identity arrow of C, 1_C is naturally the identity arrow of the said cone; given a morphism of cones $f : (C, c_J) \to (C', c'_J)$, the follow

commutes for all objects J in $\mathbf{J}$:



Given the morphisms of cones, $f : (A, a_J) \to (B, b_J)$, $g : (B, b_J) \to (C, c_J)$, and $h : (C, c_J) \to (D, d_J)$, the following commutes:



Therefore, **Cone**(D) is a well-defined category. □

Definition B.12: Limit

A *limit* for a diagram $D : \mathbf{J} \to \mathbf{C}$ is a terminal object in **Cone**(D). This is denoted as $p_I : \varprojlim_J D(J) \to D(I)$.

Remark B.13: Cocone and Colimit

By the duality principle of categories, the dual of cone, called *cocone*, can be identified. In the category of cocones, **Cocone**(D), the initial object, called the *colimit*, can be dually defined.

The colimit is denoted as $q_I : D(I) \to \varinjlim_J D(J)$.

Definition B.14: Direct System

A *directed set*, $(A, \leq_A)$, is a non-empty set A equipped with a binary relation $\leq_A$ such that the following conditions hold:

Reflexivity. $\forall a \in A,\ a \leq_A a$.

Transitivity. $\forall a, b, c \in A,\ a \leq_A b \wedge b \leq_A c \Rightarrow a \leq_A c$.

Upperbound. $\forall a, b \in A,\ \exists c \in A,\ a \leq_A c \wedge b \leq_A c$.

A *direct system*, $\mathbf{Direct}(A, \leq_A)$, is a category constructed with the elements of A as objects and the arrows $f_{ij} : i \to j$ such that

- f_{ii} is the identity arrow on i, and
- $\forall i, j, k \in A,\ i \leq_A j \leq_A k \Rightarrow f_{ik} = f_{jk} \circ f_{ij}$.

Proof. Firstly, the identity arrow is well-defined by construction. Secondly, due to the transitivity of $\leq_A$, compositions of arrows exist and they are associative. Therefore, $\mathbf{Direct}(A, \leq_A)$ is a well-defined category. □

Definition B.15: ω-Colimit

A *ω-colimit* is the colimit for the diagram $D : \mathbf{J} \to \mathbf{C}$ on an identified category $\mathbf{C}$, where $\mathbf{J} = \mathbf{Direct}(\mathbb{N}, \leq)$, the direct system of natural numbers.

Definition B.16: Product and Coproduct

Given two objects A and B in a category $\mathbf{C}$, an object P is called the *product* of A and B if there are arrows $f_1 : P \to A$ and $f_2 : P \to B$ in $\mathbf{C}$ and $P \cong \varprojlim_J D(J)$, where $D : \mathbf{J} \to \mathbf{C}$ and $\mathbf{J}$ is a finite category with two objects and only the identity arrows:

$$* \qquad \star$$

such that $D(*) = A$ and $D(\star) = B$. The *coproduct* of two objects is defined dually. Given two objects A and B, their product is usually denoted as $A \times B$ while their coproduct is denoted as $A + B$.

A category where every two objects have a product (coproduct) is said to have products (coproducts).

Definition B.17: Preservation of Limits and Colimits

A function $F : \mathbf{C} \to \mathbf{D}$ *preserves* the limits of type $\mathbf{J}$ if $p_I : \varprojlim_J D(J) \to D(I)$ is a limit for a diagram $D : \mathbf{J} \to \mathbf{C}$, then $F(p_I) : F\big(\varprojlim_J D(J)\big) \to F\big(D(I)\big)$ is a limit for the diagram $FD : \mathbf{J} \to \mathbf{D}$. Briefly,

$$F\big(\varprojlim_J D(J)\big) \cong \varprojlim_J F\big(D(J)\big)$$

Dually, a function $F : \mathbf{C} \to \mathbf{D}$ preserves the colimits of type $\mathbf{J}$ if

$$F\big(\varinjlim_J D(J)\big) \cong \varinjlim_J F\big(D(J)\big)$$

for all colimits $q_I : D(I) \to \varinjlim_J D(J)$.

Definition B.18: Polynomial Functor

Given a category $\mathbf{C}$ with products, coproducts and terminal objects, a polynomial functor is an endofunctor $F : \mathbf{C} \to \mathbf{C}$ such that for all objects X in $\mathbf{C}$,

$$F(X) = C_0 + C_1 \times X + C_2 \times X^2 + \cdots + C_n \times X^n$$

where

$$\forall n \in \mathbb{Z}^+,\ X^n = \begin{cases} X & \text{if } n = 1, \\ X \times X^{n-1} & \text{otherwise} \end{cases}$$

and "C_k" represents a coproduct of C_k objects (e.g., "1" represents a terminal object and "2" $= 1 + 1$).

Definition B.19: F-Algebra and F-Coalgebra

Given an endofunctor $F : \mathbf{C} \to \mathbf{C}$ on a category $\mathbf{C}$, a *F-algebra* consists of an identified object A in $\mathbf{C}$ and an arrow $\alpha : F(A) \to A$.

A homomorphism $h : (A, \alpha) \to (B, \beta)$ of F-algebras is an arrow $h : A \to B$ in $\mathbf{C}$ such that the following diagram commutes:

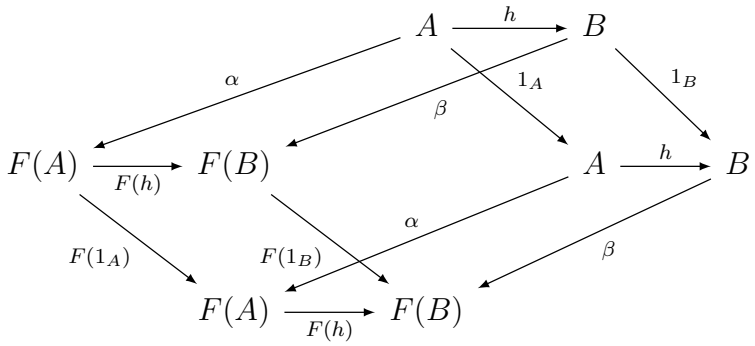
$$\begin{array}{ccc} F(A) & \xrightarrow{F(h)} & F(B) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{h} & B \end{array}$$

The category F-$\mathbf{Alg}(\mathbf{C})$ is identified with the F-algebras as objects and their homomorphisms as arrows.

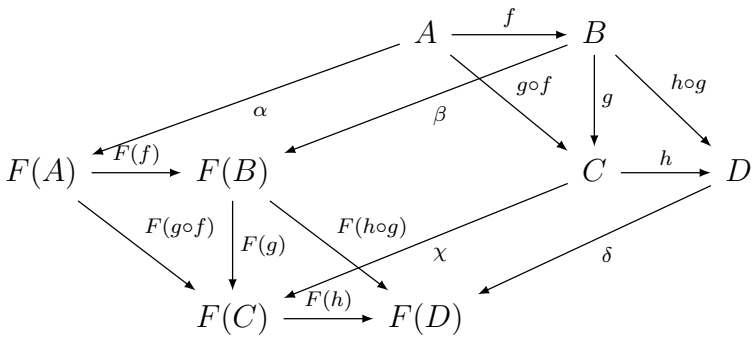
F-coalgebras can be dually defined and subsequently the category F-

Coalg(**C**) can also be identified.

Proof. The identity homomorphisms are the identity arrows in **C**. Given the homomorphism $h : (A, \alpha) \to (B, \beta)$, the following commutes:



Firstly, note that $F(1_A) = 1_{F(A)}$. Given the homomorphisms of F-algebras, $f : (A, \alpha) \to (B, \beta)$, $g : (B, \beta) \to (C, \chi)$ and $h : (C, \chi) \to (D, \delta)$, the following commutes:



Note that $F(g \circ f) = F(g) \circ F(f)$. Therefore, F-**Alg**(**C**) is a well-defined category. The proof of F-**Coalg**(**C**) is done dually. □

Lemma B.4: Lambek's Lemma

Given an endofunctor $F : \mathbf{C} \to \mathbf{C}$ on a category **C**, if $i : F(I) \to I$ is an initial F-algebra in F-**Alg**(**C**), then i is an isomorphism, meaning $P(I) \cong I$.

Proof. This is proven with the following diagram chase:

$$\begin{array}{ccccc} P(I) & \xrightarrow{P(\alpha)} & P\big(P(I)\big) & \xrightarrow{P(i)} & P(I) \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle P(i)} & & \downarrow{\scriptstyle i} \\ I & \xrightarrow{\alpha} & P(I) & \xrightarrow{i} & I \end{array}$$

The left part commutes because of $i : F(I) \to I$ being initial so there is an unique arrow $\alpha : (I, i) \to \big(P(I), P(i)\big)$. Therefore, i is an isomorphism such that $\alpha \circ i = 1_I$ and $i \circ \alpha = 1_{P(I)}$. □

Proposition B.5

If the category $\mathbf{C}$ has an initial object 0 and ω-colimit s, and the endofunctor $F : \mathbf{C} \to \mathbf{C}$ preserves ω-colimits, the F-$\mathbf{Alg}(\mathbf{C})$ has an initial algebra.

Proof. Since 0 is the initial object, there exists an unique arrow from 0 to $F(0)$: $(\!|\, F(0) \,|\!) : 0 \to F(0)$. Then, consider the "sequence":

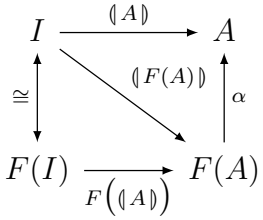
$$0 \xrightarrow{(\!|\, F(0) \,|\!)} F(0) \xrightarrow{F\left((\!|\, F(0) \,|\!)\right)} F^2(0) \xrightarrow{F^2\left((\!|\, F(0) \,|\!)\right)} \dots$$

a corresponding ω-colimit must exist. Let $I = \varinjlim_n F^n(0)$. Since F preserves ω-colimits, there is an isomorphism:

$$F(I) = F\big(\varinjlim_n F^n(0)\big) \cong \varinjlim_n F\big(F^n(0)\big) = \varinjlim_n F^n(0) = I$$

Therefore, given any arrow $\alpha : F(A) \to A$, the following diagram com-

mutes:



The arrows $(\!|\, A \,|\!) : I \to A$ and $(\!|\, F(A) \,|\!) : I \to F(A)$ come from the fact I is the initial object due to the ω-colimit. Furthermore, $F\big((\!|\, A \,|\!)\big)$ is unique due to the composition of $\cong$ and $(\!|\, F(A) \,|\!)$. Therefore the homomorphism $(\!|\, A \,|\!) : (I, \cong) \to (A, \alpha)$ is unique, meaning $\cong : F(I) \leftrightarrow I$ is the initial algebra of F-$\mathbf{Alg}(\mathbf{C})$. □

Definition B.20: Category of Sets

The category of sets, **Sets**, is a category whereby the objects are sets and the arrows are canonical set-theoretic functions. In this case, the identity arrows are the identity functions and the compositions of arrows are the compositions of functions.

Proof. The proof is immediate because by construction, **Sets** satisfies all the axioms of categories due to the similarity of definition of "identity" and "composition". □

Proposition B.6

The empty set in **Sets** is an initial object while the singleton sets are the terminal objects (i.e., isomorphic with one another).

Proof. From the empty set, there will be one function (i.e., the *null* function) to any other set. And from any set, there is only one func-

tion (i.e., maps all elements to the same element) to any singleton set. By construction, singleton sets are also isomorphic with one another because there is only one function that maps a singleton set $\{a\}$ to another singleton set $\{b\}$ (i.e., $f(a) = b$) and it is definitely bijective (i.e., $f^{-1}(b) = a$). □

Proposition B.7

The cartesian product of sets is a categorical product in **Sets** while the disjoint union of sets is a categorical coproduct in **Sets**.

Proof. Given two sets A and B in **Sets**, define two functions $p_1 : A \times B \to A$ and $p_2 : A \times B \to B$ such that $\forall (a, b) \in A \times B$, $p_1(a, b) = a$ and $p_2(a, b) = b$. For all sets Z such that there exists $z_1 : Z \to A$ and $z_2 : Z \to B$, there is an unique function $u : Z \to A \times B$ such that $\forall z \in Z$, $u(z) = \big(z_1(z), z_2(z)\big)$. This means that the cone created with $A \times B$ is the terminal object implying $A \times B$ is indeed the categorical product of A and B.

The disjoint union is proven in a dual manner. Note that since both cartesian products and disjoint unions are also sets, this means that **Sets** has all products and coproducts. □

Proposition B.8

Sets has ω-colimits.

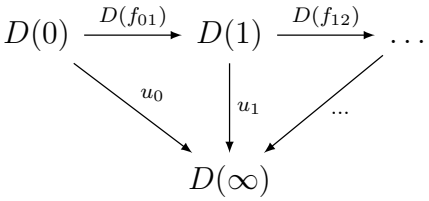
Proof. Given the diagram $D : \mathbf{Direct}(\mathbb{N}, \leq) \to \mathbf{Sets}$, define the follow-

ing equivalence relationship: $\forall n, m \in \mathbb{N}$, if $n \leq m$, then

$$\forall x_n \in D(n),\ x_n \sim D(f_{nm})(x_n)$$

Define the set $D(\infty)$ where the elements are the equivalence classes under $\sim$ of the form $[x_n]$ where $x_n \in D(n)$, $\forall n \in \mathbb{N}$ such that $[x_n] = [y_m]$ if and only if $\exists k \in \mathbb{N}$, $m, n \leq k$ and $D(f_{nk})(x_n) = D(f_{mk})(y_m)$.

By this construction, there exists an unique function from $D(n)$ to $D(\infty)$ for all $n \in \mathbb{N}$ such that the element is mapped to its equivalence class: $u_n : D(n) \to D(\infty)$, $\forall x_n \in D(n)$, $u_n(x_n) = [x_n]$. And the following commutes by the construction of $D(\infty)$:



Therefore, this is a cocone to D; in fact it is the initial object (i.e., the ω-colimit). Given any cocone to D with an identified set A such that $\forall n \in \mathbb{N}$, $\exists u'_n : D(n) \to A$, there is an unique function $f : D(\infty) \to A$ such that $\forall [x_n] \in D(\infty)$, $f([x_n]) = u'(x_n)$. Therefore, there is an unique arrow from $D(\infty)$ to any cocone to D, meaning that it is the initial object. □

Proposition B.9

Polynomial functors on **Sets** preserves ω-colimits.

Proof. Given the diagrams $D_1, D_2 : \mathbf{Direct}(\mathbb{N}, \leq) \to \mathbf{Sets}$, $\forall n, m \in \mathbb{N}$,

if $n \leq m$, then the following commutes due to the universality of products:

$$\begin{array}{ccccc} D_1(n) & \longleftarrow & D_1(n) \times D_2(n) & \longrightarrow & D_2(n) \\ \downarrow{\scriptstyle D_1(f_{nm})} & & \vdots & & \downarrow{\scriptstyle D_2(f_{nm})} \\ D_1(m) & \longleftarrow & D_1(m) \times D_2(m) & \longrightarrow & D_2(m) \end{array}$$

and the following commutes due to the universality of coproducts:

$$\begin{array}{ccccc} D_1(n) & \longrightarrow & D_1(n) + D_2(n) & \longleftarrow & D_2(n) \\ \downarrow{\scriptstyle D_1(f_{nm})} & & \vdots & & \downarrow{\scriptstyle D_2(f_{nm})} \\ D_1(m) & \longrightarrow & D_1(m) + D_2(m) & \longleftarrow & D_2(m) \end{array}$$

Therefore, products and coproducts do not affect the construction of $D(\infty)$ in the proof of Proposition B.8. Therefore,

$$\begin{aligned} & D_1(\infty) \cong \varinjlim_n D_1(n) \wedge D_2(\infty) \cong \varinjlim_n D_2(n) \\ \Rightarrow & D_1(\infty) \times D_2(\infty) \cong \varinjlim_n D_1(n) \times D_2(n) \wedge \\ & D_1(\infty) + D_2(\infty) \cong \varinjlim_n D_1(n) + D_2(n) \end{aligned}$$

Furthermore, it is direct to see that any constant endofunctor[a] on **Sets** preserves ω-colimits. Consider the functor $G : \mathbf{Sets} \to \mathbf{Sets}$ such that for all objects A in $G(A) = K$ for some object K and for all arrows f in **Sets**, $G(f) = 1_K$. Given $D : \mathbf{Direct}(\mathbb{N}, \leq) \to \mathbf{Sets}$, any cocone to $G(D)$ will have an unique morphism from K, implying that the cocone identified on K remains the initial object in $\mathbf{Cocone}\big(G(D)\big)$.

Taken together, the conclusion is that polynomial functor preserves ω-colimits. □

[a] A *constant functor* is one that maps all objects to a single object and all arrows to the identity arrow

Proposition B.10

Given a polynomial functor on **Sets**, $F : \mathbf{Sets} \to \mathbf{Sets}$, F-**Alg**(**Sets**) has an initial algebra and F-**Coalg**(**Sets**) has a terminal coalgebra.

Proof. This follows directly by the fact that **Sets** has an initial object (Proposition B.6) and ω-colimits (Proposition B.8) and by Proposition B.5, F-**Alg**(**Sets**), has an initial algebra. The terminal coalgebra of F-**Coalg**(**Sets**) is shown dually. □

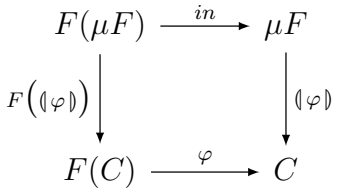
Remark B.21

Given the uniqueness of initial objects, the initial algebra of F-**Alg**(**C**) is often identified with the functor F, as such the object in **C** identified with the initial algebra is labelled as μF such that the initial algebra in F-**Alg**(**C**) is the arrow $in : F(\mu F) \to \mu F$ in **C**. On the other hand, the terminal coalgebra in F-**Alg**(**C**) is the arrow $out : \nu F \to F(\nu F)$ where νF is the identified object in **C** for the coalgebra.

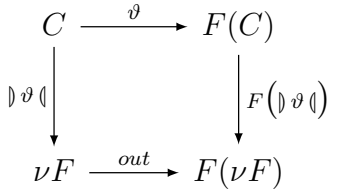
Definition B.22: Catamorphism and Anamorphism

Given the category F-**Alg**(**C**) with an initial algebra $in : F(\mu F) \to \mu F$, the unique arrow to any other F-algebra is called a *catamorphism*; for any F-algebra $\varphi : F(C) \to C$, the following commutes with the unique

catamorphism $(\!|\, \varphi \,|\!) : (\mu F, in) \to (C, \varphi)$:



Dually defined, given the category F-**Coalg**(**C**) with a terminal coalgebra $out : \nu F \to F(\nu F)$, the unique arrow from any other F-coalgebra to the terminal coalgebra is called an *anamorphism*; for any F-coalgebra $\vartheta : C \to F(C)$, the following commutes with the unique anamorphism $(\!|\, \vartheta \,|\!) : (C, \vartheta) \to (\nu F, out)$:



Remark B.23

Given a polynomial functor F : **Sets** $\to$ **Sets**, the category F-**Alg**(**Sets**) can be defined with an initial F-algebra $in : F(\mu F) \to \mu F$. By Lambek's Lemma (Lemma B.4), $F(\mu F) \cong \mu F$. In order words, μF represents as a fix-point for F. Also, due to the fact that the homomorphism in is isomorphic, every catamorphism $(\!|\, \varphi \,|\!) : (\mu F, in) \to (C, \varphi)$ may be expressed uniquely as $(\!|\, \varphi \,|\!) = \varphi \circ F\big((\!|\, \varphi \,|\!)\big) \circ in^{-1}$, where in^{-1} is the inverse of in.

As it turns out, a wide class of *inductive data types* (e.g., recursive data types and algebraic data types) of the *intuitionistic type teory* can

be represented as polynomial functor F on **Sets**. This fact, together with the universality of catamorphisms in F-**Alg**(**Sets**), means that all forms of recursive set-theoretic function can be expressed uniquely as a catamorphism.

Note that, dually, the same argument can be made on *coinductive data types* for F-**Coalg**(**Sets**). Similarly, every anamorphism $(\!|\, \vartheta \,|\!) : (C, \vartheta) \to (\nu F, out)$ may be expressed uniquely as $(\!|\, \vartheta \,|\!) = out^{-1} \circ F\big((\!|\, \vartheta \,|\!)\big) \circ \vartheta$, where out^{-1} is the inverse of out.

In functional programming languages, due to the common usage of list structures (or arrays), and the universality and expressiveness of catamorphism, there is usually a *fold* (sometimes called *reduce*) primitive that is an implementation of catamorphism on lists. In addition, as an implementation of functors on lists, the functional programming languages also have a map primitive. For example, in Haskell, there are two fold primitives and a map primitive:

$$map :: (a \to b) \to [a] \to [b]$$
$$foldl :: (b \to a \to b) \to b \to [a] \to b$$
$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

With the notion of universality and expressiveness of catamorphism in mind, one can say any function on a list may be uniquely (up to isomorphism) represented with a map followed by a fold. This is an immediate result from the definition of homomorphism between F-algebras; the application of a functor on a list is represented by the map primitive while the recursive nature of the fold primitive is implicit in the axioms of

homomorphism. If the data structure is a generic inductive data type, similar constructs may be implemented in Haskell to express in terms of catamorphisms.