Network-Aware
Big Data Processing

Lukas Rupprecht

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

January 2017
Abstract

The scale-out approach of modern data-parallel frameworks such as Apache Flink or Apache Spark has enabled them to deal with large amounts of data. These applications are often deployed in large-scale data centres with many resources. However, as deployments and data continue to grow, more network communication is incurred during a data processing query. At the same time, data centre networks (DCNs) are becoming increasingly more complex in terms of the physical network topology, the variety of applications that are sharing the network, and the different requirements of these applications on the network. The high complexity of DCNs combined with the increased traffic demands of applications has made the network a bottleneck for query performance.

In this thesis, we explore ways of making data-parallel frameworks network-aware, i.e. we combine specific knowledge about the application and the physical network to reduce query completion times. We identify three main types of traffic that occur during query processing and add network-awareness to each of them to optimise network usage.

1) **Traffic reduction for aggregatable traffic** exploits the physical network topology and the associativity and commutativity of aggregation queries to reduce traffic as early as possible. In-network aggregation trees utilise existing networking hardware and the tree topology of DCNs to partially aggregate and thereby reduce data as it flows through the network.

2) **Traffic balancing for non-aggregatable traffic** monitors the network throughput of an application and uses knowledge about the query to optimise the overall network utilisation. By dynamically changing the destinations of parts of the transferred data, network hotspots, which can occur when many applications share the network, can be avoided.

3) **Traffic elimination for storage traffic** gives control over data placement to the application instead of the distributed storage system. This allows the application to optimise where data is stored across the cluster based on application properties and thereby eliminate unnecessary network traffic.
To my family.

For their love and constant support in every aspect of my life.
I would like to thank the following people without whom this thesis would not have been possible:

First, I would like to thank Peter Pietzuch, for giving me the opportunity to pursue a PhD degree under his supervision. He guided me through these four years with patience and an infinite supply of advice on low-level technical details, the peculiarities of the research world and life. Working with him was a great and invaluable experience, professionally and personally, and I am grateful for all he taught me.

I would also like to thank the people on the NaaS team for a great working environment, some long submission nights and the whiskys afterwards. A special thanks goes to Paolo Costa, for his encouragement and enthusiasm, in particular, when deadlines became tight, and Alex Wolf, for sharing his wisdom with me.

Further, I want to thank my colleagues from 346 and all people from the LSDS group that I had the pleasure to meet during the last four years. Thanks to Alex, Alim, Andreas, Christian, Dan, Divya, Eva, Florian, Ioannis, Jeremy, Josh, Matthias, Nick, Panagiotis, Pierre, Pijika, Richard, Sei, Will, and Victoria for creating such a pleasant and fun atmosphere in the office.

A special thanks goes to Clint for always ordering dinner and keeping me company on the weekend, Luo, for making sure everybody stays motivated, Raul, for good drinks and even better music and Xi, for always listening and encouraging me. Sharing these four years with them was great fun and made things a lot easier.

I would also like to thank Rui, Dean and Bill at IBM for a great summer in 2014, the ongoing collaboration afterwards and for giving me the opportunity to peak inside industrial research.

A special thanks goes to Amani, for always being open and having a quick answer to any question concerning the PhD program, and Lloyd, for his invaluable advice on cluster management. I would also like to thank Clement and Paul for the fun nights and hours in the climbing gym which got me off my desk for a while.

I want to thank Myungjin Lee, Paul Kelly, and Thomas Heinis for carefully reading my thesis and providing insightful feedback and a great discussion during my viva which helped to improve this thesis.

Last but not least I want to thank my parents, Ruth and Martin, and my brothers, Thilo and Clemens, for always being there for me and believing in me. Without them, I would not be where I am today and I am very lucky to have such a family.
Declaration of Originality

This thesis presents my work in the Department of Computing at Imperial College London between October 2012 and November 2016.

Parts of the work were done in collaboration with other researchers:

- **Chapter 3.** The implementation of the base system for Solr was a joint effort between myself and Luo Mai. Afterwards, Luo Mai contributed the simulation results while I led the development of the Hadoop MapReduce integration and the testbed evaluation. Abdul Alim optimised the base implementation of the local aggregation tree.

- **Chapter 4.** The modelling effort for distributed joins was led by William Culhane.

I declare that the work presented in this thesis is my own, except where acknowledged above.

Lukas Rupprecht
January 2017
Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Contents

1 Introduction ........................................... 21
   1.1 Dealing with the data deluge .......................... 22
   1.2 The network bottleneck in data processing ............ 24
   1.3 Problem statement .................................. 27
   1.4 Research contributions ............................. 28
   1.5 Related publications .............................. 32
   1.6 Dissertation outline .............................. 33

2 Background ........................................... 35
   2.1 Data centre networks ................................ 36
       2.1.1 Structure and topologies ....................... 36
       2.1.2 Management and programmability ............... 41
       2.1.3 Traffic characteristics ....................... 44
   2.2 Data-parallel frameworks ........................... 49
       2.2.1 Operating principles ......................... 49
       2.2.2 Partition/Aggregate queries .................. 54
       2.2.3 Join queries ................................ 56
   2.3 Scalable storage technologies ....................... 59
       2.3.1 Block-based storage ......................... 60
       2.3.2 Object-based storage ....................... 62
   2.4 Summary ........................................ 65

3 Traffic Reduction for Aggregatable Traffic ............... 67
   3.1 Overview ........................................ 67
       3.1.1 Design challenges ............................ 68
       3.1.2 Design space ................................ 69
       3.1.3 Contributions .............................. 70
   3.2 NETAGG system design ............................ 70
       3.2.1 Deployment ................................ 70
       3.2.2 Feasibility study ........................... 72
       3.2.3 Requirements in data centre environments .... 74
   3.3 NETAGG components .............................. 76
       3.3.1 Agg boxes ................................ 76
### Contents

3.3.2 Shim layers ........................................... 79  
3.3.3 Application deployments ................................. 80  

3.4 Evaluation .............................................. 80  
3.4.1 Simulation-driven evaluation ............................ 81  
3.4.2 Testbed-driven evaluation .............................. 85  

3.5 Summary ................................................ 91

4 Traffic Balancing for Non-Aggregatable Traffic 93  
4.1 Overview ............................................... 93  
4.1.1 Design challenges .................................... 94  
4.1.2 Design space ......................................... 95  
4.1.3 Contributions ........................................ 96  

4.2 Analysis of network skew in joins .......................... 96  
4.2.1 Distributed joins ..................................... 97  
4.2.2 Network usage ....................................... 97  
4.2.3 Receiver-side network skew ............................ 99  
4.2.4 Sender-side network skew ............................. 101  

4.3 Lazy partitioning ......................................... 103  
4.3.1 Lazy partitions ....................................... 103  
4.3.2 Detecting and balancing network skew .................. 104  
4.3.3 Consumption without network skew ..................... 107  

4.4 SQUIRRELJOIN implementation ........................... 108  
4.4.1 Architecture ......................................... 108  
4.4.2 Mitigating estimation error ............................ 110  
4.4.3 Fault tolerance ....................................... 111  

4.5 Evaluation ............................................... 111  
4.5.1 Experimental set-up .................................... 111  
4.5.2 Different network skew ................................ 113  
4.5.3 Different workloads ................................... 117  
4.5.4 Robustness .......................................... 118  
4.5.5 Scalability and overhead ............................... 120  
4.5.6 Parameter choices ..................................... 122  

4.6 Summary ................................................ 124

5 Traffic Elimination for Distributed Storage I/O 127  
5.1 Overview ............................................... 127  
5.1.1 Design challenges .................................... 128  
5.1.2 Design space ......................................... 129  
5.1.3 Contributions ........................................ 129  

5.2 Performance issues for analytics on object stores ......... 130
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Experimental set-up and workloads</td>
<td>130</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Overall overhead</td>
<td>131</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Map phase overhead</td>
<td>132</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Reduce phase overhead</td>
<td>133</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Discussion</td>
<td>134</td>
</tr>
<tr>
<td>5.3</td>
<td>SwiftAnalytics design</td>
<td>135</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Object chunking</td>
<td>135</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Placement control</td>
<td>136</td>
</tr>
<tr>
<td>5.4</td>
<td>Locality-aware writes</td>
<td>137</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Locality using two namespaces</td>
<td>137</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Local rename and upload operations</td>
<td>138</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Implications on object stores</td>
<td>139</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Implementation in Swift</td>
<td>140</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation</td>
<td>140</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Effectiveness of local renames</td>
<td>141</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Effectiveness of object chunking</td>
<td>144</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary</td>
<td>145</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>147</td>
</tr>
<tr>
<td>6.1</td>
<td>Thesis summary</td>
<td>148</td>
</tr>
<tr>
<td>6.2</td>
<td>Future work</td>
<td>151</td>
</tr>
</tbody>
</table>

References 153
List of figures

1.1 Evolution from MapReduce to arbitrary dataflows ........................................ 23
1.2 Time line of work from the database community on network optimisation .... 25
2.1 A multi-rooted tree DCN topology .................................................................. 36
2.2 Example of the bisection bandwidth of a network (all links are 1 Gbps) ...... 37
2.3 Data parallelism across multiple machines and cores ................................. 50
2.4 Example for logical and physical operator pipelines ................................. 51
2.5 Different forms of edge-based aggregation .................................................... 55
2.6 A distributed join ......................................................................................... 56
2.7 HDFS architecture ......................................................................................... 60
2.8 Object Store architecture at the example of OpenStack Swift .................... 63
2.9 Comparison of four different MapReduce workloads running on top of Open-

Stack Swift and HDFS ................................................................................. 64
3.1 On-path aggregation leveraging the DCN topology and middleboxes. The x/y
notation refers to "layer/node" ........................................................................ 68
3.2 A NETAGG deployment with sample workflow ........................................... 71
3.3 Benefit of NETAGG for different processing rates ...................................... 73
3.4 Cost/performance benefit of NETAGG over other topologies .................... 74
3.5 Architecture of a NETAGG Agg box with two deployed applications (shading
represents different aggregation functions) .................................................... 77
3.6 CDF of flow completion times for all traffic ................................................. 82
3.7 CDF of flow completion times for non-aggregatable traffic ....................... 82
3.8 Relative FCT with varying output ratio \(\alpha\) .................................................. 82
3.9 CDF of link traffic \((\alpha = 10\%)\) ................................................................. 82
3.10 Relative FCT with varying fraction of aggregatable traffic ....................... 83
3.11 Relative FCT with varying oversubscription ratio ...................................... 83
3.12 Relative FCT for different partial NETAGG deployments ....................... 84
3.13 Relative FCT with in 10 Gbps network with varying oversubscription ....... 84
3.14 Relative FCT with varying straggler ratio .................................................. 85
3.15 Throughput of an in-memory local aggregation tree ................................... 86
3.16 Throughput with varying number of Solr clients ........................................ 86
3.17 Latency with varying number of Solr clients .............................................. 87
3.18 Throughput with varying output ratio .......................................................... 87
3.19 Latency with varying number of Solr backend servers .......................... 88
3.20 Throughput for CPU-intensive aggregation and different number of Agg boxes 88
3.21 Throughput for different number of CPU cores ................................. 89
3.22 Throughput and completion time for different MapReduce jobs ......... 89
3.23 Completion time for different input data sizes .............................. 90
3.24 Comparison of scheduling approaches in NetAgg ...................... 91
4.1 Network channel potential and utilisation .................................... 98
4.2 Example of lazy partitioning (some lazy partitions have already been assigned) .... 104
4.3 SQuIReLJ0In architecture (solid lines represent the data plane and dashed lines represent the control plane) ................................. 109
4.4 Completion times under background traffic from different applications ... 113
4.5 Completion times for different numbers of reducers ...................... 114
4.6 Completion times for different background traffic start times .......... 114
4.7 Completion times for different background traffic stop times .......... 115
4.8 Completion times for different background network skew waves ...... 115
4.9 Completion times for different numbers of background flows ........ 116
4.10 CDF of completion times for different probe table sizes ................ 117
4.11 Completion times for different join queries .............................. 118
4.12 Completion times for different degrees of data skew .................. 119
4.13 Completion times for different degrees of CPU skew .................. 119
4.14 Estimation error in each detection iteration ............................... 120
4.15 Completion times for different cluster sizes ................................ 121
4.16 Completion times for increasing input data sizes ......................... 121
4.17 Completion times of the assignment algorithm for increasing cluster sizes ... 122
4.18 Completion times for different skew thresholds $\tau_{skew}$ ............... 123
4.19 Completion times for different consumption thresholds $\tau_{consume}$ .. 123
4.20 Completion times for different assignment intervals $\tau_{assign}$ ........ 124
4.21 Completion times for different lazy partition sizes .................... 124
5.1 Completion times on HDFS and Swift ...................................... 131
5.2 Breakdown of time spent in map and reduce phase ........................ 132
5.3 Amount of data read during map phase Wordcount .................... 132
5.4 Sum of time spent in calls to Hadoop’s FileSystem API ............... 133
5.5 Sum of time spent in HTTP requests to Swift API (Wordcount) ...... 133
5.6 SWIFTAnalytics design ..................................................... 136
5.7 Locality-aware writes in a consistent-hashing based object store .... 137
5.8 Comparison of different renaming approaches for MapReduce workloads .... 141
5.9 Comparison of different renaming approaches for Facebook MapReduce workload 141
5.10 Comparison of different renaming approaches for Spark workloads .... 142
List of figures

5.11 Comparison of different replication strategies for MapReduce workloads . . . 142
5.12 Comparison of local renames and link files under 1-way replication . . . . . . 143
5.13 Request latencies for different object sizes (dark-coloured lines represent SWIFT-
ANALYTICS and light-coloured Swift) . . . . . . . . . . . . . . . . . . . . . . . . . . 144
5.14 Map time for a Wordcount job with different object chunk sizes . . . . . . . 145
Big Data has been described by some as a “cultural movement” that allows us to discover how humans behave and interact with each other and the world [Big16]. While the exact description is arguable, there is no doubt that Big Data has become one of the main driving forces for both science and industry in the current century. The analysis of large data sets has allowed businesses to understand their users’ behaviour and based on these findings, personalise experience, recommend relevant products, or display ads that might be of interest [LSY03] [MHS+13]; it has enabled artificial intelligence to perform complex tasks that have long been thought to be impossible [SHM+16] [Goo16a]; and it has facilitated major scientific breakthroughs such as the discovery of the Higgs Boson [CKS+12].

Besides the new opportunities that Big Data provides, it also poses new challenges to the systems that store and process data. Traditional architectures based on monolithic mainframes are unable to cope with the rapidly increasing amounts. Their vertical scalability (scale up) model makes them expensive to scale as to store and process more data, more powerful hardware is required but data is growing faster than Moore’s Law [Sto13]. This has caused companies to build large-scale data centres based on off-the-shelf servers instead of small mainframe clusters. Rather than vertically, these data centres scale horizontally (scale out), which means that more storage and processing capacity can be added by adding more servers. Servers are based on commodity hardware and hence scaling becomes highly cost effective.
While the scale out approach allows to deal with the massive amounts of data, one major drawback is that computation becomes distributed. As more and more servers participate in data processing, network communication between these servers increases and becomes a critical factor for achieving high performance and is often a bottleneck. This problem is intensified by the complexity of modern data centre networks, which use special topologies and have to support a large variety of applications simultaneously. Applications are not aware of this complexity and hence can experience additional performance issues. In this thesis, we present approaches to make Big Data processing systems *network-aware*, i.e. we extend them with knowledge about the physical network to optimise their interaction with the network during large-scale data analysis.

1.1 Dealing with the data deluge

Decision-support systems have been used and developed since the 1970s [SWC+02]. However, *data-driven* decision making has only become possible recently based on two developments: first, the rapid growth of the Internet and more recently, the Internet-of-Things has created a vast amount of potential data sources and made the collection of data cheap; second, the cost for storage has been constantly decreasing [Dis16], making it feasible to store data at an unprecedented scale. As a result, businesses and scientists are increasingly utilising the potential of large-scale data analysis.

The rise of Big Data that followed and the corresponding infrastructure shift from scale up-based processing to scale out has caused the emergence of new, distributed software solutions for data processing to effectively utilise the available hardware. To support horizontal scalability, systems deploy a *shared-nothing* architecture, i.e. each server or node in the distributed system is an autonomous entity that does not share memory or disk with other nodes [Sto86]. This approach allows to scale to very large numbers of servers. While shared-nothing architectures have been used in the design of parallel databases for more than 30 years [DGS+90], the scale of Big Data and data centres has introduced new challenges. First, failures (both hardware or software) are common rather than an exception. For example, out of 10,000 servers in a data centre, 10 will fail each day [Dea08]. This is due to the fact that cheaper commodity hardware is less reliable. As a result, fault-tolerance is one of the most important requirements for modern data processing systems. Second, data can take many forms such as user logs, images, or plain text and is often unstructured or semi-structured. This requires new APIs that allow to specify queries beyond the relational model and SQL.

To satisfy the above requirements, a new breed of systems has evolved. These *data-parallel processing systems* [DG04] [IBY+07] [ZCD+12] provide intuitive query interfaces for arbitrary user-defined functions (UDF) and transparently hide failures. In 2004, Google introduced the MapReduce system [DG04] to easily program queries, which are automatically parallelised
1.1 Dealing with the data deluge

MapReduce programming model

DAG-based programming model

(a) MapReduce programming model
(b) DAG-based programming model

Figure 1.1: Evolution from MapReduce to arbitrary dataflows

and executed across many machines in a data centre with fault tolerance. MapReduce gained rapid popularity due to its open source implementation in the Hadoop framework [Had16a].

MapReduce exposes a simple programming model in which users program a query or job by implementing a map and a reduce function. The job is then executed on a subset of machines (called a cluster) in the data centre performing a map and a reduce phase (see Figure 1.1a). Both phases have a number of map and reduce tasks, which each operate on a partition of the input data in parallel. This is called data-parallelism. The MapReduce system automatically detects failures and restarts failed tasks instead of having to rerun the entire query. Additionally, it takes care of assigning tasks to physical nodes and scheduling them.

The simplicity of the 2-stage MapReduce model allows fast development of jobs. However, because of that simplicity, MapReduce is also inflexible and more complex data processing workflows are tedious to realise. As a result, systems were created that generalise MapReduce to arbitrary dataflows represented by directed acyclic graphs (DAGs). Dryad proposed a DAG-based programming model in which queries can be constructed by combing multiple high-order functions (or operators) such as map, reduce, filter, or aggregate to form a DAG (see Figure 1.1b). As data flows through each stage of the DAG, the individual functions are applied and then data is propagated downstream to the next stage. Since the publication of Dryad, a myriad of systems have been developed for different application domains such as graph processing [MAB+10] [GLG+12], machine learning [ABC+16], SQL processing [AXL+15] [KBB+15], or stream processing [MMI+13] [CFMKP13] that follow this approach. Their ease of use and the variety of available systems have made them the most prominent choice for any kind of large-scale data processing.
The scale-out approach of data-parallel frameworks allows them to utilise large compute clusters of hundreds and thousands of nodes to store and process data. However, this ability comes at the cost of an increased amount of network communication during query execution. As shown in Figure 1.1, some operators such as reduce, aggregate, or join require to exchange data between tasks, which leads to network traffic. As more and more data is processed, the data that is sent across the network during job execution also increases [SOA+15]. As a result, the network becomes a critical component for the performance of a job and is often cited as a bottleneck [AGM+10] [CZS14] [JBK+13]. For example, Facebook reports that their MapReduce jobs spend up to 70% of their time in network transfers [CZM+11] while Microsoft reports a median increase in job completion times by 62% [AKG+10] due to network communication. Other work has found that analytical SQL queries spend up to 70% of their execution time transferring tuples across the network [PSR14].

Network optimisations for data processing systems have been a consistent theme in database research. However, existing work has not sufficiently considered the new data centre environment. Data centre networks resemble a closed system, usually under the control of a single operator, and with well known properties such as available hardware or network paths. This is contrary to the Internet, which consists of many different entities under different control domains and has a high degree of heterogeneity. On the other hand, data centre networks often support a multitude of applications that range from data processing to web services and distributed storage, which all have to share the network. Hence, the network is not a dedicated resource as, for example, in a small distributed database cluster. This introduces unique challenges and optimisation opportunities for data processing systems that are deployed in such an environment.

In this thesis, we argue that data-parallel processing systems should be network-aware, i.e. they should understand the environment that they are deployed in. This includes understanding both physical properties of the environment, such as the network topology and the available networking hardware, and operational properties, such as co-located applications and their usage of the network. This knowledge can then be used to optimise the communication during query execution and relieve pressure from the network. In the remainder of this chapter, we will discuss the data centre networking environment in more detail and present its challenges. We will then formulate the problem statement and present the research contributions of this thesis. The chapter concludes with the outline of the dissertation.

1.2 The network bottleneck in data processing

Network optimisations within the scope of data processing have been a recurring topic throughout the years. Figure 1.2 shows some important examples of work from the database...
Figure 1.2: Time line of work from the database community on network optimisation community on this matter during the last 20 years\(^1\). While this is not an exhaustive list, it demonstrates that the network has always been a critical component in data processing systems, and there has been a constant effort to improve network communication during query execution. The figure also illustrates how the environment has continuously been changing, starting from wide-area distributed databases [AFTU96] [UFA98] and then moving to federated databases [AH00] [WBTD08], sensor-networks [MFHH02], and stream processing [SHCF03] [SMW05]. More recent work has looked at optimising network communication for data-parallel processing systems [BPE+10] [ETO+11] [PSR14]. However, the focus has been on reducing communication solely from an application point-of-view without considering the specifics of the underlying network. We identify three main properties that come with the data centre environment and that make efficient network communication challenging in data-parallel processing and eventually lead to the network becoming a bottleneck:

1. **Complex network topologies.** As data centres grow, the interconnecting network topology becomes more complex. To connect a large number of servers, data centre networks are arranged in multiple layers forming a *multi-rooted tree*. Compared to simpler topologies such as a star topology or a fully connected topology, multi-rooted trees provide better scalability. However, data centre operators often face a trade-off between the available bandwidth at end hosts and cost. A *full-bisection bandwidth network* allows all end hosts to communicate with each other at all times using the entire bandwidth provided by their network interface card (NIC). Proposals for cost-efficient full-bisection bandwidth networks exist [AFLV08] [GHJ+09] [GLL+09], however, they are still expensive. In 2008, a full-bisection 10 Gbps network with 27,648 servers was estimated to cost $690 million [AFLV08]. While prices have decreased by an order of magnitude in the last decade [GMP+16], traffic demands have been increasing at the same time [SOA+15]. Hence, 10 Gbps networks are slowly replaced by faster technology such as 40 Gbps networking, which again is more expensive. This gap is likely to persist in the future.

\(^1\)We will discuss some of the work in more detail in Chapter 2.
Chapter 1. Introduction

As a result, the individual bandwidth of a single server is usually scarce, limiting the throughput during large data transfers. Additionally, data centre networks are often oversubscribed, meaning that the aggregate bandwidth of all end hosts is larger than the aggregate bandwidth of the core network, i.e. the infrastructure that interconnects all end hosts [FA13]. This further reduces the throughput when many servers exchange data.

2. Network sharing between different applications. Data centres provide a vast amount of compute and storage resources. To not waste any resources and maximise the utilisation of the data centre, a multitude of applications are deployed and share the available hardware. For example, at Google, data centres run a mix of high-priority production systems such as search or Gmail, co-located with lower priority batch jobs for data processing [VPK+15], and virtual machines from their cloud offering [Goo16b]. Sharing can create interference between the different applications, which can lead to slowdown and underutilisation. Hence, resource isolation and scheduling is needed to coordinate access to the hardware. Existing mechanisms such as virtual machines [BDF+03] or containers [Doc16] achieve good isolation for compute and memory resources, and cluster schedulers such as Mesos [HKZ+11] or Yarn [VMD+13] allow scheduling of applications based on compute and memory requirements. However, these approaches do not consider the network as an equal resource [PDHK12].

Consequently, network sharing can lead to interference and result in unstable throughput [SDQR10] [LDGB13] [XHN14] and network hotspots. At these hotspots, different applications have to share the available bandwidth, and as a result each application can only utilise a fraction of it. This slows down data transfers and the network becomes a bottleneck.

3. Diverse network requirements of different applications. The variety of applications running in data centres also have different requirements on the network. Some applications, such as key-value stores [NFG+13] or stream processing engines [CFMKP13], are latency-sensitive, i.e. they depend on the fast delivery of individual packets but do not utilise a large amount of bandwidth. They often communicate via an $N$-to-1 pattern which means that many senders send data to few receivers for aggregation or result delivery [AGM+10]. Other applications such as MapReduce [DG04] are batch-oriented applications and require high bandwidth and high throughput. They exchange data in an $N$-to-$M$ pattern where often $N = M$ (see Figure 1.1a) [DG04]. Optimally supporting both types of applications is challenging as the individual goals are conflicting: latency-sensitive applications need short buffers at network switches to avoid queueing and ensure fast delivery; on the other hand, batch applications require large buffers to achieve high throughput [AKE+12].

These different requirements lead to a variety of problems. When deployed concurrently, latency-sensitive applications suffer from queueing at switches and experience packet loss and long tail latencies [AKE+12] [GSG+15]. This is made worse by the TCP outcast
1.3 Problem statement

The above presented problems reveal that, due to their scale and the variety of hosted applications, data centre networks are increasingly complex. This has created a disconnect between applications and the network as the traditional assumptions that applications make, such as uniform bandwidth or dedicated network access, do not hold anymore. This disconnect is due to two major problems, which we discuss in more detail below:

1. Applications are unaware of data centre-specific problems

   Existing applications are unaware of the specifics of data centre networks, which can lead to low job performance. We identify the following three main reasons that cause job performance to degrade:

   (a) Longer transfer times due to scarce edge bandwidth and oversubscription

       As mentioned above, modern data centre networks often have to trade off cost for bandwidth. Full-bisection bandwidth networks are expensive and become infeasible for high-speed networks. As a result, end hosts only have a limited amount of network bandwidth available and the network experiences a certain amount of oversubscription. This limits transfer speeds during large data exchanges.

   (b) Emergence of stragglers due to shared network accesses

       As many applications share the data centre network, different network paths may experience a higher/lower utilisation at different points in time. This network skew can lead to variations in the available bandwidth for the individual flows of a query at runtime and an imbalance in network utilisation. Slower flows will stall the progress of the query while faster ones finish early and leave resources idle.

   (c) Additional job latency due to distributed storage I/O

       As data is too large to be stored on a single machine, distributed storage systems are needed to hold them. Hence, reads from or writes to the storage can cause additional network traffic as data might have to be transferred from/to remote locations. While distributed file systems for data processing applications support data locality, other storage systems such as object stores trade off locality for scalability, which leads to delays during storage accesses.

A network-aware solution knows about the specifics of the environment and hence could account for the above described problems by exploiting application knowledge. This problem [PDHK12]. Additionally, many-to-few patterns overload the few receivers which leads to buffer overflows and TCP incast [CGL+09]. As a result throughput collapses and senders, sending to the affected receivers, experience a slow-down.
would improve not only job performance but also benefit other applications that require network resources as it relieves network pressure and reduces contention.

2. Applications and networks are only loosely integrated

Current approaches for network optimisation in data processing systems are mainly focused on application-level improvements. This leads to a loose integration between the systems that use the network and the physical network itself. As a result, additional optimisation opportunities are missed. Modern data centre networks represent a controlled environment with a well-defined topology and advanced properties such as multiple routing paths, additional networking hardware such as middleboxes, and advanced control mechanisms such as software-defined networking (SDN).

This safe and predictable environment would allow a network-aware solution to have a tighter integration with the network. That means that static knowledge and control is made available to the system, enabling it to reason about the network and thereby improve its interaction.

Existing approaches to optimise the network only partially solve the above problems as they do not consider the data centre environment, i.e. they treat the network as a black box. As a result, they miss out on additional optimisation opportunities. As data sizes increase and jobs place higher load on the network, it is important to consider the environment in order to keep job completion times low and use the network efficiently.

1.4 Research contributions

In this thesis, we identify opportunities to solve the above discussed data centre specific problems. We consider three main types of traffic that can occur during job execution:

1. **Aggregatable traffic** denotes a data exchange in which receivers compute an aggregation function such as a sum or an average over the received data.

2. **Non-aggregatable traffic** denotes a data exchange in which receivers cannot aggregate received data such as during a join or sort operation.

3. **Storage traffic** is the traffic that occurs when a data-parallel system reads input data from or writes result data to the distributed storage.

For each of these types, we show how data centre networks can impact performance and then present solutions to overcome these problems. This leads to better individual query performance and also to a better overall utilisation of the network for all deployed applications. The core idea of our solutions are based on a tighter integration between the data-parallel system and the network. This allows us to open up new opportunities for optimising transfers.
that previous approaches could not realise as their main focus is on generic application-level optimisations. We demonstrate how exploiting the specifics of the data centre environment and active consideration of networking resources can help improve the performance of data-parallel jobs. We introduce three novel approaches to optimise network traffic, each of them focusing on a specific type of traffic:

**Traffic reduction for aggregatable traffic**

We propose in-network aggregation trees to reduce aggregatable traffic as early as possible when it flows through the network. Early traffic reduction decreases transfer times and saves bandwidth for other transfers in the data centre, in particular when the network is oversubscribed. The approach is based on three main observations about applications and the physical network: (1) aggregation functions in large-scale data processing queries often have the purpose to reduce data to extract their main essence and are associative and commutative; (2) data centre networks are often arranged in a multi-rooted tree topology; and (3) modern data centre networks deploy a variety of middleboxes that can be used as additional computational devices in the network. By combining this application knowledge with the knowledge about the underlying network, we can construct in-network aggregation without changes to the infrastructure.

The contributions of this work are:

1. **Middlebox-based in-network aggregation trees**: We leverage existing middleboxes in data centres and exploit the tree topology of the network to construct in-network aggregation trees. During the exchange of aggregatable traffic, data flows along these trees and at each hop in the tree, all incoming flows are aggregated to form a single outgoing flow. This reduces partial data as early as possible and decreases network utilisation.

2. **Transparent application support**: We propose a distributed architecture that is able to transparently reroute flows through the in-network aggregation trees without having to change existing queries. The architecture is based on shim layers, which transparently intercept query traffic at network sockets, forward it to the next hop in the tree, and correctly hide the altered number of flows from the data-parallel system.

3. **Efficient aggregation computation**: To allow users to reuse their existing queries without any changes, we propose an efficient, generic runtime system that decomposes the aggregation computation and parallelises the execution across multiple cores. The runtime is written in Java in order to support the majority of data-parallel systems.

We implement the above design in the NetAgg system for two applications: Apache Hadoop MapReduce [Had16a] and the Apache Solr search engine [Sol16]. We evaluate NetAgg on a
private 16-node research testbed and in large-scale simulations and show that we can achieve a 5.2× speed-up for MapReduce jobs and a 9.3× speed-up for Solr queries.

**Traffic balancing for non-aggregatable traffic**

We propose *dynamic traffic rebalancing* to change the receivers of data partitions while a query is executed. Traffic rebalancing can even out imbalances in network utilisation due to network skew and thereby increase single query completion times and overall network utilisation. To always optimally utilise the network, traffic needs to be shifted from network hotspots to the underutilised portions of the network at runtime. By using knowledge of the current network utilisation, an application can detect network skew and take measures to reroute its traffic accordingly. We focus on large-scale join operations, which are particularly prone to suffer from skewed network accesses as they require to exchange a large amount of data and hence have long running network transfers.

The contributions of this work are:

1. *Lazy partitioning:* We propose *lazy partitioning*, a method for dynamically changing the receivers of portions of the data sent during the execution of a distributed join. Distributed joins are stateful, i.e. their correctness depends on the data that they have already received. Hence, changing receivers for parts of the data at runtime can produce incorrect results. Lazy partitioning is able to change receivers and guarantee correctness by buffering selected parts of the data at senders in lazy partitions. These partitions can be dynamically assigned to receivers without corrupting existing state.

2. *Network skew-aware join processing:* We introduce **SquirrelJoin**, a novel distributed join algorithm that uses lazy partitioning to dynamically react to and rebalance network skew. **SquirrelJoin** monitors the current network utilisation and based on those measurements estimates the completion time for each node. It then decides which lazy partitions to assign to which receivers in order to minimise join completion time. **SquirrelJoin** uses a lightweight heuristic to efficiently compute periodic assignments.

3. *Formal problem analysis:* We present a formal model for the problem of *dynamic key reassignment*, i.e. the on-the-fly assignment of data partitions to receivers under network skew, in distributed joins. We formally analyse the problem for the two cases in which network skew affects the available bandwidth of senders and of receivers. We provide bounds on the improvements that can be achieved for both cases when using lazy partitioning and incorporate these findings into the design of **SquirrelJoin**.

We implement **SquirrelJoin** in the Apache Flink framework [Fli16a] and evaluate its effectiveness on a 16-node Google Compute cluster with a variety of real-world applications.
sharing the network and synthetic benchmarks. Our results show that SQUIRRELJOIN achieves speed-ups of up to 2.3×, while only adding a fixed 10s increase in job completion time in case of no network skew.

Traffic elimination for storage traffic

We propose a locality-aware object storage architecture to efficiently serve data-parallel frameworks as a storage layer. Object stores have a decentralised architecture that allows them to scale arbitrarily but sacrifices control over data placement—object stores are location agnostic. Improved placement control can increase data locality and eliminate additional network I/O when the object store is used as a source/sink for data-parallel frameworks and thereby reduce job completion times. Adding the notion of location to an object store should not affect their scalability, i.e. not introduce a centralised component. By keeping their decentralised nature, locality-aware object stores represent a highly scalable storage solution for large-scale data analysis.

The contributions of this work are:

1. **Object storage for Big Data processing:** We propose a locality-aware object store design that is able to efficiently serve queries from data-parallel frameworks without introducing significant overhead. The design choices are based on a detailed benchmarking study that reveals two major problems when running data analytics on top of object stores:

   **Write-locality.** Object stores work on the principle of consistent hashing that allows locating stored objects across the cluster in a decentralised way. This creates a fixed mapping between object names and storage devices, which does not work well for some operations. For example, renaming an object degenerates to re-uploading the same file with a different name, resulting in additional network traffic. As data-parallel systems require several renames of the job results, performance decreases significantly.

   **Read locality.** Object stores store complete objects, such as files, instead of chunking them into multiple blocks. This coarse storage granularity generates imbalances in the I/O load during reads and reduces data locality, leading to network hotspots.

   To overcome these problems, our object store design provides placement control to break the mapping between object names and devices and transparently chunks uploaded objects into smaller parts.

2. **Locality-aware writes:** We describe an approach to enable placement control in a decentralised, consistent-hashing based object store that does not require any additional centralised entities. Storage destinations can be explicitly specified via additional metadata and the additional location information is stored decentralised in the object store such that it can be accessed efficiently with only a small overhead.
Chapter 1. Introduction

We demonstrate the feasibility of our design by implementing it in SwiftAnalytics, a locality-aware object store based on OpenStack Swift [Swi16]. With locality-aware writes, we show that result data is written up to 8.5× faster, which speeds up jobs by a factor of 5.1×, while object-chunking provides up to 3.4× improvement during the reading of input data.

1.5 Related publications

This thesis is based on the following publications:


- **SwiftAnalytics: Optimizing Object Storage for Big Data Analytics.** Lukas Rupprecht, Rui Zhang, Bill Owen, Peter Pietzuch, Dean Hildebrand. In Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E), 2017 [RZO+17].


Additionally the following publications have been co-authored over the course of this thesis and have impacted but not directly contributed to the work presented here:


1.6 Dissertation outline

The remainder of this thesis is organised as follows:

Chapter 2 introduces the relevant technical background for the thesis and discusses the related work for each of the contributions. It first presents data centre networks and discusses approaches to improve their performance and manageability. The chapter then provides an overview of data-parallel frameworks with a focus on their interaction with the network. Alongside, it covers prior approaches for optimising network traffic during large-scale data analysis. The chapter finishes by discussing the storage layer of data-parallel frameworks. In particular, it compares block-based distributed filesystems with object stores and explains the advantages of object stores when used in combination with data-parallel frameworks.

Chapter 3 presents our in-network aggregation trees, realised in the NetAgg systems. It describes the design of the system both in terms of the distributed aggregation tree and the local aggregation nodes inside the tree. This includes the shim layers for transparent rerouting of traffic and the parallelisation of the aggregation function across many cores. The approach is then evaluated on a 16-node cluster with two data-parallel applications and in simulation.

Chapter 4 introduces SquirrelJoin. The chapter starts by formally introducing the problem of reassigning portions of data to different receivers during a distributed join and presents bounds on the effectiveness of such an approach. It then introduces the idea of lazy partitioning and how it can be employed by SquirrelJoin to dynamically react to network skew. This includes details on network monitoring, computing an optimal reassignment, and the implications on the join algorithm. The chapter finishes with an evaluation of SquirrelJoin under various real-world network skew scenarios and synthetic microbenchmarks.

Chapter 5 presents the SwiftAnalytics object store. It first provides a detailed study of the performance problems that arise when running data-parallel frameworks on top of an object store. The study reveals several problems related to the object store architecture. The chapter then continues to present the design of SwiftAnalytics to solve the uncovered problems. It introduces transparent object chunking and describes locality-aware writes for decentralised placement control. It finishes with a comparison of SwiftAnalytics to an unmodified object store and a block-based distributed filesystem.

Chapter 6 concludes the thesis by summarising its contributions and discussing directions for future research.
In this chapter, we present the relevant technical background and terminology that this thesis is based on. Section 2.1 starts by introducing the main concepts and properties of data centre networks in terms of their topologies, functionality, and management capabilities. The goal is to understand how modern data centres look today and what problems are associated with them. We also discuss recent advances that have tried to improve the performance and manageability of data centre networks from a networking point of view.

In Section 2.2, we introduce data-parallel frameworks. We first discuss their approach to large-scale data processing, including architecture, programming model, and resource management. After that, we focus on their interaction with the network. We look at the traffic patterns that different types of jobs incur and explain how data centre environments impact the performance during transfers. We also present existing approaches for optimising such traffic.

Section 2.3 focuses on distributed storage technologies that serve as input sources to data-parallel frameworks. This part introduces the basic concepts of distributed filesystems and compares them to object stores as a highly scalable alternative. We discuss both advantages and disadvantages of each technology and their specific architectures. In particular, we focus on approaches that avoid network transfers during storage accesses to increase data locality and present existing optimisations for both types of storage.
2.1 Data centre networks

Data centre networks (DCN) play an important factor in the design of a data centre. They not only ensure inter-connectivity between servers in the data centre but also provide a gateway to the Internet to expose services to the public. DCNs are expensive, and their cost significantly contributes to the overall cost of the data centre [PRI+10], accounting for 10–20% of the total equipment budget [GLM+08]. In this section, we look at the basic structure and topologies of DCNs and discuss how traffic is managed.

2.1.1 Structure and topologies

The most common structure for DCNs is a hierarchical tree topology with two or three layers, also called tiers [AFLV08]. The number of tiers determines how many servers can be interconnected in the data centre. Figure 2.1 shows an example of a 3-tiered topology. The bottom layer is called the edge layer as it contains the end hosts. End hosts are arranged in compute racks of usually 20–40 servers [GHJ+09]. Each rack has a single top-of-rack (ToR) switch, which connects the rack to the next layer, called an aggregation layer, via uplinks. The aggregation layer in turn consists of multiple aggregation switches, which each connects a certain number of racks to form a layer 2 broadcast domain. Finally, aggregation switches connect to core routers in the top tier of the hierarchy, which connect the different network segments. This layer is called the core layer. The aggregation and core layers together form the core network.

Bisection bandwidth and oversubscription

One of the most important properties of a DCN is its bisection bandwidth. Farrington et al. define bisection bandwidth as follows [FRV09]:

![Figure 2.1: A multi-rooted tree DCN topology](image-url)
2.1 Data centre networks

Definition. A bisection of a network is a partition into two equally-sized sets of nodes. The sum of the capacities of links between the two partitions is called the bandwidth of the bisection. The bisection bandwidth of a network is the minimum such bandwidth along all possible bisections.

Figure 2.2 illustrates this definition. The example network consists of 10 nodes and is split into two partitions of 5 nodes each. If we assume that all links have a capacity of 1 Gbps, the resulting bisection bandwidth is 2 Gbps. Intuitively, the bisection bandwidth of a DCN determines the maximum aggregate network throughput that can be achieved when all end hosts communicate with each other.

The bisection bandwidth is an important performance property of a DCN. As can be seen in Figure 2.1, the network tree has several core routers instead of a single one, i.e. the tree is multi-rooted. This allows the DCN to scale to a larger number of end hosts while still providing a high bisection bandwidth [AFLV08]. In the optimal case, the network provides full bisection bandwidth, i.e. arbitrary pairs of nodes can communicate with each other at the bandwidth provided by their NICs.

However, providing full bisection bandwidth is expensive [AFLV08]. As a result, data centre operators often oversubscribe the network [FA13] [BAM10] [GHJ+09]. The amount of oversubscription is expressed by the oversubscription ratio $1:x$ where $x$ denotes the factor by which the full bisection bandwidth is reduced [CDRO12]. Intuitively, oversubscription between two layers in a DCN means that the sum of the capacities of all uplinks at a layer is less than the sum of the capacities of its downlinks. For example, the network in Figure 2.2 has an oversubscription ratio of 1:1.5 as the full bisection bandwidth is 3, but the actual bisection bandwidth is only 2. In practice, oversubscription ratios range from 1:10 between the edge and aggregation layers [FA13] to up to 1:240 between the aggregation and core layers [GHJ+09].
While oversubscription enables cost savings, it decreases the available bandwidth and thereby the performance of applications in the data centre. This leads to a cost/performance trade-off.

Alternative topology designs

The described cost/performance trade-off in DCNs has a large impact on application performance. Hence, different topologies have been proposed to reduce cost while still providing full bisection bandwidth. We will discuss these topologies next. While these proposals come with advanced mechanisms to improve routing and load balancing, we will focus on their topological features. We classify the proposals into tree categories: Clos topologies, random topologies, and server-centric topologies.

Clos topologies. The fat-tree topology [AFLV08] is based on a Clos network [Clo53] and allows to build large-scale networks from commodity hardware, reducing cost. A $k$-ary fat-tree consists of $k$ pods, each pod containing $k^2/4$ end-hosts and hence $k^3/4$ hosts can be supported in total. Each pod has two layers of $k/2$ $k$-port switches, each of which is connected in a complete bipartite graph. The remaining $k/2$ free ports of the lower switch layer connect end hosts while those from the upper layer connect to the $(k/2)^2$ core switches. In a fat-tree, all switches have $k$ ports, i.e. no special hardware is needed in the aggregation or core layers. A downside of this design is that it incurs high cabling costs as it must connect $k^3$ ports.

VL2 is another DCN design that is build on top of a Clos topology [GHJ+09]. However, it does not use the fat-tree design but rather constructs a more general Clos network. The main difference regarding the topology is that VL2 does not use homogeneous $k$-port switches across the whole DCN but rather leverages faster switch-to-switch components to reduce the number of switches that is required in the aggregation and core layers. In the presented design, server-to-switch connections are 1Gbps while switch-to-switch connections in the core network leverage 10Gbps. This approach reduces cabling costs while still providing full bisection bandwidth.

Random topologies. Jellyfish takes a different approach in order to provide maximum extensibility [SHPG12]. Instead of a hierarchical structured topology, it constructs a random $r$-regular network graph on top of the ToR switches. In this design, each ToR switch uses $r$ ports to connect to other, randomly selected, ToR switches. This provides a high degree of flexibility to incrementally add more servers to the data centre. A Jellyfish network, even though constructed randomly, still offers a large capacity due to its shorter path lengths. If data has to traverse fewer links to reach its destination, more bandwidth from other links is available for different transfers.

Server-centric topologies. The BCube topology [GLL+09] is a server-centric topology. Servers are equipped with routing intelligence, which allows them to perform routing and traffic balancing. The server-centric topology requires only low-end switches with low port
counts, making it highly cost-effective. A BCube is a cube-like topology in which switches are never connected to other switches but only to servers. It is recursively defined by constructing a BCube$_k$ from n BCube$_{k-1}$ where a BCube$_0$ is a simple star topology. The topology was designed for modular, smaller-scale data centres, consisting of a few thousand instead of tens of thousands of servers. DCell [GWT+08] is the corresponding cube-like topology for large-scale data centres.

CamCube [ALCR+10] takes the server-centric design even further. It completely avoids switches for all internal network traffic by using a 3D torus in which each server directly connects to 6 other servers in the torus. While this can further reduce cost, a 3D torus does not provide full bisection bandwidth [Par99].

**Enhanced DCN designs**

To further reduce cost and improve performance, enhancements to the DCN have been proposed. They build on the assumption that data centre traffic is localised and rarely requires the entire full bisection bandwidth. Hence, they allow for oversubscription but offer additional on-demand capacity if hotspots are detected. We categorise the approaches into wireless extensions and optical networks.

**Wireless extensions.** Kandula et al. introduce the concept of flyways using 60 GHz high-bandwidth wireless technology [KPB09] [HKP+11]. ToR switches are enhanced with several wireless devices, which can be used to establish direct connections between racks and add additional capacity. This flexibility allows to provision more bandwidth on-demand when a ToR switch is detected as a hotspot while keeping the base wired network oversubscribed. While additional management complexity is added to make optimal use of these additional flyways, the approach is highly cost effective.

Zhou et al. improve the idea of wireless flyways by reducing signal interference and reliability [ZZZ+12]. The authors use 3D beamforming [ZZY+11], a technique to construct a wireless link between two antennas via an indirection in the third dimension. Beamforming radios can direct focused signals towards the ceiling from which they are reflected towards the receiver. This has the advantage that signal losses due to obstacles are avoided and interference is minimised as a third dimension is added to the signal paths.

**Optical networks.** Similar to the hybrid wired/wireless approaches above, other work has focused on integrating optical circuit switching into data centre networks to form a hybrid packet and circuit switched data centre network architecture (HyPaC) [WAK+10]. Compared to traditional packet-switched networks, optical circuit-switched networks offer large bandwidth capacity, on the order of tens of Gbps between two endpoints [FPR+10]. However, switching times are considerably longer taking on the order of milliseconds to establish a circuit between two different endpoints.
c-Through is one of the first to propose a HyPaC \cite{WAK+10}. It adds an additional circuit-switched network to an oversubscribed tree topology to interconnect the ToR switches. This adds additional and on-demand available capacity to each rack. However, the optical link can only be used to connect to a single other rack. Connections can be reconfigured but, as mentioned above, this incurs high latency. Hence, optical connections are only used for long-lasting, non latency-sensitive transfers while all other transfers are sent via the packed switched network. Helios presents a similar approach to c-Through. It is targeted at interconnecting small-scale modular data centres consisting of up to 1000 servers in a hybrid fashion \cite{FPR+10}. Again, the core network consists of both electrical packet switches and optical switches, and traffic is assigned to either one depending on its requirements, i.e. low latency or high bandwidth.

OSA is a fully optical architecture \cite{CSS+12}. In OSA, ToR switches are connected to $k$ ports of an optical switch instead of just one. This directly connects a ToR switch to $k$ other ToR switches instead of just a single one and allows to form a connected $k$-regular graph from the ToR switches. OSA can adjust the graph according to current traffic demands. As there is no packet-switched infrastructure anymore, OSA uses hop-by-hop stitching of circuits to provide all-to-all connectivity for latency-sensitive traffic. ToR switches can forward data to one of their available $k$ circuits and thereby stitch circuits together to reach racks that are not directly connected. This architecture provides more flexibility compared to the HyPaC architectures as any $k$-regular graph can be formed between ToR switches.

One of the most recent approaches, ProjectToR \cite{GMP+16}, employs free-space optics to build a wireless DCN based on optics. It uses digital micromirror devices (DMDs) and a mirror assembly on the data centre ceiling to connect arbitrary pairs of racks. DMDs provide low switching times in the range of tens of microseconds and hence rack connections can be reconfigured fast. ProjectToR uses a topology similar to the above hybrid structures in which long-lived connections are established between racks to form a base network. Additionally, opportunistic connections are created based on traffic demands, which may change frequently.

Discussion

While the above approaches can provide higher bisection bandwidth with lower cost, they come with several drawbacks. Full-bisection topologies such as fat-tree or BCube significantly increase wiring complexity and are hard to extend. For example, a fat-tree network can only support a specific number of servers depending on the switch port counts. Increasing the amount of servers would require upgrading most of the existing switches. Additionally, the network is over-provisioned for peak periods and hence can remain underutilised.

Wireless extensions avoid the over-provisioning of the network and can be deployed incrementally. However, they cannot sufficiently enhance the DCN to provide the required bandwidth.
for large-scale data-intensive applications [ZZZ+12]. Additionally, wireless technology depends on many external factors, which makes it less robust compared to wired networks. Optical links can provide more on-demand bandwidth, however, they face the challenge of longer switching times. Hence, additional management complexity is added in order to measure and predict traffic demands in the data centre and, based on that, reconfigure the network.

These drawbacks show that besides the advances in hardware and DCN design, it is important for applications to be aware of the specifics of the topology. If applications use the network in a more efficient way, network pressure is reduced, which results in higher performance and lower cost at the same time.

2.1.2 Management and programmability

The structured and static topology of a DCN differs from a decentralised and dynamic environment such as the Internet. This has opened up new opportunities in terms of managing the data centre network and adapting it to the needs of applications. In this section, we introduce the key concepts that are used in modern data centres to improve manageability via middleboxes and control via software-defined networking (SDN).

Middleboxes

Middleboxes are an important component of DCNs to extend the network with additional functionality to better manage traffic. Carpenter and Brim define middleboxes in RFC 3234 [CB02] as follows:

**Definition.** A middlebox is defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host.

This means that middleboxes sit between source and destination hosts and operate on network flows, i.e. the data that is sent from an application between a single source-destination pair. Middleboxes can implement many different network functions such as network address translation (NAT), tunnelling, load balancing, or firewalls and intrusion detection systems (IDS) [CB02]. Large-scale data centres contain thousands of middleboxes to improve the DCN in terms of performance and security. In most cases, data centres contain as many middleboxes as they contain switching and routing devices [SHS+12].

Traditionally, middleboxes have been hardware-based, proprietary appliances. This has several disadvantages including high cost, poor extensibility, narrow applicability, and heterogeneity [SRR+11]. As a result, new approaches have been developed that enable software-based middleboxes, i.e. implementations of network functions in software that are executed on
commodity servers, co-located with other network functions that share the available hardware. This approach is called network functions virtualisation (NFV) [Por12]. Various abstractions and frameworks have been proposed to ease the development of software middleboxes. We classify them into packet-based and flow-based abstractions.

**Packet-based abstractions.** These types of frameworks give developers a way of performing fast per-packet processing on a stream of packets, i.e., a network flow. Click is one of the first systems that provides a modular architecture for software routers [KMC+00]. In Click, a router is composed of different modules that can be connected in a directed graph. Packets are then processed by the individual modules along the graph. To reach the performance of hardware-based switches, **RouteBricks** extends Click to be scalable to multiple cores and servers [DEA+09].

Click does not support the consolidation of different network functions within a single server. To enable consolidation and hence, reduce cost and management complexity, software middlebox frameworks have been developed that support resource sharing. **CoMb** presents an initial design for a consolidation approach [SER+12]. It uses Click to implement processing and has a centralised network controller to optimally assign traffic to middleboxes under certain policies, e.g., a packet has to traverse middleboxes in a specific order. **ClickOS** provides stronger isolation guarantees than CoMb and enables multi-tenancy by running each network function in a virtual machine (VM) on the Xen hypervisor [MAR+14]. To reduce the overhead that comes with isolation at VM granularity, ClickOS is based on the lightweight MiniOS operating system, which is part of Xen. Again, network functions are implemented in Click.

Packet-based software middleboxes have also benefited from novel technologies that allow fast packet processing in user-space. Prominent examples are netmap [Riz12] [RCC12] and Intel’s Data Plane Development Kit (DPDK) [DPD16]. These frameworks perform highly efficient packet capturing by optimising the path a packet takes from the NIC to the application by, for example, using zero-copy mechanisms or entirely bypassing the kernel.

**Flow-based abstractions.** The above packet-based approaches allow for efficient customised packet processing. However, implementing more advanced network functions that operate on layers above TCP/IP and require to interpret application data is complex in these frameworks. Flow-based approaches expose network flows as a stream of bytes rather than a sequence of packets with headers. This is a more natural abstraction, which eases the implementation of application-layer functions such as HTTP parsing or application load balancing.

**xOMB** [ABK+12] offers such an abstraction. Similar to Click, xOMB consists of a pipeline of modules that processes streams from incoming TCP connections. Modules implement the application logic, such as transforming or parsing a stream, while xOMB handles low-level tasks such as setting up and tearing down connections or buffering the stream. Additionally, modules can store and access state across different modules and connections, which enables more
advanced functionality compared to stateless, packet-based approaches. FlowOS [BAM13] is a platform similar to xOMB. It runs a pipeline of processing modules as kernel threads through which flows are passed. Contrary to xOMB, FlowOS does not terminate TCP connections at the middlebox to prevent copying data between TCP connections and increase performance. Instead, it provides minimal TCP support for correct flow control.

With the introduction of flow-based abstractions, network functions also became more focused on application-specific processing. This means that they increasingly perform application-level computation such as caching, stream processing, or storage, and make the network part of the computing infrastructure [CMPW12]. SmartSwitch [ZWRH14] is an example for such a middlebox platform. It implements a load balancer for memcached, a key-value based distributed caching service, and supports in-network caching of requested key-value pairs. This brings data closer to end hosts and improves request latency. SmartSwitch is based on DPDK for fast packet processing. Another common approach is to use existing flow-oriented server implementations such as Apache [Apa16] or Nginx [Ngi16] to implement functions such as application-specific load balancing.

**Software-defined networking**

Besides middleboxes, data centre networks have sparked the development of a new technology that allows to program network devices and implement custom routing strategies. This concept is referred to as *software-defined networking* (SDN). According to Feamster et al. [FRZ14] SDN is mainly concerned with the separation of the control plane and the data plane of the network. The control plane is responsible for constructing routing tables and decide the path of packets through the network while the data plane is the actual logic that forwards data along those paths. Additionally, SDN has a consolidated control plane, usually implemented as part of a controller, that is able to control the forwarding devices in the entire network. While SDN has a long history that started with the development of active networks in the early 1990s, it only became practical and deployable recently with the advent of OpenFlow [FRZ14].

OpenFlow [MAB+08] is an architecture to implement the above separation of the control and data plane. It consists of OpenFlow switches, a controller, and the OpenFlow protocol specification and is based on the common use of flow tables in commercial switching hardware. A flow table defines actions that should be executed on a packet matching a specific flow such as forwarding it to a given port or dropping it. The OpenFlow protocol specifies how an external entity, i.e. the controller, can alter the flow tables of an OpenFlow switch by adding new matching rules. A rule describes one of the following four actions: (1) forward the packet to a specific port; (2) forward the packet to the controller; (3) drop the packet; and (4) forward the packet using the standard forwarding logic. The fourth action allows to isolate production traffic from research traffic and ensure compatibility with existing networks. As a
result, the OpenFlow protocol allows to implement custom network protocols for forwarding and routing without requiring to access the proprietary interfaces of commercial switches.

Since its inception, OpenFlow has been used and extended to implement more complex network control mechanisms and to provide more general and flexible abstractions. This includes higher level abstractions to ease the development of control applications, such as the network operating system NOX [GKP+08], and to support broader functionality by making packet processing protocol independent, such as done by POF [Son13] and P4 [BDG+14]. These developments are steering data centre networking towards a fully programmable model in which developers can freely implement network protocols and network functions on top of an operating system-like abstraction to guarantee a high degree of extensibility and flexibility.

Discussion

The advances in middleboxes and SDN give network operators an unprecedented level of control over their networking infrastructure. By stepping away from closed-source hardware-based systems and moving towards software-based approaches, tasks such as extending the network with complex processing functionality or deploying optimised protocols are now feasible. However, the presented approaches mainly focus on packet-based processing or provide only low-level flow abstractions. This restriction makes it difficult to realise higher-level processing in the network that requires application data.

Systems for implementing network functions face a trade-off between functionality and performance. If packets need to be assembled and deserialised to reconstruct application-level data types, performance necessarily suffers. Hence, the above approaches mainly expose packet-based APIs in which limited processing can be performed on the headers and payloads of single packets. The discussed flow-based abstractions improve on this but still only implement a small set of network-level services to be able to maintain high performance.

This shows that existing approaches are not sufficient to implement a network-aware data-parallel processing system. To effectively support applications, network functions must be able to interpret application data types and perform complex processing on them while still maintaining high performance.

2.1.3 Traffic characteristics

Data centre network traffic is composed of a variety of flows from different applications occurring in different traffic patterns. While traces from production clusters are scarce, some existing studies try to characterise traffic in real-world data centres [BAM10] [KSG+09] [GHJ+09]. Next, we summarise these characteristics and explain the specific problems that arise due to them in DCNs. We also discuss existing solutions to these problems.
2.1 Data centre networks

Flows sizes and burstiness

Data centre traffic is a mix of small, short flows and large, long flows called *mice* and *elephant* flows, respectively. Greenberg et al. found that in a Microsoft data centre that is used for data analysis, the majority of flows are mice flows, i.e. 99% of all flows are smaller than 100 MB. However, 90% of all data is transferred in elephant flows, larger than 100 MB [GHJ+09]. This finding is consistent with a study by Benson et al. who analysed the traffic across a variety of data centres and report that 80% of flows are smaller than 10 KB but most of the bytes are transferred by the largest 10% of flows [BAM10].

Traffic is also highly variable and hard to predict. While the two studies examine different data centres, both find a high variability in packet inter-arrival times, traffic demands, and participating servers [BAM10] [KSG+09]. Benson et al. found that packet inter-arrival times have very distinct ON/OFF periods, i.e. periods in which the inter-arrival times are smaller (ON) and larger (OFF) than its 95th percentile [BAM10]. The length of ON and OFF periods can be modelled by heavy-tailed distributions, in particular, a log-normal distribution, which provides the best fit in most cases. This shows that traffic is bursty. The traffic in the Microsoft data centre also exhibits a high degree of instability [GHJ+09] [KSG+09]. Clustering traffic matrices over a day of measurement data does not yield any predictable pattern nor a small set of representative matrices that can model all traffic. Additionally, even in periods during which the overall traffic demand does not vary significantly, the servers that are involved in data exchanges still change frequently.

The mix of small and large flows and the burstiness of traffic lead to the problem of *queue buildup* and *buffer pressure* [AGM+10]. A switch has packet queues for each outgoing port. To buffer the packets in those queues, most commodity switches use a shared buffer pool to reduce hardware cost. Large flows require a large amount of buffer space to achieve high throughput which can lead to high latencies for small flows due to two reasons: first, small flows may experience packet loss as all buffer space is utilised by the large flows. As a result, packets have to be retransmitted; second, in case of no packet loss, small flows may still experience increased latency due to longer waiting times at switches (queue buildup). These problems are likely to occur during burst periods when traffic demands increase rapidly in a short time. The shared buffer memory intensifies this effect as packets do not even have to be queued at the same outgoing port to be impacted by the high memory demands of the larger flows (buffer pressure). As small flows are usually initiated from latency-sensitive applications, the responsiveness of these applications will decrease. Several solutions to the problem of high tail-latency in data centres have been proposed in the literature.

**ECN-based approaches.** Data centre TCP (DCTCP) [AGM+10] is a novel congestion control mechanism for TCP that uses Explicit Congestion Notification (ECN) [RFB01] to react to congestion before actual packet losses occur. ECN allows to notify endpoints when queues
build up and packet loss is likely to happen. Notification is done by setting an ECN mark, i.e. a bit in the IP header. DCTCP uses this mechanism to extract a multi-bit feedback stream from the ECN marks and adjusts TCP’s congestion window proportionally to the number of marked packets. This proportional reaction allows DCTCP to keep throughput high for flows but prevent queue buildup at switches. Other work has extended the ECN-based, proportional congestion control of DCTCP to, for example, add deadline-awareness to flows [VHV12] or reduce queueing-incurred latencies to near zero by keeping the amount of utilised bandwidth slightly below the maximum available bandwidth [AKE+12].

**Rate limiting-based approaches.** Other approaches use rate limiting at end hosts to bound the size of switch queues and thereby achieve predictable delays. These approaches are based on the fact that rate limiting senders yields a deterministic upper bound for the length of queues at switches [PG93] [PG94]. *Silo* [JSBM15] uses this property and combines it with delay-aware VM placement in multi-tenant clouds to bound transmission and queueing delays. The placement algorithm ensures that VMs are spread across enough switches such that bursty traffic to a single VM cannot overflow switch buffers and cause packet loss. *QJUMP* [GSG+15] uses a similar reasoning but introduces priorities that trade off throughput for latency. Latency-sensitive applications can request higher priorities to reduce queueing delay but will be rate-limited more while throughput-sensitive applications receive lower priorities but less rate limiting. Priorities are supported by switches, which allow high priority packets to *jump* queues, leading to low, predictable latencies.

**Background traffic and network skew**

In most cases, data centres host a variety of applications that generate traffic and utilise the network. If different flows from different applications use the same paths in the DCN, the available bandwidth has to be shared across all flows. This leads to network congestion and lower throughput for all applications and causes interference. Traffic that influences an application but is not initiated by that application is called background traffic [AGM+10]. Background traffic is a mix of different flows coming from other queries, distributed file system activities, and control traffic such as monitoring or synchronisation. In a Microsoft data centre, a machine sees around 10 concurrent incoming and outgoing flows in the median [GHJ+09]. However, there’s a long tail and 5% of the time, machines see more than 80 concurrent flows. Results from another cluster report a median number of concurrent flows of 36 with a 99.99\textsuperscript{th} percentile of over 1600 [AGM+10].

The shared nature of data centre networks increases network congestion as more applications demand bandwidth. Kandula et al. found that in the Microsoft data centre, congestion is common [KSG+09]: they found 665 episodes of congestion during a single day where 86% lasted at least 10 s and 15% lasted over 100 s. Benson et al. [BAM10] report similar patterns
2.1 Data centre networks

across their 10 studied data centres: in one deployment, a small subset of network links are hotspots, defined as using at least 70% of their bandwidth, 90% of the time. Hot spots were most severe in the data centres that ran data analysis jobs. In these, approximately 20% of the links in the network core experience congestion for more than 50% of the time. An interesting observation from both studies is that longer lasting congestion periods do not affect network links evenly but rather are localised to a small subset of links. We term this behaviour network skew, i.e. parts of the network experience a higher utilisation than others.

Two problems arise from the above observations: first, background traffic reduces the available bandwidth for applications as it now has to be shared. Most applications use TCP as a transport-layer protocol, which implements max-min fairness to equally split the bandwidth across flows. For example, if an application has to share a link with another 9 applications, each of them initiating one flow, then the available bandwidth for each flow will drop to 1/10. This increases the duration of transmissions and in turn job completion times; second, as congestion is often skewed, only parts of the links are affected. This causes some nodes to have slower transmissions and fall behind, i.e. they become stragglers and stall the entire job. Similar to the above described queue buildup, this can lead to high tail latencies and low responsiveness. To overcomes these problems, researchers have proposed many solutions, which we classify into flow scheduling and network isolation.

**Flow scheduling.** Flow scheduling approaches try to make informed decisions to determine the optimal path and time to run for a flow. *Hedera* [AFRR+10] proposes a dynamic flow scheduling method that computes flow-to-path assignments that maximise utilisation and minimise interference. Assignments are computed based on flow demands, which are estimated based on a flow’s allocated max-min bandwidth. It uses a simulated annealing heuristic to compute the actual assignment and OpenFlow-enabled switches to implement the assignment using forwarding rules in the DCN. Using a guided approach, Hedera can reduce path interference and achieves better overall utilisation than random schemes such as Equal-cost Multipath Routing (ECMP) [Hop00] or Valiant Load Balancing (VLB) [GHJ+09].

The *coflow* abstraction [CS12] proposes to view network traffic not as individual flows but rather as collections of flows in which each collection belongs to a single application such as a MapReduce job. *Varys* [CZS14] then uses this abstraction to schedule coflows rather than single flows and improves coflow completion times significantly. Varys uses a centralised scheduler design and assumes that coflow properties, such as number of flows and sizes of flows, are known in advance. *Aalo* [CS15] relaxes this assumption and uses a modification of the least-attained service heuristic to make scheduling decisions based on the amount of data sent per coflow. The more data has been sent from a coflow, the lower the priority becomes. This allows Aalo to effectively schedule coflows without knowing their specific properties.
Network isolation. Isolation approaches attempt to isolate bandwidth in DCNs to reduce network interference between different applications/tenants in a shared cloud data centre. This is similar to the local isolation of CPU and memory as done by virtual machines. Oktopus [BCKR11] introduces a virtual network abstraction which allows tenants to choose between a virtual cluster (no oversubscription) or a virtual oversubscribed cluster topology and specify a bandwidth requirement. A centralised network manager then allocates VMs to physical machines, making sure that enough bandwidth is available between pairs. Local enforcement modules in the hypervisor make sure that VMs do not exceed their requested bandwidth. Hadrian [BJK+13] extends the problem of network isolation to both intra- and inter-tenant traffic. It uses hierarchical bandwidth requirements and explicit tenant traffic dependencies to allow tenants to specify their bandwidth needs.

CloudMirror [LTL+14] moves to a more application-driven way of specifying bandwidth requirements. Instead of allowing tenants to choose from certain network topologies, it uses tenant application graphs (TAGs). A Tag represents the different components of an application as vertices and edges represent network communication between these components. Edges can be annotated with bandwidth requirements, and TAGs are then mapped to the physical topology, again, using a VM placement algorithm and a rate limiter to enforce bandwidth constraints. The TAG model makes it easier for tenants to determine their network requirements as they do not have to deal with details about the physical network anymore.

Discussion

The above properties of data centre traffic show that its diversity and unpredictability is challenging for existing network protocols and applications. The presented approaches, while significantly improving on the status quo, still cannot fully solve the problems. Solutions to prevent queue buildup and buffer pressure that are based on DCTCP require ECN-enabled switches. However, not all switches ship with ECN support. Rate limiting solutions on the other hand have to trade some of the available bandwidth to guarantee limited queue lengths, and hence leave a small fraction of resources unutilised.

Approaches to mitigate the effects of background traffic and network skew also have several drawbacks. Flow scheduling approaches are usually centralised. However, flow arrival rates in DCNs are high and hence schedulers have to make many decisions in a short time frame, which makes it challenging to scale [KSG+09]. Additionally, these approaches make static a priori decisions that do not account for future traffic, i.e. decisions can still be suboptimal if patterns change. Network isolation mechanisms prevent interference in the first place, however, they only work between tenants. If a single tenant has a variety of deployed applications or the cluster is not virtualised, interference can still occur. Furthermore, specifying bandwidth requirements is challenging as traffic is bursty and peak demands can change.
Hence, sole network-level solutions can still benefit from application-level awareness. If applications can minimise their traffic or are able to react to background traffic themselves, they can complement solutions on the network-level and improve their performance and overall network utilisation.

2.2 Data-parallel frameworks

After discussing data centre networks, we consider data-parallel frameworks. Data-parallel frameworks such as MapReduce [DG04], Dryad [IBY+07], Spark [ZCD+12], and Flink [Fli16a], have become a popular choice for performing large-scale data analysis. Their scalability and simple API allows non-expert users to implement complex queries and deploy them on hundreds or thousands of machines. We will first look at how these frameworks achieve these two properties and then describe the interaction of different queries with the network in detail.

2.2.1 Operating principles

Data-parallel frameworks have become the number-one choice for performing a large variety of data analysis tasks. These tasks range from batch processing such as ETL-like transformations [DG04] [PPR+09] and graph processing [MAB+10] [GLG+12] over interactive processing of SQL queries [AXL+15] [KBB+15] and data streams [MMI+13] [CFMKP13] to advanced analytics using machine learning [ABC+16]. With the recent explosion of data availability due to easier data collection and cheaper storage, processing these large amounts on a single machine became infeasible. Data processing hence shifted to a vertical scaling model in which compute and storage capacity is added to a cluster by adding more machines (see §1.1). Data-parallel frameworks allow to easily program those clusters to perform data analysis tasks in parallel on the available hardware.

Data-parallelism

As their name suggests, data-parallel frameworks exploit data-parallelism to parallelise computation across multiple machines and CPU cores of a single machine. In data-parallelism, the input data is partitioned across a set of nodes and each node runs tasks to process the input data simultaneously. Tasks are equivalent and execute the same processing logic [HS86]. The granularity of tasks depends on different factors, and there is no standard way of finding the optimal number. In MapReduce, launching a task has significant overhead and hence, longer running tasks are preferable [XRZ+13]. On the other hand, in systems such as Spark or Flink, task creation is efficient. Hence, a larger number of small tasks should be used as that increases fairness and reduces the impact of stragglers [OWZS13]. As an established
rule of thumb, a single task should process between 16 MB and 64 MB [DG04]. However, this parameter may need tuning for different queries. Figure 2.3 shows an example with 3 nodes. Each node has an input partition, either stored locally or retrieved remotely, which it processes using 4 tasks.

Data-parallel frameworks usually employ a shared-nothing architecture [Sto86]. That means that individual nodes are independent of each other as they do not share CPU, disk, or memory resources. This allows to scale the system by adding more nodes, which can then process the same input data faster or a proportionally larger set of input data in the same time. In Big Data analysis, the limiting factor is often the I/O required to read the large amounts of input data. Data-parallel frameworks split this work across many machines and hence achieve high performance and scalability with huge data sizes.

Operator APIs and dataflow graphs

Data-parallel frameworks expose a simple API that allows users to program queries which are then automatically parallelised and executed on the cluster. The APIs are based on operators that provide the building blocks for queries. Operators are higher-order functions that execute a specified computation on items of input data. They can either have predefined semantics or implement user-defined functions (UDF). Examples of predefined operators are join, group by, or distinct, while examples of UDF operators are map, filter, or reduce. Operators can also be viewed as transformations on the input data [ZCD+12]. To program a query, operators can be chained together to form an operator pipeline. The first operator applies its function to the raw input data while each subsequent operator processes the output of its predecessor.

Most data-parallel frameworks operate on a key/value data model. Input data is often only semi-structured or unstructured. Hence, users with domain knowledge define the parsing logic to read single data items and split them into key and value. For example, a data item (or record) could be equal to one line of an input text file, and the key and the value are separated
by a ‘;’ symbol. Operators then perform their processing on each single key/value pair. For more specific tasks such as SQL- or graph-processing, in which input data is structured, e.g. relational or as a matrix, frameworks provide extensions that natively support these data models [TSJ+09] [XRZ+13] [AXL+15].

Figure 2.4 shows an example of a query. The operator pipeline consists of 3 operators, a filter, a group by, and a sum. Key/value pairs are parsed from the input data and passed to the first operator. Note that if the input format is not integrated with the framework, the parsing has to be performed by a separate map operator. The output of the filter operator excludes all key/value pairs with a key equal to k2 and is then grouped by the key and, finally, the sum of all values for each group is computed. Listing 2.1 displays the concrete code for the example query as it would be programmed using Apache Flink’s functional API [Fli16b].

First, the dataset is read, assuming that records are split at a line, and key and value are split by a comma. Then the operators are chained together and applied to the input. We can see that the filter operator requires a function as an argument to specify its exact semantics, while group by is predefined, and aggregate only requires the aggregation function.

The logical user view as shown in Figure 2.4a is equal to the pipeline being executed on a single node. To be run on multiple nodes, the framework duplicates the pipeline and executes it in parallel on different input data partitions across all nodes in the cluster. This forms a

![Logical and Physical Operator Pipelines](image)

**Figure 2.4:** Example for logical and physical operator pipelines

### Listing 2.1: Code for the example operator pipeline using Apache Flink’s scala API

```scala
val input = env.readCsvFile[(String, Int)]("input/file").as('key', 'value')
val counts = input.filter { item => item.key != 'k2' }
    .groupBy("key")
    .aggregate(Aggregations.SUM, "value")
counts.print()
```

51
Chapter 2. Background

dataflow graph as shown in Figure 2.4b. A dataflow graph is a directed acyclic graph (DAG) consisting of several stages. Each operator now consists of multiple tasks and represents one stage. Stages are connected via directed edges in predefined patterns that depend on the semantics of the operator. There are two main patterns, the one-to-one and the all-to-all or shuffle pattern. In the one-to-one pattern, there are only connecting edges between the subsequent tasks on a single node, while, in the shuffle pattern, each task from the previous layer is connected to all tasks from the next layer in a full bipartite graph.

Which connection pattern is used depends on the operator. Some operators can be executed independently on a partition of data. For example, the tasks of a filter operator can apply the filter function to each data item in their input partition in isolation and the result will still contain only correct data items. Operators such as sum or join on the other hand require the exchange of data in a shuffle to produce the correct result. Consider the sum operator from the example query, which sums up all values for each individual key. If different input partitions contain key/value pairs with equal keys, the corresponding values have to be added up by a single node, otherwise each node produces only a partial sum. As can be seen in Figure 2.4b, the data exchange requires communication between tasks/nodes and hence incurs network communication. As a result, operators that require data shuffling are impacted by the network performance. We will discuss these operators in more detail in §2.2.2 and §2.2.3.

Resource management and fault tolerance

Data-parallel frameworks do not only automatically parallelise queries but also take care of scheduling tasks across a set of available machines and dealing with failures. We discuss these two aspects next.

Resource management. When a query is submitted, its tasks have to be assigned to nodes for execution. Large clusters have a multitude of hardware resources available to run the tasks of a data-parallel job. However, to maximise utilisation and minimise cost, clusters are usually shared between different applications which each have different resource requirements at different points in time [HKZ+11]. This can create interference between different data-parallel jobs. To ensure a job gets all the resources it needs to complete successfully and does not interfere with already running jobs, tasks have to be scheduled. In earlier versions of MapReduce, scheduling was performed by the system itself [DG04]. However, to decouple data processing from control, cluster schedulers have been developed that now take care of the task scheduling for a variety of different frameworks simultaneously [HKZ+11] [VMD+13] [SKEM13].

Two of the most popular cluster schedulers are Mesos [HKZ+11] and Yarn [VMD+13]. Both systems share a similar architecture and are designed to schedule tasks from different frameworks. Mesos consists of a resource master that controls a set of slave nodes and their available resources. The master can make resource offers to frameworks based on certain
2.2 Data-parallel frameworks

policies such as fair sharing. Frameworks can reject these offers and wait for better ones if they do not fulfil their requirements. Yarn also has a master node to communicate with different frameworks and allocate resources for them. However, instead of receiving offers, frameworks request resources, and Yarn tries to find a node that matches the specified request. Both frameworks use containers to achieve actual resource isolation [Lin16] [Cgo16]. Containers are lightweight as they use kernel features to split resources between applications instead of requiring a full operating system on top of a hypervisor, as is the case for virtual machines.

Besides achieving fairness and meeting the resource requirements of different jobs and frameworks, another common and important scheduling goal is data locality [ZBSS+10]. The aim is to move the computation to where the data is, i.e. the task that is responsible for a certain input partition is preferably scheduled on the node that physically stores this partition. This avoids additional network traffic due to remote fetches of data and can significantly improve performance when networking resources are limited. While initial scheduling policies followed a best effort approach to simultaneously maximise data locality and fairness, more sophisticated policies have been developed. For example, delay scheduling [ZBSS+10] trades latency for locality. In case a task cannot be assigned to the data local node, this policy delays the scheduling decision for a short amount of time to see if resources on the required node are freed, and the task can be assigned. Otherwise, the task is scheduled on a remote node and has to retrieve the input data over the network.

Fault tolerance. As clusters are scaled out and more hardware is added, failures are common rather than an exception [Dea08]. Data-parallel frameworks are designed to transparently and efficiently deal with failures and still correctly execute jobs. MapReduce uses a simple fault tolerance mechanism that reruns failed map tasks and recomputes their output data [DG04]. The shared-nothing design and the fine-grained task-based computation allows to only recompute the failed task(s) rather than the entire query enabling efficient fault recovery. Spark introduces a more advanced recovery mechanism based on lineage for longer operator pipelines. Lineage tracks the operators that have been applied to the input dataset instead of materialising intermediate results. This avoids the need for checkpointing while still enabling fast fault recovery. Again, only the portions of the result data of failed tasks have to be recomputed. Recomputation can even be performed in parallel by dividing the input of the failed task between more tasks.

As another form of fault-tolerance, data-parallel frameworks also try to reduce the impact of straggling tasks. A widely adopted approach is speculative execution [DG04]. Data-parallel frameworks periodically receive progress estimates of running tasks. If a task does not make enough progress, a duplicate or speculative task is launched, which processes the same input data and now competes with the original task. The output of the "winning" task is used for further processing while the "losing" task is killed. This approach has been extended by proposals such as Mantri [AKG+10] and Dolly [AGSS13]. Mantri performs more sophisticated
progress estimation to perform a guided placement of speculative tasks while Dolly runs copies of entire jobs. While these approaches improve the effectiveness of straggler mitigation, they fundamentally follow the concept of executing duplicate tasks.

After explaining the basic functionality of data-parallel frameworks, we will now look at the operators that cause network traffic in more detail. We start by discussing aggregation operators and then explain join operators.

### 2.2.2 Partition/Aggregate queries

Aggregation queries, such as computing a sum or the top-$k$ elements, are common in data-parallel frameworks. Usually, the input data records are grouped by their key, and the aggregation function is applied over all values of a group. To correctly compute the aggregation function, nodes have to exchange data to make sure that all key/value pairs with the same key are grouped and aggregated by a single node. Aggregation queries are often part of a partition/aggregate pattern.

In the partition step, a job is divided into independent sub-tasks, which are executed in a data-parallel fashion by different worker nodes. Each node operates on its input partition and generates a partial result. In the aggregation step, all partial results are collected by one or few master nodes, which group all key/value pairs and apply the aggregation function to produce the final result. One example for this pattern are web requests such as search queries. An incoming search query is replicated by the web frontend and sent to multiple backend servers in parallel, each hosting a small portion of the entire web index. Each backend processes the query locally and returns the top-$k$ results best matching the query. The partial results are aggregated by the frontend to select the overall top-$k$ elements and returns them to the user. Another example is MapReduce in which the input data is processed by the map tasks in parallel. The output of each map tasks is then sent to one or many reduce tasks which aggregate the individual map results. Other frameworks, e.g. for graph [MAB\textsuperscript{+10}] or stream processing [CFMKP13], work in a similar way.

#### Network congestion during partition/aggregate queries

The partition/aggregate pattern is challenging for the network because during the aggregation step, all worker nodes send their partial results to the few aggregator node(s) (see Figure 2.5a). As can be seen from the figure, this creates congestion at the master node if its incoming bandwidth is less than the combined outgoing bandwidth of the workers. If the network is oversubscribed, this congestion already happens earlier at the different layers of the data centre network. As a result, worker nodes have to reduce their sending rate and are not able to fully utilise the available outgoing bandwidth.
Typically, the aggregation functions used in such queries exhibit two properties. First, they are associative and commutative [YGI09]. That means that the aggregation step can be performed through a sequence of partial aggregations without affecting the correctness of the final result. Examples of aggregation functions with this property include max, sum and top-\(k\). While there are some functions that do not satisfy this property, e.g. median, it does hold for many aggregations, especially those used in analytics queries [MGL+10] and graph processing [YGI09]. Second, in most cases, the aggregation function reduces the size of its input, i.e. the resulting aggregated data is smaller than the combined size of the inputs. For example, the average final output data size in Google jobs is 40% of the intermediate data sizes [DG04]. In Facebook and Yahoo, the reduction in size is even more pronounced: in 82% of Facebook MapReduce jobs with a reduce phase, the final output size is only 5% of the intermediate size, while for Yahoo jobs the number is 8% in 91% of the jobs [CGGK11]. Similar trends also hold for traces collected in Microsoft Scope, which show a reduction factor of up to two orders of magnitude between the intermediate and final data [AGN+13].

These two properties can be exploited to perform early data reduction by aggregating partial results on worker nodes. We call this approach edge-based aggregation. For example, MapReduce comes with the option of combiner functions that already apply the (associative and commutative) reduce function to the map output of each mapper before sending out the results. This decreases the amount of data that has to be shuffled across the network [DG04]. Other work on interactive services [MGL+10] and dataflow frameworks [LTWY11] [YGI09] proposes strategies to reduce inter-rack traffic through rack-level aggregation (see Figure 2.5a): one server per rack acts as an aggregator and receives all intermediate data from the workers in the same rack. The chosen server aggregates the data and sends it to another server for final aggregation. Rack-level aggregation can be generalised to form a \(d\)-ary tree of servers that aggregate data within a rack first and then progressively across racks. Intuitively, a lower \(d\) leads to a higher maximum transmission rate per worker. This includes the case of \(d = 1\), when the tree becomes a chain (see Figure 2.5b).
Chapter 2. Background

Discussion

Edge-based aggregation helps to reduce network traffic during partition/aggregate queries. However, its main drawback is that its performance is still limited by the inbound bandwidth of the aggregator. For example, assuming 40 servers per rack and 1 Gbps edge network links, the maximum transmission rate per worker is only approximately 25 Mbps. $d$-ary trees can eliminate some of the shortcomings of rack-level aggregation but introduce new challenges. First, small values of $d$ increase the depth of the tree. As data now has to traverse more hops before reaching the final aggregator, latency increases. Second, small values of $d$ increase intra-rack bandwidth usage because now, incoming links of workers, as opposed to only outgoing links, are used to move data. This can drastically reduce the performance of other network flows that cannot be aggregated.

A fundamental drawback of any edge-based aggregation approach is that it only applies to intra-rack traffic and not traffic in the core of the network. As shown in Figure 2.5c, if aggregation computation spans multiple racks, the links between aggregation switches can become a bottleneck, especially in the presence of oversubscription. A network-aware application could exploit the tree topology of modern data centre networks to execute additional partial aggregation steps at intermediate hops to further reduce the data.

2.2.3 Join queries

The other type of operator that requires data exchange between tasks are joins. Joins are similar to aggregation operators as the correctness of the result also requires that all records with the same key are joined by the same node. Otherwise, not all join matches are computed.
Figure 2.6 shows an example for a distributed equi-join, i.e. the join predicate is equality, of tables R and S. The dataset stored across the cluster nodes is repartitioned based on the join key, and workers on nodes are assigned partitions to process. Each worker therefore sends its local data to a set of other workers for processing. This repartition join is a common join implementation in data-parallel frameworks [BPE+10] and results in the highest network utilisation because the entire input tables must be shuffled across the network.

A common repartitioning function is hash repartitioning, i.e. keys are hashed and the resulting value determines the target node. If all nodes use the same hash function, the join result is guaranteed to be correct. There are two classes of hash-based join algorithms: asymmetric and symmetric. Asymmetric joins, such as the GRACE [KTMO83] hash join and the hybrid hash join [DKO+84], operate in two phases to optimise memory usage. First, in a build phase, workers hash the records of the smaller table and send them to receiving workers, which each construct a local hash table. After that, in a probe phase, workers hash the records from the larger table and send them to the receiving workers that contain the corresponding keys in their hash tables for joining. In contrast, symmetric hash join algorithms such as XJoin [UF00] do not progress in phases but rather maintain two hash tables at each worker node, one for each of the input tables. As described above, distributed joins are stateful, i.e. the correctness of the join result depends on which join keys the individual workers process. In the case of a hash join, the state is comprised by its hash tables.

The network traffic caused by joins is different from the traffic from aggregation operators as it can neither be aggregated nor partial results be computed locally. Additionally the result dataset can even be larger than the input. Hence, none of the described network optimisations for aggregation queries can be applied. To optimise network traffic in distributed joins, two basic strategies exist: semi joins and broadcast joins [BPE+10]. During a semi join, only the join key column of the larger relation (assume S) is shuffled across the network and a distributed semi join is computed with the smaller relation (assume R). The result of this semi join contains only the records in R that have a match in S. It is then sufficient to shuffle those records across the network and avoid sending records that do not have a match in S. If the total size of the join key column of S and the reduced size of R is less than the non-reduce size of R, less network traffic is generated. If $|R| \ll |S|$, it can be beneficial to broadcast R to all nodes instead of repartitioning and shuffling both R and S. This is called a broadcast join.

**Skew in distributed join processing**

The completion time for a distributed join is determined by its slowest worker. Hence, the performance of a join is heavily influenced by stragglers and reducing the join completion time requires minimising the completion time of the heaviest straggler. There are two main reasons that lead to stragglers during a distributed join: data skew and resource skew.
Data skew. Data skew occurs when the distribution of the join key across all records is heavy-tailed instead of uniform. That means that there are few keys that occur very frequently, also called heavy hitters, and many keys that only occur rarely. The workers which process the heavy hitters are likely to become stragglers as they have to receive more data over the network and have to join more records. Approaches to address data skew can be divided into static and dynamic techniques.

Static techniques [PSR14] [RMU+14] [RIKN16] attempt to apply more balanced partitioning functions that account for the skewed data distribution. NeoJoin [RMU+14] collects the key distribution across different nodes to determine an optimal partitioning that equalises the amount of data each node has to send and receive during the shuffle. Additionally, it schedules the traffic such that all links are always fully utilised. TrackJoin [PSR14] uses a separate tracking phase during which the optimal join location, i.e. the node that computes the matches, is determined for each key such that network communication is minimised. These two techniques both require a preprocessing step to gather statistics on the key distribution before determining an optimal partitioning. To minimise this preprocessing overhead, FlowJoin [RIKN16] samples a small subset of the data to reliably determine the heavy hitters and adapt the partitioning. This is similar to situation-aware mappers [VBBE12] for MapReduce, which sample an initial histogram and determine an optimal partitioning.

Dynamic techniques react to data skew as it develops and then change the work assignment on-the-fly. SkewTune [KBHR12] dynamically assigns work to nodes that finish their tasks earlier than others and are idle. This assumes that keys are ordered and new keys can be assigned to arbitrary workers. More general solutions migrate already partitioned data between nodes at runtime. Eddies [AH00] act as tuple routers: they change the order in which tuples are processed by operators, essentially reordering the query plan at runtime. To undo bad routing decisions, Deshpande and Hellerstein extend eddies to modify and migrate operator state [DH04]. The Flux operator [SHCF03] balances transient data skew. By keeping a single buffer per receiver at the senders, Flux changes the arrival order of incoming tuples and thereby avoids temporary bursts to receivers. To address persistent skew, Flux also migrates state. Elseidy et al. [EEVK14] focus on adaptive theta joins, i.e. joins with arbitrary predicates, with provable bounds for state migration cost to minimise the migration effort.

Resource skew. The second type of skew is resource skew, in which node resources used by workers such as CPU, disk, or network are not uniform across the cluster (see also §2.2.1). In practice, resource skew is addressed by isolating resources using cluster resource managers, such as Mesos [HKZ+11] or Yarn [VMD+13]. These can provide workers with guaranteed resources through containers to prevent stragglers.

More sophisticated approaches deal with resource skew directly. Resource Bricolage [LNN14] determines the properties of the available (heterogeneous) nodes and the workload, and
2.3 Scalable storage technologies

formulates an optimisation problem to fully utilise all available hardware. Xiong et al. [XHN14] focus specifically on network skew, i.e. the case in which the network utilisation is uneven across nodes and hence leads to stragglers. The authors use SDN-based network monitoring techniques to identify congested regions of the network and adapt the query plan accordingly. Some of the above approaches such as Flux [SHCF03] or by Elseidy et al. can also identify resource skew and repartition the data accordingly using their state migration-based approach.

Discussion

Distributed joins are an expensive operation in terms of network utilisation. Due to their nature, only limited traffic reduction mechanisms can be applied. While the presented approaches are able to successfully deal with data and general resource skew, they have several shortcomings, making them less applicable to network skew.

Data skew is a static property that will hold throughout the computation of the join. Once the relevant statistics about the dataset are obtained, an optimal a priori decision can be made that will remain valid for the entirety of the join. This is inherently different to network skew, which is caused by dynamic background traffic. Accurately predicting when network skew will appear and disappear and how much the influence on the join will be is a difficult problem. Hence, an optimal partitioning that accounts for network skew cannot be determined before executing the join.

Existing techniques to prevent stragglers due to resource skew are also not applicable to network skew. Resource managers such as Mesos or Yarn provide mechanisms to isolate CPU and memory resources but do not isolate network bandwidth. Dynamic approaches such as Flux can react to network skew and adapt the partitioning accordingly but require state migration. Under network skew, an approach that migrates data between workers is not possible because the network itself is the cause of the skew and thus a constrained resource. The approach by Xiong et al. [XHN14] that specifically targets network skew only allows to change data transfers between operators in a query but not within a single operator.

As a result, distributed joins are susceptible to network skew, which causes stragglers and slows down the entire join computation. A network-aware join could plan for potential network skew from the start in order to react to it without having to migrate state.

2.3 Scalable storage technologies

So far, we covered data centre networks and how data-parallel frameworks interact with them. In the last part of this chapter, we will discuss the distributed storage layer, from which data-parallel frameworks read input data and to which they write the results of a job.
Again, we focus on the potential network traffic that can occur when data-parallel frameworks interact with distributed storage. The two main architectures that we discuss are block-based and object-based distributed storage.

### 2.3.1 Block-based storage

When Google designed MapReduce, they co-designed the Google File System (GFS) [GGL03], a scalable distributed file system that serves as a data repository for MapReduce. When Hadoop MapReduce was released, it also came with a distributed storage solution, the Hadoop Distributed File System (HDFS), modelled after GFS [SKRC10] [HDF16]. Currently, HDFS is widely deployed as the primary storage layer for MapReduce and is supported by almost all data-parallel frameworks, such as Spark [ZCD+12], Flink [Fli16a], or Impala [KBB+15]. In the following, we will explain the basic design and functionality of HDFS.

#### HDFS Basics

HDFS is a block-based distributed file system. Like a block device such as a hard disk, HDFS stores files as a set of blocks. The default block size is 128 MB. If a file is larger than the block size, HDFS automatically splits the file and stores each block separately. HDFS follows a master/slave architecture as shown in Figure 2.7. The master is called NameNode and stores all the filesystem-relevant metadata such as a block map, which keeps the locations of all blocks for every file stored in the file system, directory hierarchies, and permissions. The slaves are called DataNodes and are the actual storage nodes, which store the data on their locally attached storage. When a client issues a write to HDFS, it will first contact the NameNode to obtain the locations for each file block. The client will then chunk the file and stream the individual blocks to the specified locations (see Figure 2.7).
To achieve reliability, HDFS replicates single blocks (by default, HDFS uses 3-way replication). Data is *chain replicated*, i.e. when a block is streamed to HDFS, HDFS establishes a pipeline between the 3 DataNodes that have been chosen to store the replicas and forwards the data along that pipeline to each DataNode. To increase reliability, HDFS tries to distribute replicas in a cluster across different racks so that if a rack goes down, the data is still available. When a client issues a read for a file, HDFS tries to read from the replica closest to that client first. If the read fails, it will revert to a replica outside of the current client’s rack. Note that the client is often executed from a node within the storage cluster itself and hence the closest node is often the local node from which the client is executed.

**Interaction with data-parallel frameworks**

During the processing of a job, data-parallel frameworks have two main interactions with the distributed storage layer: first, when they read the input data and, second, when they persist the results. During a MapReduce job, the mappers read the input data from HDFS while the reducers write the output to it. Multi-stage frameworks such as Spark or Flink operate the same way as data during intermediate stages is not written to the distributed storage\(^1\). In the default case, one input task from the first stage reads a single block from the distributed file system.

Blocks provide a fine storage granularity, which can be adjusted by controlling the block size. Additionally, the centralised design allows HDFS to control the block locations from the NameNode. This has the major advantage of being able to evenly spread blocks across the storage nodes, which in turn results in high data locality. As blocks are evenly distributed in the cluster, tasks that read input data can be scheduled on the node that stores the input data with high probability. This avoids remote reads. Additionally, writer tasks can write results to their local nodes and thereby avoid remote writes. Furthermore, I/O load is also spread evenly across the cluster which provides high aggregate disk throughput.

When a job has finished, the tasks from the last stage write the results back to the distributed storage. Tasks follow a two-phase commit protocol to commit their outputs. First, a task writes its output to a temporary location. The write uses the same chain-replication as described above to create the specified number of block replicas. The task then informs an *application master*, i.e. a special coordinator node that is responsible for managing the job, that it has finished. If allowed to commit by the master, the node moves the temporary file to its final location and exits. This protocol is necessary if optimisations such as speculative execution (see §2.2.1), in which multiple tasks process the same input data, should be supported. It prevents speculative tasks from overwriting each others results. After all tasks

---

\(^1\) These frameworks provide the option to persist intermediate results to distributed storage for fault-tolerance purposes, which would result in additional storage interaction. However, we do not consider this case here.
have finished, the application master moves the output files from an internal staging directory to the output directory specified by the user to signal the user that the results are now available. This is done sequentially for each output file (one per task). Note that, as HDFS uses the NameNode as the central metadata server, the application master must only contact the NameNode to update the metadata.

### 2.3.2 Object-based storage

While HDFS is the main deployment option with data-parallel frameworks, other storage backends have been used with these frameworks. For example, existing work has evaluated MapReduce on the HPC file systems PVFS [TSP+11] and GPFS [AGP+09]. Here, we focus on another class of storage backend that has gained popularity in recent years: **object stores**. Object stores such as Amazon S3 [Ama16], OpenStack Swift [Swi16], and Cleversafe [Cle16] have established themselves as a prominent solution for large-scale storage. Most cloud vendors’ storage services are based on object stores because they allow to store arbitrary (i.e. structured and unstructured) data with high availability and almost unlimited scalability [Bar16]. As a growing amount of data is now stored in object stores, this stored data can provide valuable insights and, as a result, new efforts have been made to run data-parallel frameworks directly on top of objects stores. For example, Google has recently released a connector for their cloud platform to give its MapReduce service direct access to the data stored in its object-based Google Cloud Storage [GCP16]. Another example is Netflix, which chose S3 as its data source for their Hadoop-based data warehouse due to its high scalability and availability [Net16].

**Object store basics**

Object stores are fundamentally different from a distributed file system such as HDFS. In an object store, data is stored and exposed to clients at the granularity of BLOBs as compared to blocks in a traditional filesystem [FMN+05]. A BLOB can be any kind of binary data such as images, videos, or documents and is usually immutable [MLR+14]. Clients interact with the store via a simple key-value-based RESTful API, with facilities to store (PUT), retrieve (GET) and delete (DELETE) objects based on their keys. The object store exposes a flat namespace in which the key is an arbitrary identifier for the object, in most cases, its name. Stores provide an additional layer of hierarchy to group objects, called containers in OpenStack Swift or buckets in S3. As in distributed filesystems, object stores use replication for reliability.

Different types of object stores exist such as NoSQL key/value stores [DHJ+07], caching servers [Mem16] or large storage infrastructures that are used in enterprises or for cloud storage [Ama16] [CDM+12] [BKL+10]. Our focus is on the third class because it provides highly-scalable storage for unstructured data, which is of particular interest for large-scale
2.3 Scalable storage technologies

Data analysis. We will explain the main architecture of an object store using the example of OpenStack Swift [Swi16]. However, the general concepts also apply to other enterprise-class object stores.

A major advantage of object stores over distributed filesystems is their scalability. Swift scales due to two key properties: first, it uses consistent hashing [KLL+97] to locate objects without a centralised metadata service. In consistent hashing, storage nodes are randomly assigned positions in a logical ring structure. The ring is discretised by the output domain of a hash function and closed at the smallest and largest hash values. The location of an object is then determined by hashing its name and assigning it to the node whose position is closest to the object’s hash on the ring (see Figure 2.8); second, all object metadata, such as its creation time or checksum, is stored with an object instead of on a separate metadata server. These two properties allow object stores to keep all metadata decentralised and hence no single scalability bottleneck exists in the system.

Figure 2.8 shows the request processing path in Swift. When a client submits a request to retrieve object A, it contacts a proxy server. While the proxy server is a single entry point to the storage cluster, it is stateless and hence can be scaled arbitrarily. After receiving the request, the proxy computes the object hash and looks up its position in the ring in order to determine the responsible object server. It then forwards the request to the identified object server for processing and returns its response to the client. Besides the object servers, the cluster has container servers that store the listings of objects grouped within a container. Containers are also located via the hash function.

Interaction with data-parallel frameworks

To integrate data-parallel frameworks with different storage backends, different *connectors* are developed. A connector is an additional layer that intercepts filesystem calls from the
Chapter 2. Background

Figure 2.9: Comparison of four different MapReduce workloads running on top of OpenStack Swift and HDFS

data-parallel framework and translates them into correct calls to the new storage backend. For example, OpenStack released such a connector to allow any system that supports HDFS to communicate with the Swift object store [Had16b]. However, due to the different architecture of object stores, which achieve their very advantages over HDFS and similar distributed filesystems, the integration with data-parallel frameworks is not seamless. Figure 2.9 demonstrates this. It shows the completion times of four different MapReduce jobs when running on HDFS and two different OpenStack Swift deployments\(^2\). We can see that the performance of the jobs degrades heavily when executed on Swift. For the Terasort workload, one of the Swift deployments does not even finish as the job crashes before it can complete.

Chapter 5 analyses and explains the observed performance problems in detail but we provide a short explanation here. The longer job completion time with an object store is due to two main problems: first, as explained above, object stores work on the principle of consistent hashing that allows locating stored objects across the cluster in a decentralised way. While this allows for decentralised object lookups, it also creates a fixed mapping between object names and storage devices, which does not work well for some operations. For example, renaming an object degenerates to re-uploading the same file with a different name. This generates additional network traffic and adds job latency. As the data-parallel frameworks require several renames of the job results to ensure atomicity, performance decreases significantly; second, object stores are not block-based and do not typically chunk objects across all storage servers, but rather upload them in their entirety. This coarser storage granularity generates imbalances in the I/O load and reduces data locality which again, leads to additional network traffic. Eliminating this extra traffic is essential in ensuring a smooth integration of data-parallel frameworks and object stores.

\(^2\)We will cover the details of this experiment in Chapter 5.

64
2.4 Summary

The community around Spark has an ongoing effort to solve the problem of slow renames, when Spark is executed on an object store. The `DirectOutputCommitter` [Rot16] has been proposed to directly write results to an object store without performing a two-phase commit. The recently open-sourced Stocator project [Ver16] is an optimised connector between data-parallel frameworks and object stores. It supports writing results directly to an object store and prevents potential data loss due to competing tasks by appending a unique suffix to the output file name of each single output task.

Discussion

Analytics on object stores are an attractive combination. Removing the need for an additional data silo while still enabling analytics on the data saves time and cost in terms of performance, management, and hardware. However, the integration is not seamless and leads to inferior performance compared to distributed filesystems. While some existing approaches try to solve these problems exist, they still are not perfect. The `DirectOutputCommitter`, for example, can still lead to data loss, when speculative execution is used [SPA16]. While Stocator solves this, it comes at the expense of a changed output file name, other than the user has specified. This makes it harder to retrieve the results of a job or use them in a longer workflow.

We believe that, to solve the above presented problems, it is necessary for object stores to incorporate a notion of locality, i.e. to become aware of their distributed nature. As in HDFS, locality avoids additional network traffic and I/O due to copying and hence can remove the additional latency that object stores add to jobs.

2.4 Summary

In this chapter, we reviewed the basic concepts and terminology that is required for this thesis. We started with an overview of data centre networks and introduced their structure and topology. We discussed a variety of different topologies including multi-rooted trees, random and server-centric designs, and enhancements using optical and wireless networking. While different designs have different advantages, we found that all of them have certain drawbacks, which have not yet been solved from the infrastructural side and hence can benefit from support from the application. Further, we covered the programmability aspect of data centre networks with middlebox platforms and software-defined networking. While these approaches ease the development of network functions such as firewalls or load balancing, they do not support the implementation of high-level, application-specific functions. We finished by looking at traffic patterns in data centres and found that traffic is highly variable and often skewed towards a subset of servers which leads to bandwidth and throughput variations for applications.
Next, we introduced data-parallel frameworks. We discussed their main concepts, including data-parallelism, dataflow graphs, and fault tolerance, and also explained resource management approaches. We then focused on two specific classes of queries, partition/aggregate queries and join queries, which both have a high network utilisation. Existing optimisations to reduce network traffic for partition/aggregate queries are all edge-based, which means that the degree to which traffic can be reduced is limited and hence queries can still suffer due to oversubscription and scarce edge bandwidth. On the other hand, join queries have been optimised to mitigate data skew and balance the network utilisation statically but are still susceptible to dynamic network skew.

We finished this chapter by discussing scalable storage systems from which data-parallel frameworks read their input data and to which they write the results of a query. We introduced HDFS as the prime example for a distributed, block-based filesystem and explained its concept of locality. We then looked at object stores as an attractive and highly scalable alternative to HDFS. Due to their different architecture, object stores are not able to match the performance of HDFS when used as a storage backend for analytics jobs. This is due to the fact that the optimisations that have been built into HDFS and are designed to reduce I/O and network communication cannot be ported to object stores. While some existing work has attempted to solve the performance problems, it either risks data loss or affects usability.

In the following, we will propose solutions to the three major problems discussed. First, we will reduce the traffic in the network during partition/aggregate queries to mitigate the effects of oversubscription and N-to-1 traffic bursts. We then dynamically change the partitioning during a distributed join to adapt to the occurrences of network skew. Finally, we eliminate the additional network traffic that occurs when running a data-parallel job on an object store.
This chapter introduces NetAgg, a system for transparent in-network aggregation for partition/aggregate queries. NetAgg exploits the tree topology of modern data centre networks and the abundance of middleboxes to expose an in-network aggregation service that allows to partially aggregate query traffic on-path as it flows through the network. This reduces traffic as early as possible and thereby reduces bandwidth consumption and improves query performance.

### 3.1 Overview

NetAgg is based on two common properties of DCNs:

1. DCN topologies are multi-rooted, two- or three-layered trees;

2. DCNs contain many middleboxes, which implement various network enhancements.

We leverage these two features of the network to provide an in-network aggregation service. The service allows users to implement application-specific aggregation functions for partition/aggregate queries and deploy them on existing middleboxes, called Agg boxes in the network. Query traffic is then routed through these Agg boxes, and the corresponding aggregation function is applied to the partial results of a query at each hop in the network, which continuously reduces the data as it flows towards the master node. This saves bandwidth
Chapter 3. Traffic Reduction for Aggregatable Traffic

Figure 3.1: On-path aggregation leveraging the DCN topology and middleboxes. The $x/y$ notation refers to 'layer/node'

and relieves pressure on links that would otherwise exist due to oversubscription or the scarce edge bandwidth of the master node. In contrast to existing, edge-based aggregation approaches [DG04] [LTWY11] [MGL+10] [YGI09], NetAgg’s in-network aggregation reduces data even after it entered the core network.

We assume a set-up similar to the one depicted in Figure 3.1, in which Agg boxes are directly attached to network switches and perform on-path aggregation. We create an in-network aggregation tree which forms a spanning tree across all edge hosts and switches involved in the aggregation query. The root of the tree is the final master node, its leaves are the workers, and the internal nodes are the Agg boxes. Each Agg box aggregates the data coming from its children and sends it downstream to the next Agg box or the master at the last hop. As modern data centre networks offer multiple paths between pairs of end hosts, a single query can construct multiple aggregation trees and equally split the data among them using a scheme such as ECMP [Hop00]. This allows NetAgg to utilise all available paths. NetAgg is compatible with existing middlebox deployments and hence can be deployed with low effort. Additionally, an incremental deployment, in which the Agg boxes are attached to only a subset of the switches, is possible.

3.1.1 Design challenges

NetAgg has three main design challenges:

**Application transparency.** A deployment of NetAgg should be transparent to the application, i.e. neither the data-parallel framework nor the partitionaggregate query should have to be altered to support in-network aggregation. This ensures that NetAgg can be deployed with different frameworks and queries without additional effort. To achieve this, NetAgg uses *shim layers*, i.e. layers between the application and the network socket at edge servers that transparently intercept application traffic and redirect it to Agg boxes.
3.1 Overview

**Multi-application support.** Agg boxes should be able to support multiple applications. When the data centre and the aggregation service are shared between different applications, each one should receive a fair share (according to some policy) to provide predictable latencies and prevent interference. In NetAgg, Agg boxes manage the scheduling of aggregation tasks. This means that they control how their resources are allocated to applications to, for example, give priority to latency-sensitive applications with critical deadlines over throughput-oriented batch applications.

**Fast processing.** NetAgg has to be able to aggregate partial results at the highest possible rate to not become a bottleneck. To achieve this, Agg boxes decompose aggregation computation into cooperatively scheduled aggregation tasks, which are executed in parallel across many CPU cores.

3.1.2 Design space

NetAgg differs from the existing middlebox proposals for network function virtualisation as presented in §2.1.2 in two important ways. First, it allows users to easily port the aggregation computation without any changes from the target framework to the NetAgg system. While existing solutions expose low-level abstractions [KMC+00] [MAR+14] [ABK+12] [BAM13] that require expert knowledge to correctly implement the desired computation, NetAgg can use an existing aggregation function, provided in a high-level language such as Java, and execute it without any changes while simultaneously providing high processing rates. Second, while there are flow-based abstractions that allow to implement higher-level, application-specific functionality [ABK+12] [BAM13] [ZWRH14], they only support processing on a single flow. NetAgg allows to combine multiple network flows and uses this to implement application-specific aggregation for traffic reduction.

In-network aggregation has also been explored as part of research in wireless sensor networks [MFHH02] [MNN08] [IGE+03] and in overlay networks for Internet-scale distributed aggregation [CH08] [JKM+07] [YD04]. This work demonstrated that early aggregation can significantly improve query performance. However, it targets different deployment environments and hence does not deal with the challenges that are required for an aggregation service in data centres. In particular, high performance and multi tenancy support are specific to data centre deployments.

The closest existing work to NetAgg is Camdoop [CDRO12]. Camdoop is based on the server-centric CamCube topology [ALCR+10] in which servers act as routers and forward data towards their destination. Camdoop exploits this to perform in-network aggregation on the forwarding path. While this is similar to NetAgg, Camdoop relies on a specific and server-centric topology, which is not common in existing data centres, making it harder to deploy as a solution.
3.1.3 Contributions

In this chapter, we make the following contributions:

1. We present the design of NetAgg, a system to perform fast in-network aggregation in data centre networks. NetAgg leverages the common tree topology of DCNs and existing middleboxes to expose a compatible and cost-efficient aggregation service to users to reduce traffic as it flows through the network (§3.2).

2. We introduce Agg boxes as generic middlebox platforms to deploy application-specific aggregation functions in the network. Agg boxes expose the aggregation interface of existing data-parallel frameworks, which allows users to deploy their aggregation function on NetAgg without any changes. Agg boxes parallelise aggregation computation to achieve high processing rates (§3.3.1).

3. To make the NetAgg service transparent to users and hence ease its adoption, we propose shim layers. Shim layers intercept aggregatable traffic at end hosts and reroute it to Agg boxes. Shim layers also take care of correctly passing the aggregated results to the master node to make sure the results are returned correctly to the user (§3.3.2).

We evaluate NetAgg both at scale, using simulations, and on a 34-server testbed. Currently, NetAgg supports two applications: Apache Solr [Sol16], a distributed search engine, and the Apache Hadoop MapReduce framework [Had16a]. In simulation, we show that NetAgg reduces the flow completion time of data centre flows by up to 88% compared to existing, edge-based aggregation solutions. Results from our testbed evaluation reveal that NetAgg is able to improve the performance of Solr search queries by up to $9.3 \times$ and Hadoop jobs by up to $5.2 \times$.

3.2 NetAgg system design

We describe the design of the NetAgg middlebox platform. We first give an overview of its basic deployment and present a cost/performance analysis to study its feasibility. We then provide details on the implementation of Agg boxes and shim layers and report on two application case studies.

3.2.1 Deployment

Figure 3.2 shows a deployment of NetAgg in a data centre. It consists of two main components, Agg boxes and shim layers. Agg boxes are connected to all or a subset of the switches in the data centre via high-bandwidth network links. The high-bandwidth link is required to
prevent oversubscription on the path to the Agg box. We will consider the impact of different bandwidths on performance and compare their upgrading cost to full bisection bandwidth topologies in §3.2.2. Agg boxes perform the on-path aggregation of application data according to aggregation functions implemented by the user. Edge servers are fitted with shim layers that intercept result data flows. Shim layers forward the intercepted results to Agg boxes and also receive the aggregated results at the master node. The master expects multiple incoming flows carrying result data. However, it now only receives a single flow with already aggregated data. The shim layer at the master node makes sure to hide this fact to avoid modifications to the master.

The Agg boxes cooperate to establish an in-network aggregation tree that aggregates partial results from worker nodes before sending the final result to the master node. Partial aggregation relies on the associativity and commutativity of aggregation functions to ensure the correctness of the final result. Figure 3.2 shows the workflow between Agg boxes and shim layers during a partition/aggregate query.

1. A client submits a query to a master node, which partitions it into multiple sub-queries (see Figure 3.2a).

2. The master node sends the sub-queries, which pass through the shim layer of the master node without modification, to a set of worker nodes.

3. The partial results generated by the workers are intercepted by the shim layers of the worker nodes. The shim layers redirect the data to the first Agg box along the network path from the worker to the master node (Figure 3.2b). For example, Agg box 2,
connected to an aggregation switch, aggregates partial results from workers in other parts of the data centre.

4. Each Agg box aggregates the partial results according to the aggregation function (Figure 3.2c). It sends its partially aggregated results to the next Agg box that is along the network path to the master node.

5. The Agg box nearest to the master node (i.e. Agg box 1 in the example) sends the fully aggregated results to the master node. The shim layer of the master node passes the results to the application.

Using this on-path aggregation scheme, data is reduced at every hop in the network. Depending on the reduction or output ratio of the query, i.e. the ratio between the amount of data flowing in and out of an Agg box, the effects of oversubscription and of the scarce edge-bandwidth at the master node can be significantly mitigated. If the output ratio is equal to the oversubscription ratio, queries can run as fast as in a full bisection bandwidth network.

3.2.2 Feasibility study

One question in the design of NetAgg is what bandwidth is required for the link connecting an Agg box to a switch to get the most benefit and what is the cost of upgrading these links in the data centre. Additionally, a potential issue in a software-only approach based on commodity middleboxes is the lower processing rate that the aggregation computation can achieve on these middleboxes compared to custom hardware solutions. To understand these trade-offs, we conduct a number of simulations to understand the minimum processing rate $R$ of an Agg box required to achieve noticeable benefits. We thereby also determine the bandwidth of the connecting link and the performance/cost trade-off involved in upgrading the required links.

We describe the simulation set-up and the full set of results in §3.4.1. Here, we briefly sketch the set-up and highlight the main results, which led to the NetAgg design described above.

We consider a three-tier, multi-rooted network topology, modelled after the Clos topologies as described in §2.1.1. Each server is connected through a 1 Gbps link to a ToR switch. We assume an oversubscription ratio of 1:4, which is consistent with values reported in the literature [GHJ+09]. We also show the results for the case when full bisection bandwidth is available. We consider a data centre with 1024 servers and run a synthetic workload modelled after Facebook network traces in which 46% of traffic is aggregatable while the remaining 54% cannot be aggregated. [CKS13]
3.2 NetAgg system design

**Performance requirements**

As our baseline, we consider rack-level aggregation, i.e. one server per rack collects the partial results of all workers in that rack and then sends them to the master (see §2.2.2). We assume ECMP [Hop00] as the routing protocol and use flow completion time (FCT), i.e. the time elapsed between the start and the end of a flow, as our metric [DM06]. In the simulations, we vary the maximum processing rate that can be sustained by the Agg boxes.

Figure 3.3 shows that relatively modest processing rates around 4 – 6 Gbps are sufficient to achieve significant benefits. Even a rate of 1 Gbps per Agg box already reduces the total completion time by more than 74% for the 1:4 oversubscribed scenario and 63% for the non-oversubscribed one (88% and 90% for the rate of 8 Gbps, respectively). This shows that, although the bandwidth per Agg box is identical to the one used by the rack-level aggregator, performing aggregation constantly as data flows through the network significantly reduces the pressure placed on the network core. Note that these results include all flows, i.e. aggregatable and non-aggregatable ones. The reason is that NetAgg reduces the traffic for aggregatable flows, which leaves more available bandwidth to other flows and hence improves their performance simultaneously.

The results show that NetAgg can benefit even existing 1 Gbps infrastructures. However, upgrading to 10 Gbps yields close to a speed-up of 2× for flows and we will use this as the link capacity to Agg boxes. In §3.4.2, we show that an Agg box can perform aggregation fast enough to saturate a 10 Gbps link.

**Cost analysis**

Next we perform a simple cost analysis to understand the cost benefits of NetAgg over simply increasing the network bandwidth. We consider three alternative options to NetAgg: a full-bisection network topology with 1 Gbps edge links (Full Bis. (1 Gbps)); a 1:4 oversubscribed network with 10 Gbps edge links (Oversub (10 Gbps)); and a full-bisection network topology with 10 Gbps edge links (Full Bis. (10 Gbps)).
Chapter 3. Traffic Reduction for Aggregatable Traffic

We compare the performance and cost when using rack-level aggregation in the three configurations above to deploying NETAGG in the base set-up (1:4 oversubscription and 1 Gbps network). We use the same workload as in the previous experiment, and assume a processing rate \( R \) of 9.2 Gbps. This is similar to the rate of our prototype. For NETAGG, we consider two deployment options: NETAGG, in which an Agg box is connected to each switch in the topology, and Partial NETAGG, in which Agg boxes are only connected to switches in the middle tier. We adopt the prices for network equipment from a past study [PRI+10]. We assume hardware specifications for servers and Agg boxes as used in our testbed (see §3.4.2).

Figure 3.4 shows the performance improvement and upgrade costs with respect to the base set-up. As expected, upgrading to a 10 Gbps full-bisection network provides the largest benefit (92% reduction of FCT), but also incurs the highest cost. Full Bis. (1 Gbps) has a much lower deployment cost but only reduces FCT by 24%. In contrast, deploying NETAGG achieves almost the same performance improvement (88% FCT reduction) as the 10 Gbps full bisection network and the oversubscribed 10 Gbps network (87% FCT reduction) at significantly less cost. A NETAGG deployment cost only 18% of the Full Bis. (10 Gbps) and only 22% of Oversub (10 Gbps). Partial NETAGG is also a practical deployment option: it only incurs 4% of the cost of the oversubscribed 10 Gbps network but can reduce FCT by 75%.

While we compare the performance of on-path Agg boxes against the performance of an upgraded network infrastructure, our solution remains complementary. Even with more available network bandwidth, on-path aggregation can reduce bandwidth consumption and provide more bandwidth to other flows which are not aggregatable. However, as edge bandwidth increases in the network infrastructure (e.g. using 10 Gbps links), the performance of Agg boxes also has to increase to keep the same relative benefit.

3.2.3 Requirements in data centre environments

To be deployable in large-scale data centres, NETAGG must deal with a variety of different scenarios in terms of applications and failures as discussed next.
Multiple applications. Multiple applications may use the NetAGG aggregation service concurrently and construct different in-network aggregation trees. To forward data belonging to the same query along the correct aggregation tree, an Agg box uses knowledge of its child and parent nodes. An Agg box determines the next Agg box on the network path for a query by hashing a unique application/query identifier. It then forwards the partial results to this hop towards the master. This ensures that partial data for a given application and query always traverses the same in-network aggregation tree.

Multiple aggregation trees per application. With a single aggregation tree per application, it is not possible to exploit DCN topologies that support multiple routing paths. This can waste the higher core network bandwidth that multi-path topologies offer. NetAGG therefore supports multiple aggregation trees for a single query that are used concurrently by partitioning the aggregation load among the trees. Each aggregation tree uses a disjoint set of Agg boxes (except for the Agg box that is in the same rack as the master) and thereby can utilise different paths.

With multiple aggregation trees per application, the shim layers at the worker nodes partition partial results across the trees. Typically this can be achieved by hashing request identifiers for short queries or keys in long-running batch jobs. The master node is the root node for all aggregation trees and performs the final aggregation step.

Multiple Agg boxes per switch. To increase the throughput of an Agg box connected to a switch, it is possible to scale out processing by load-balancing aggregation computation across multiple Agg boxes connected to the switch. In this case, aggregation trees are assigned to Agg boxes in a way that balances the load between them. This does not affect the correct forwarding for a single query as an additional Agg box at a switch only means an additional parent node for upstream Agg boxes/workers.

Agg boxes provide additional hardware for the computation of the aggregation function. As a result, the load on the master node is distributed across the Agg boxes which speeds up the aggregation computation, even in case Agg boxes are CPU bound. The only exception is the case, in which the combined processing resources across all Agg boxes are less than what is available at the master node. As a result, rerouting traffic through Agg boxes can slow down the transfer and should be avoided. However, as in-network aggregation trees in large DCNs consist of many nodes and hence, many Agg boxes, this case is rare.

Handling failures. The NetAGG design uses a lightweight failure detection service, running at both the Agg boxes and the master shim layer, that monitors the status of upstream Agg boxes in the distributed aggregation tree. When a node $i$ (either an Agg box or the master node) detects that an upstream Agg box $j$ has failed, it contacts the child nodes (either Agg boxes or worker nodes) of $j$ and instructs them to redirect future partial results to itself, skipping one intermediate step. To avoid duplicate results, when $i$ contacts the
upstream nodes of $j$, it also sends the last result that has been correctly processed so that already-processed results are not resent.

**Handling stragglers.** NetAgg is designed to be compatible with existing mechanisms used by applications to handle straggling worker nodes (see also §2.2.1). In case of a straggler, Agg boxes just aggregate the available results from the non-straggling nodes, while later arriving results are sent directly to the master.

Since an aggregation tree involves multiple Agg boxes, each Agg box itself can potentially become a straggler, delaying the computation of the final result. To handle this scenario, NetAgg uses a similar mechanism to the one for failures: if a node detects that an upstream Agg box $i$ is too slow (based on an application-specific threshold), it contacts the upstream nodes of $i$ to redirect future results to itself. The difference with the failure protocol is that the redirection is only applied to results of the same request because the cause of straggling may be specific to it. However, if an Agg boxes is repeatedly marked as slow by different queries, it can also be considered failed.

### 3.3 NetAgg components

We now introduce Agg boxes and shim layers in detail. We implement a prototype version of NetAgg in Java. As many data-parallel frameworks, e.g. Hadoop, Spark, or Flink, are implemented in Java, this permits the Agg boxes to execute unmodified aggregation functions for these frameworks.

#### 3.3.1 Agg boxes

An important goal for Agg boxes is to use available hardware resources efficiently in order to process data with high throughput. Their implementation is therefore data-parallel. They decompose aggregation functions and parallelise their execution across CPU cores using **cooperative scheduling**. In cooperative scheduling, tasks are never preempted but always run to completion.

To further improve performance, Agg boxes only run the application-specific aggregation function on data traffic. Any other, control-related traffic is not forwarded to Agg boxes, and they do not participate in control interactions of the framework. Shim layers are able to distinguish between control and data traffic. For multiple applications to share a NetAgg deployment, it is necessary to schedule access to the limited CPU and bandwidth resources of Agg boxes. Agg boxes allow to specify priorities for different applications, e.g. latency-sensitive applications normally would have a high priority while batch processing has lower
priority. Agg boxes schedule the execution of aggregation computation belonging to different applications using adaptive weighted fair queueing.

The architecture of an Agg box is depicted in Figure 3.5. An Agg box executes *aggregation tasks* that wrap the applications’ aggregation computation. Tasks are organised into a *local aggregation tree* that parallelises the aggregation function and are scheduled cooperatively across CPU cores by a *task scheduler*. A *network layer* serialises and deserialises data. We discuss each of these components in more detail below.

**Aggregation tasks.** An Agg box represents computation as aggregation tasks, which are fine-grained compute units that can be scheduled in parallel on CPU cores. Different applications require different interfaces for aggregation functions and hence NetAgg provides different aggregation wrappers that wrap the application-specific interface to support easy portability of existing aggregation functions. For example, a Hadoop aggregation wrapper exposes the standard interface of combiner functions: `Combiner.reduce(Key k, List<Value> v)`. This allows users to deploy their existing combiner functions on Agg boxes without any modifications.

**Local aggregation trees.** Similar to how aggregation computation is decomposed across multiple Agg boxes, Agg boxes decompose aggregation further across multiple CPU cores using a local aggregation tree of tasks. Local aggregation trees have a large fan-in and are executed in a pipelined fashion by streaming data across the aggregation tasks. This allows...
for CPU- and memory-efficient, scalable aggregation, as aggregation executes in parallel and no large buffer space for intermediate data is required.

As shown in Figure 3.5, the intermediate tree nodes of a local aggregation tree are aggregation tasks, acting as producers and consumer of data. Leaf nodes receive partial results from worker nodes or downstream Agg boxes and forward them to their parent aggregation tasks for aggregation. Tasks are scheduled by the task scheduler (see below) when new input data is available at a producer and there is sufficient buffer space at the corresponding consumer task. Partially aggregated results propagate up the tree until the root node obtains the final result. If the computation in the local aggregation tree is not fast enough to sustain the rate at which data arrives from the network, TCP back-pressure ensures that the workers reduce the rate at which they send partial results.

**Task scheduler.** To reduce the overhead of thread synchronisation, aggregation tasks are scheduled by a task scheduler using cooperative scheduling: when aggregation is possible, a task is submitted to a task queue and waits to be scheduled. Threads from a fixed-sized thread pool, usually equal in size to the number of available CPU cores, are used to run the tasks to completion. Currently, we assume that the user-defined aggregation functions are well-behaved and terminate. Mechanisms for isolating faulty or malicious tasks that try to hog resources are left for future work.

If there are multiple applications using an Agg box, resources must be shared to achieve acceptable processing times for the individual aggregation computation of each application. Agg boxes maintain a separate task queue per application and adopt a weighted fair queueing policy over these queues. This enforces weighted fair shares among applications, similar to other cluster schedulers [ZBSS+10]. In weighted fair queueing, when a thread becomes available, the scheduler offers that thread to a task of application $i$ with probability $w_i / \sum_{i=1}^{n} w_i$, where $w_i$ is application $i$'s weight and $n$ is the number of applications on the Agg box. The probability for a task of a specific application to be scheduled is hence proportional to the allocated weight of this application.

Initially, the scheduler assumes that tasks from different applications have the same execution time. However, in most cases, the computation requirements of tasks varies depending on the application. To handle this case, the scheduler periodically adapts the weights of an application $i$ according to its average task execution time $\bar{t}_i$ measured at runtime. Based on these measurements, the scheduler then normalises the execution times by incorporating the differences into $w_i$. For example, if two applications run on an Agg box with equal shares but the tasks of one application take twice as long compared to the other, the scheduler halves the weight of that application to achieve the targeted fair share. More formally, given a target resource share $s_i$ for application $i$, $w_i$ is set to $s_i \bar{t}_i / \sum_{i=1}^{n} s_i \bar{t}_i$. Our implementation uses a moving average to represent the measured task execution time.
3.3 **NetAgg components**

**Network layer.** To optimise the communication within the NetAgg system, Agg boxes transfer data with an efficient binary protocol using KryoNet [Kry16]. KryoNet is an NIO-based network communication library for Java, which uses Kryo for fast serialisation/deserialisation. This avoids the overhead of potentially wasteful application-specific network protocols, such as HTTP or XML. The shim layers also maintain persistent TCP connections to their parent Agg boxes and deserialisation is parallelised by the leaf nodes in the local aggregation trees. Since the actual application data must be deserialised before the aggregation wrapper can call the aggregation function, the network layer of the Agg box includes a serialiser/deserialiser taken from the application. For example, to support the aggregation of Hadoop key/value pairs, the Agg box uses Hadoop’s SequenceFile reader and writer classes.

3.3.2 **Shim layers**

To make it easy for applications to benefit from on-path aggregation, NetAgg’s shim layers transparently intercept data flows and redirect them to Agg boxes. The shim layers control the redirection of application data and manage the collection of partial results.

**Network interception.** A shim layer intercepts network traffic at the level of network sockets by wrapping the actual Java network socket class in a NetAgg socket. By using Java’s ability to change the default socket implementation via the SocketImplFactory, applications transparently generate an instance of the custom NetAgg socket class every time a new socket is created. The NetAgg socket overwrites the read and write methods of the default socket and adds logic which identifies aggregatable application data and forwards it to an Agg box. All other data is written through to the underlying socket.

**Partial result collection.** A challenge is that Agg boxes must know when all partial results were received before executing the aggregation function. However, for many applications including Apache Solr, it is not known ahead of time how many partial results will be returned by the worker nodes. As the master node always knows how many responses it expects, we currently solve this problem by recording information about the number of responses, typically found in request headers, at the master shim layer. This information is then sent to the Agg boxes. Another option would be to wait until a certain timeout and then aggregate all data that is available. This is similar to the straggler-handling mechanism of NetAgg.

**Empty partial results.** The master node expects to receive partial results from all worker nodes, but with NetAgg this is no longer the case. All responses will be returned as a single, fully aggregated response from the final Agg box on the path to the master node. This means that the shim layer at the master node must emulate empty results from all the workers except one as the master is still waiting for responses from these workers. The empty results do not affect the aggregation logic because we assume that aggregation functions are commutative and associative.
3.3.3 Application deployments

To demonstrate the generality of NETAGG, we describe its deployment with two partition/aggregation applications: Apache Solr, a distributed full-text search engine [Sol16], and the data-parallel framework Apache Hadoop [Had16a] MapReduce. NETAGG can support both applications after providing application-specific serialiser/deserialiser, aggregation wrapper, and shim layer code. The required implementation effort is summarised in Table 3.1.

Apache Solr performs full-text search across an index stored at multiple backend server nodes, acting as the workers. Clients send small search queries to a frontend node (the master), which dispatches sub-queries to the backend nodes. The backends return partial search results to the frontend node, which combines them into a ranked result using an aggregation function. By default, the aggregation function computes the top-k results but users can specify a custom aggregation by implementing the QueryComponent interface. To support Solr, NETAGG requires around 600 lines of code. The aggregation wrapper wraps the QueryComponent interface. For Solr, an Agg box buffers all partial results before invoking the local aggregation tree, which is feasible because results are only on the order of hundreds of kilobytes.

Apache Hadoop MapReduce uses worker nodes to execute the mappers, which produce the partial results. These results are then sent to and aggregated by reducers. There are fewer than 800 lines of code needed to support Hadoop. The aggregation wrapper implements the Hadoop interface for combiner functions. The deserialiser for Hadoop is slightly more complex than for Solr. Hadoop jobs usually involve large amounts of data, which are streamed in chunks through Agg boxes. Hence, the serialisation/deserialisation code must account for incomplete pairs at the end of each chunk.

3.4 Evaluation

We begin our evaluation by presenting the simulation results, extending the study from §3.2.2. We then show the results from our NETAGG 34-server testbed with different applications.
3.4 Evaluation

3.4.1 Simulation-driven evaluation

We simulate a three-tier, multi-rooted network topology based on a Clos topology as introduced in §2.1.1. The network consists of 320 20-port switches connecting 1024 servers via 1 Gbps links, unless stated otherwise. As a default, we assume a 1:4 oversubscription ratio at the ToR tier, which we vary from 1:1 to 1:10 in one of the experiments. We use the OMNeT++ packet-level simulator [OMN17], which implements TCP max-min flow fairness and uses standard ECMP for routing. Unless noted otherwise, we assume that each Agg box is connected to a switch through a 10 Gbps link and can process network flows at 9.2 Gbps. As we show in §3.4.2, this is representative of the performance of our current NetAgg prototype implementation.

We model the traffic according to published network traces from a large cluster running data analysis jobs [GHJ+09]:

- Flow sizes are Pareto-distributed with mean 100 KB and shape 1.05 [GHJ+09].
- The number of flows is determined such that the load at edge links is 15% [AKE+12].
- 46% of flows are aggregatable, while the rest is non-aggregatable background traffic [CKS13].
- The number of workers generating flows follows a power law distribution with 61% of queries having fewer than 10 workers [AGSS13].
- We use an output ratio $\alpha$ of 0.1 but also vary it in experiments.

Except when simulating straggler nodes, all flows start at the same time, which is a worst case for network contention. We also ran experiments using dynamic workloads with various arrival patterns, obtaining comparable results (between 1%–4% of the reported FCT values). Workers are deployed using a locality-aware allocation algorithm that greedily assigns workers to servers as close to each other as possible.

Metric and baselines. We use the flow completion time (FCT) as our evaluation metric with the goal of reducing it as much as possible. Since our focus is on network bottlenecks that increase FCTs for individual flows, we report the 99th percentile of FCT, unless stated otherwise. We compare the performance of NetAgg against the following aggregation strategies as described in §2.2.2: rack-level aggregation (rack), a binary aggregation tree (d=2; binary) and chain aggregation (d=1; chain). Unless reporting absolute FCTs, we normalise the performance of the other strategies against rack.

Distribution of flow completion times. We report the cumulative distribution function (CDF) of the absolute FCTs for all flows in Figure 3.6. As explained in §2.2.2, binary and chain increase link utilisation as more traffic is exchanged between workers. This in turn
Chapter 3. Traffic Reduction for Aggregatable Traffic

Figure 3.6: CDF of flow completion times for all traffic

Figure 3.7: CDF of flow completion times for non-aggregatable traffic

Figure 3.8: Relative FCT with varying output ratio $\alpha$

Figure 3.9: CDF of link traffic ($\alpha = 10\%$)

reduces the bandwidth available for other flows, which explains why these strategies reduce the FCT tail but worsen the median. In contrast, NetAgg improves the performance for all flows by reducing the traffic of the aggregatable flows in the network. As shown in Figure 3.7, NetAgg can also improve the FCT for non-aggregatable flows because due to the traffic reduction for aggregatable flows, more bandwidth resources are available for other flows.

Aggregation output ratio. The impact of the output ratio $\alpha$ is shown in Figure 3.8. We vary $\alpha$ from 0.05, which represents a high data reduction that emulates aggregation functions such as top-$k$, max or count, to 1 (i.e. no aggregation is possible). As expected, the benefit of NetAgg decreases for high values of $\alpha$. For small values of $\alpha$, NetAgg reduces the 99th percentile of FCT by up to 88% compared to rack and 80% and 52% compared to binary and chain, respectively.

For very high values of $\alpha$, NetAgg does not provide a benefit anymore. Data passes through the Agg boxes but cannot be reduced. While NetAgg is unable to improve this case, the results show that it also does not make it worse, meaning that rerouting and processing data at Agg boxes is efficient. Additionally, production traces show that most applications exhibit an output ratio lower than 10% [AGN+13, CGK11] and hence a small $\alpha$ is representative of typical workloads. We adopt $\alpha = 10\%$ in the rest of the experiments.
3.4 Evaluation

It is also worthwhile to analyse why chain is always outperformed by rack for $\alpha > 70\%$. The reason of this counter-intuitive behaviour is that chain utilises more link bandwidth compared to rack. As shown in Figure 3.9, with $\alpha = 10\%$, the median link traffic for chain is $4 \times$ higher than for rack ($2.5 \times$ for binary, respectively). When the network is highly loaded and $\alpha$ is large, this aspect dominates, thus lowering throughput.

**Fraction of aggregatable flows.** As described in §3.2.2, a typical data centre traffic workload contains a mix of flows, of which only a fraction can be aggregated. We conduct a sensitivity analysis in which we vary the fraction of aggregatable flows from 0.1 (only 10% of flows can be aggregated) to 1 (all flows can be aggregated). Figure 3.10 shows the 99th percentile of FCT relative to rack. With more aggregatable flows, the benefit of all three aggregation strategies increases. With more than 60% of aggregatable flows, however, the effectiveness of binary and chain starts to decrease again because their more wasteful usage of network bandwidth introduces network bottlenecks. In contrast, NetAgg consistently maintains the lowest FCTs.

**Oversubscription.** Figure 3.11 quantifies the performance impact of the bisection bandwidth on NetAgg and the baselines. We vary the oversubscription ratio from 1:1, which represents a full-bisection bandwidth network without oversubscription, to 1:10. As expected, NetAgg performs best for high oversubscription ratios. By aggregating flows along the network paths, it reduces congestion in the core due to oversubscription. However, NetAgg can also provide a benefit for the full-bisection bandwidth network. In that case, the inbound bandwidth of the rack and the master node becomes the bottleneck which NetAgg is able to alleviate due to its continuous traffic reduction in the network.

**Partial deployment.** An obvious question is whether a single aggregation node per rack, connected via a high bandwidth link, would be sufficient to achieve similar performance to NetAgg. This represents a sole edge-based approach. In a broader sense, we want to understand what should be the best deployment configuration, i.e. where should Agg boxes be deployed to achieve optimal performance, when there is only a limited number of Agg boxes available. In Figure 3.12, we compare the performance of a full NetAgg deployment against
three configurations: Agg boxes (i) only at the ToR switches; (ii) only at the aggregation switches; and (iii) only at the core switches. The results show that the biggest improvement is achieved with Agg boxes at the aggregation or core network tiers. Deploying them at the rack tier provides limited benefits, improving the FCT by only 38% against 83% and 75%, respectively.

This comparison, however, does not take into account that in a DCN topology the number of switches at each tier varies significantly. For example, our simulated topology has 128 ToR switches, 128 aggregation switches and 64 core switches. Therefore, we fix the number of Agg boxes to 64 to provide a fair comparison of the different deployment options. We measure the performance when deploying the 64 Agg boxes (i) at the core tier only; (ii) uniformly distributed at the aggregation tier; and (iii) uniformly distributed at the two tiers. Interestingly, the first configuration achieves the largest improvements (75%) while the other two achieve 29% and 43%, respectively. The reason is that the core tier has the highest potential for aggregating data because it intercepts more flows originating from the workers. This result is important because it shows the benefit of performing aggregating in the network and not just at the edge. It also demonstrates NetAgg’s ability to be deployed incrementally.

**Upgrading to a 10 Gbps network.** Next we explore the performance of NetAgg when deployed in a 10 Gbps network. We use the same topology and workload as before but with 10 Gbps links connecting servers to ToR switches. We vary the oversubscription from 1:1 to 1:10 and plot the results in Figure 3.13. Similar to the results from Figure 3.11, NetAgg provides significant benefit for large oversubscription values compared to rack. Although edge servers have the same bandwidth as Agg boxes, the oversubscription still limits the throughput in the network core which NetAgg is able to alleviate due to its in-network aggregation. This again demonstrates the benefits of aggregating flows at each network hop as opposed to at edge servers only.

For smaller oversubscription values, the processing rate of Agg boxes becomes the bottleneck, reducing benefit. To further increase performance, we explore a scale out configuration in which we attach 2 and 4 Agg boxes to each switch. We use a strategy similar to ECMP to
assign flows of the same application to the same set of Agg boxes. As the results show, 2 Agg boxes are sufficient to reduce the FCTs by up to 85% compared to rack. This demonstrates the value of scaling out, making NetAgg compatible with future network upgrades.

**Stragglers.** To quantify the impact of straggling workers on NetAgg, we run an experiment in which we artificially delay the starting time of some of the flows of a given query, following the distribution of stragglers reported in the literature [AKG+10]. The results in Figure 3.14 show that the benefit of NetAgg decreases linearly with the amount of stragglers. This is expected as a higher number of stragglers allow for less opportunities to aggregate data. However, NetAgg is still able to outperform the edge-based solutions.

### 3.4.2 Testbed-driven evaluation

Next we describe the results of the testbed experiments designed to evaluate the effectiveness of our NetAgg middlebox prototype. We use the two applications described in §3.3.3, i.e. Solr and Hadoop MapReduce. The two applications set different objectives for NetAgg: increased request throughput (Solr) and reduced job completion time (Hadoop). We first evaluate the applications in isolation and then run them simultaneously on a shared Agg box.

**Testbed set-up.** Our testbed consists of 34 servers arranged in two racks. Each rack contains one server with two 8-core Intel Xeon CPUs at 2.9 Ghz and 32 GB of RAM, acting as a master node, and ten servers with a 4-core 3.3 Ghz Intel Xeon CPU and 8 GB of RAM, acting as the workers. In addition, each rack has five servers with a 4-core AMD Opteron CPU at 1.8 GHz. These less powerful machines act as the clients for the Solr use case and generate the queries. All servers have 1 Gbps Ethernet network links to their ToR switches. Agg boxes have equal specifications as the master nodes and are attached to the ToR switches via 10 Gbps links.

**Performance of local aggregation tree.** We first conduct a micro-benchmark that evaluates the processing performance of differently-sized local aggregation trees. We use one of the Agg box servers and create an \( n \)-input local aggregation tree, which is fed data from
the Hadoop WordCount workload (see below). Data is read from memory to avoid processing to be limited by the throughput of the network connection. The aggregation ratio $\alpha$ is 10%.

Figure 3.15 shows the processing throughput for different thread pool sizes when varying the number of leaves in a tree ($L$). We consider only binary trees and hence the number of internal nodes, i.e. the aggregation tasks, is equal to $L - 1$. We observe that 8 leaves are sufficient to saturate a 10 Gbps link. When the number of leaves increases, more aggregation tasks can be executed concurrently, and the throughput increases accordingly. Additionally, we also observe that a thread pool equal to the size of the cores yields the best performance for all numbers of leaf nodes. Adding more threads beyond that number does not provide further benefit.

Apache Solr

We begin our testbed evaluation with Apache Solr deployed in a single-rack with 1 frontend, 10 backends and 5 client servers. We create an index over all Wikipedia pages from June 2012 [Wik12b]. Each client continuously submits a query for three random words, and Solr returns the most relevant Wikipedia pages matching the search terms. To generate different workloads, we vary the number of concurrent clients generating requests. Unless stated otherwise, error bars indicate the 5th and 95th percentiles.

We use two aggregation functions with different computational costs, representing extremes for functions commonly used in search engines. The first function, \texttt{sample}, has low computational cost. It returns a randomly chosen subset of the documents to the user according to a specified output ratio $\alpha$. The second function, \texttt{categorise}, is CPU-intensive. It classifies Wikipedia documents according to their base categories [Wik12a] and returns the top-$k$ results per category. Each base category contains several sub-categories, and categorisation is performed by parsing the document content for category strings and determining the base category for the majority of strings. Parsing is equivalent to evaluating a regular expression over the text content of the returned documents, which is expensive. $k$ determines the output ratio.
### 3.4 Evaluation

**Figure 3.17:** Latency with varying number of Solr clients

**Figure 3.18:** Throughput with varying output ratio

**Throughput.** Figure 3.16 shows the median throughput with an increasing number of Solr clients, comparing Solr with and without NetAgg. To test the capability of NetAgg to process at line rate, we use the sample function for aggregation with a fixed output ratio of $\alpha = 5\%$ to prevent the link to the frontend from becoming a bottleneck.

For plain Solr, the throughput with 5 clients is limited by the client workload. As the number of clients increases, the number of queries and hence, the throughput also increases until the 1 Gbps link to the frontend is saturated. With NetAgg, the throughput grows steadily up to 50 clients and then starts to saturate until it becomes bound by the 10 Gbps link to the Agg box. We can see that the Agg box is able to aggregate data at a median throughput of 9.2 Gbps for 70 clients, corresponding to a $9.3 \times$ speed-up compared to Solr.

**Latency.** Besides improving the throughput, NetAgg also reduces the query latency. Figure 3.17 shows the 99th percentile of the response times when varying the number of clients. For Solr, as the throughput of the frontend is limited to 1 Gbps, response times rise significantly when the workload increases. NetAgg can serve a higher load with low response times: with 70 clients, the response time for Solr increases from 0.6 s to 5.1 s, while with NetAgg it only increases from 0.2 s to 0.4 s. This shows that NetAgg achieves low response times because it removes congestion from the network link to the frontend.

**Output ratio.** Next we vary the output ratio $\alpha$ of Solr search queries with the sample aggregation for a fixed client population of 70. Figure 3.18 shows the throughput for Solr and NetAgg. Since the performance of Solr is network-bound, the throughput does not depend on the output ratio. As we have demonstrated in simulation, a higher output ratio limits the benefits of NetAgg as less opportunity for aggregation exits. At a ratio of 100%, the performance of plain Solr and NetAgg are equal. However, reduction ratios typically vary between 5% to 10% and are at most 40% (see §2.2.2), in which case NetAgg can offer significant benefit.

**Two-rack deployment.** To investigate the performance of NetAgg with multiple Agg boxes, we extend our set-up to two racks, each with one Agg box. We also add a second Solr...
deployment to generate a higher request load. Each rack hosts one Solr frontend, and we vary the number of backends per Solr deployment. Backends for a single deployment are divided equally between the rack. For example, 4 backends means that each Solr deployment uses 4 backends of which two are located in the first rack and two in the second. This generates cross-rack traffic and both Agg boxes have to aggregate data from both deployments.

Figure 3.19 compares the throughput of a single Agg box in a one rack deployment to the aggregate throughput of two Agg boxes in the two rack case. For both configurations, throughput scales linearly with the number of backends. With two racks, the Agg boxes serve twice as many backends compared to one rack, and hence their aggregate throughput nearly doubles. This shows the ability of NetAgg to operate in larger deployments.

**Scale out.** To show the feasibility of increasing the performance of NetAgg by connecting multiple Agg boxes to a single switch, we conduct an experiment in which we compare the performance of a single Agg box to two Agg boxes attached to the same switch. Requests are load balanced between the Agg boxes by hashing request identifiers. We use the computationally-expensive categorise aggregation function to ensure that the Agg box is the bottleneck. The results in Figure 3.20 confirm our expectation. While a single Agg box can only reach a processing rate of approximately 5Gbps due to the computational complexity of the categorise function, two Agg boxes are able to double the throughput and again saturate the 10Gbps link.

**Scale up.** To demonstrate the effectiveness of data-parallel processing on Agg boxes, we increase the number of active CPU cores on a single Agg box from 1 to 16 for both of our aggregation functions. The results in Figure 3.21 show that, while the performance of the computationally-inexpensive sample function is network bound, the performance for the categorise function increases linearly with more CPU cores due to increased parallelism. Hence, it is possible to improve the performance of an Agg box, e.g. after a network upgrade, by upgrading the Agg box server.
3.4 Evaluation

![Figure 3.21: Throughput for different number of CPU cores](image1)

![Figure 3.22: Throughput and completion time for different MapReduce jobs](image2)

Apache Hadoop

Next we investigate the performance of NETAGG when used with Apache Hadoop MapReduce as an example for a batch processing application. We deploy Hadoop in one rack with 8 mappers and 1 reducer. The workload consists of a set of benchmarks: (i) WordCount (WC), counting unique words in text; (ii) AdPredictor (AP), a machine learning job for generating click-through predictions from search engine web logs [GCBH10] [KDD12]; (iii) PageRank (PR), an implementation of the PageRank algorithm [HHD+10]; (iv) UserVisits (UV), a job for computing ad revenue per source IP address from web logs [AGN+13]; and (v) TeraSort (TS), a sorting benchmark with an identity reduce function [HHD+10].

Job completion time. We deploy each benchmark job on plain Hadoop and on Hadoop with NETAGG. For each job, we measure the total shuffle and reduce time relative to Hadoop, which consists of the time it takes to shuffle the data across the network, aggregate it at the Agg box and write the job output at the reducer. Additionally, we separately report the time spent on aggregation, the overall job completion time including the map phase, and the processing throughput of the Agg box for each job. Figure 3.22 shows the shuffle and reduce, aggregation, and job completion times (normalised with respect to plain Hadoop) and the Agg box processing rate. NETAGG is able to speed up the shuffle and reduce time by up a factor of up to $4.5 \times$ compared to plain Hadoop. Only for TS, there is no benefit as sorting does not involve data reduction. Besides TS, AP exhibits the lowest speed-up of $1.9 \times$ because the benchmark is CPU bound. As the map phase is not affected by NETAGG, the improvements for the overall job completion times are lower compared to the shuffle and reduce times. However, we can still observe speed-ups of up to $1.85 \times$.

Notably, in all cases, the time spent on aggregation at the Agg box is only a small fraction of the total shuffle and reduce time. This is due to the fact that the reducer is unaware that the results received from the Agg box are already aggregated and hence reads them again. This is a conscious design decision because one of NETAGGs requirements is transparency. However, a more invasive system that modifies the application to prevent unnecessary processing at the
Chapter 3. Traffic Reduction for Aggregatable Traffic

![Graph showing completion time for different input data sizes](image)

**Figure 3.23:** Completion time for different input data sizes

The reducer could improve the shuffle and reduce time further, bringing it closer to the aggregation time. The figure also shows that the Agg box processes traffic at around 6 Gbps for almost all jobs. Only the processing rate for TS is bound by the 1 Gbps link to the reducer as there is no data reduction.

**Data size.** We explore how the performance benefit of NetAgg is affected by the amount of data processed by the reducer. We use the WC benchmark and create synthetic datasets of different sizes in which we vary the number of word repetitions such that the output ratio is always 10%. Figure 3.23 shows the absolute shuffle and reduce times for different intermediate data sizes, ranging from 8 GB to 128 GB. The shuffle and reduce time of plain Hadoop increases linearly with the data size as more data has to be sent across the network. The shuffle and reduce time with NetAgg also increases linearly but significantly more slowly, yielding a speed-up of $5.2 \times$ for 128 GB. Early reduction also saves CPU cycles and disk I/O at the reducer, which adds to the benefit of NetAgg.

**Multiple applications**

In our final experiment, we evaluate the behaviour of NetAgg in the presence of multiple deployed applications, focusing on the fairness that the adaptive task scheduler of an Agg box can achieve. For this, we execute a Hadoop job on NetAgg while running a Solr deployment at the same time. We route all aggregation traffic through a single Agg box and record the CPU share of each application by measuring the time, the tasks of each application spent running, during a fixed interval. As the resource consumption of a single Solr task is significantly higher than that of a Hadoop task, the NetAgg scheduler must be able to account for this heterogeneity.

Figure 3.24a shows the CPU usage of the two applications when the task scheduler uses regular weighted fair queueing with fixed weights, which are set according to application priorities (see §3.3.1). Although each application is assigned a desired utilisation of 50%, this is not reflected in the achieved CPU usage: a Solr task takes, on average, 34 ms to run on the CPU, while a Hadoop task runs only for 2 ms. This translates to actual shares of 95% for Solr.
3.5 Summary

This chapter presented NetAgg, a system to perform in-network aggregation during partition/aggregate queries. NetAgg makes use of the tree topology of data centre networks and existing middleboxes to construct aggregation trees in the network along which query result data can be partially aggregated. These partial aggregation steps occur at every hop/switch and thereby continuously reduce traffic as it flows through the network. While the rate of reduction depends on the query, a large class of queries shows output ratios of less than 5%, which can significantly benefit from NetAgg.

NetAgg consists of Agg boxes and shim layers. Agg boxes expose high-level interfaces to deploy aggregation functions from existing frameworks. Underneath, Agg boxes parallelise aggregation across multiple CPU cores using local aggregation trees, which allows them to achieve processing rates up to 9.2 Gbps when data arrives from the network. Shim layers transparently intercept traffic at the socket layer, which is a common layer across most data-parallel frameworks. They forward traffic to Agg boxes and enforce the correct protocol at the master. This allows to deploy NetAgg with existing frameworks without any changes. NetAgg’s generic design allows to support new frameworks with only little additional effort.

NetAgg realises one aspect of network-awareness as the physical structure and hardware of the network are used for application-specific processing. We demonstrated in this chapter that the network-aware solution is superior to existing edge-based solutions, which shows the benefit of considering more advanced aspects of the network for large-scale data processing.

Figure 3.24: Comparison of scheduling approaches in NetAgg

(a) Non-adaptive fair sharing
(b) Adaptive fair sharing

and only 5% for Hadoop. Fixed weights therefore lead to a starvation of the Hadoop tasks. In contrast, the adaptive scheduler continuously measures the task execution times and is able to detect and correct the shares dynamically. Figure 3.24b shows the same set-up using the adaptive scheduler, which is able to achieve a fair split between the two applications.
Traffic Balancing for Non-Aggregatable Traffic

In this chapter, we introduce SQUIRRELJOIN, a novel join algorithm that is able to adapt to network skew by dynamically changing the data partitioning during the shuffle phase of a join. SQUIRRELJOIN measures network throughput to detect network skew and uses lazy partitioning to adapt the partitioning without state migration. This prevents stragglers due to imbalances in network usage and speeds up join processing in shared data centre environments. Additionally, it maintains a high network utilisation as it efficiently uses the available resources.

4.1 Overview

SQUIRRELJOIN is a join algorithm designed for running in environments in which the network is shared between different applications. As this can cause performance degradation due to skewed accesses and straggling workers (see §2.2.3), SQUIRRELJOIN detects such situations by measuring the throughput of each worker and estimates their completion times. It then redistributes the data such that slower workers receive less, and faster workers receive more data to even out the skew and have workers complete at the same time. This balancing of skew not only significantly reduces the join completion time under network skew conditions, but also helps keep a high overall network utilisation as all links are evenly used. Compared to existing approaches that try to dynamically react to skew using state
Chapter 4. Traffic Balancing for Non-Aggregatable Traffic

migration [AH00] [SHCF03] [EEVK14], SQUIRRELJOIN avoids state migration, and thereby additional network traffic, by employing a new technique called lazy partitioning.

The key idea behind lazy partitioning is to retain parts of the input data at sending workers instead of sending out the data as soon as it is read. While reading input data, join workers partition some of the records into a large number of small partitions called lazy partitions. These lazy partitions are stored in main memory buffers at the workers\(^1\). Each worker uses the same partitioning scheme (e.g. hash partitioning) to generate its lazy partitions and hence, each worker creates an equal set of lazy partitions and lazy partition \(i\) contains the same subset of keys on each worker.

Lazy partitions can be assigned to receiving workers to be joined. Until a lazy partition is assigned, it is guaranteed that no receiving worker receives a record with a key within this lazy partition. That means that no join state for any of the keys of an unassigned lazy partition is accumulated. As a result, lazy partitions can be flexibly assigned to any receiving worker without migrating existing state. SQUIRRELJOIN uses this mechanism to efficiently redistribute data based on network skew measured during the runtime of the join. SQUIRRELJOIN remembers assignment decision to ensure that any subsequent records after the assignment are sent to their correct receivers.

While network skew can slow down either receiving or sending workers, SQUIRRELJOIN only focuses on receiver-side skew because our theoretical analysis shows that any mitigation of sender-side skew can offer only a limited benefit in practice. To mitigate receiver-side skew, SQUIRRELJOIN assigns proportionally more lazy partitions to faster receivers.

4.1.1 Design challenges

To realise SQUIRRELJOIN, we overcome three design challenges as discussed below:

**Accurate skew detection.** To make correct assignment decisions, SQUIRRELJOIN requires a mechanism to accurately measure the progress of each receiving worker and compute their projected completion times. If these projections are wrong, SQUIRRELJOIN will wrongly assign lazy partitions that might generate imbalances in an otherwise balanced case. As SQUIRRELJOIN takes online measurements, these can be distorted by noise due to various reasons such as TCP’s slow convergence [AGM+10], and hence SQUIRRELJOIN must smooth the measurements and filter potential noise. To achieve this, SQUIRRELJOIN employs basic statistical testing and signal smoothing techniques.

**Optimal lazy partition assignment.** When network skew is detected, SQUIRRELJOIN needs to compute an assignment of lazy partitions to nodes. This computation has to be

\(^1\) Hence the name SQUIRRELJOIN, as this approach is similar to how squirrels stash away food before winter.
efficient, as reaction latency should be as low as possible, and effective, such that existing imbalances even out. SQUIRRELJOIN uses a lightweight bin-packing heuristic, which can compute an assignment in linear time.

**Minimal overhead.** In case there is no network skew, SQUIRRELJOIN’s lazy partitioning mechanism should not incur significant overhead. This comes with two challenges: (i) the lazy partitioning mechanism itself cannot add overhead to the actual record processing and should be as lightweight as possible; (ii) lazy partitions should be assigned or consumed as quickly as possible in order to not waste resources by holding back too much data. However, there should also be enough data left in lazy partitions such that any occurring network skew can still be balanced. SQUIRRELJOIN’s architecture decouples the control and data plane of lazy partitions and minimises the amount of necessary synchronisation between workers, making it highly efficient. Additionally, it provides a set of parameter settings to keep the balance between storing and consuming lazy partitions.

### 4.1.2 Design space

SQUIRRELJOIN is an adaptive approach, which reacts to network skew at runtime. This is in contrast to existing data skew mitigation approaches that determine the key distribution a priori and define an even partitioning based on that information [RMU+14] [PSR14] [LNN14]. As data skew is a static property when processing a finite amount of data, these approaches can effectively balance the processing. Network skew, however, occurs dynamically and hence is hard to predict.

Dynamic skew mitigation approaches, on the other hand, [AH00] [SHCF03] [EEVK14] also perform runtime adaptation. As the main target of these approaches is streaming joins on (infinite) streams of data, the key distribution can change over time and hence joins need to be able to change the partitioning dynamically. However, all of these approaches rely on state migration but, under network skew, an approach that migrates data between workers is not possible as the network itself is the cause of the skew and thus a constrained resource.

The dynamic approaches closest to SQUIRRELJOIN are Flux [SHCF03] and SkewTune [KBHR12]. Flux changes the arrival order of incoming tuples using a small sender-side buffer to avoid head-of-line blocking but does not repartition data, which only allows to balance short transient skew. SkewTune dynamically reassigns keys to different receivers when a slower worker is detected but requires input data to be sorted. However, in most cases, this assumption does not hold and hence, an additional sorting step before the join is required. This is expensive, in particular, if data cannot be sorted in memory. In that case, an external sort is required which incurs high I/O overhead as the data needs be read an additional two times. Using lazy partitioning, SQUIRRELJOIN can permanently repartition data while not imposing any restrictions on the ordering of the data.
Besides avoiding state migration, SQUIRRELJOIN exploits another property unique to network skew that is not considered by existing approaches. Due to full-duplex network links, network skew is unidirectional. That means that it can independently affect the sender or receiver portion of a worker node, contrary to, for example, CPU skew, which would slow down all processes of a worker simultaneously. We will show later that SQUIRRELJOIN can accurately detect network skew while not being affected by either data or resource skew. This means that existing techniques to mitigate other types of skew are compatible with SQUIRRELJOIN.

### 4.1.3 Contributions

This chapter makes the following contributions:

1. We formalise a model for key reassignment to show the relative effects of rebalancing network skew at senders and receivers. This analysis reveals the limited effectiveness of balancing senders using key reassignment and informs our use of a simple, yet effective, heuristic to balance receiver-side skew in SQUIRRELJOIN (§4.2).

2. We introduce lazy partitioning, a technique that allows to dynamically repartition data during distributed join processing without state migration. In addition, we present an efficient heuristic that can detect network skew and compute a lazy partition assignment to balance it (§4.3).

3. We describe SQUIRRELJOIN, a distributed join processing technique that uses lazy partitioning to react to network skew. SQUIRRELJOIN collects robust progress metrics to make accurate decisions about the assignment of lazy partitions. Additionally, it comes with a lightweight architecture to minimise the overhead of lazy partitioning (§4.4).

We implement SQUIRRELJOIN in Apache Flink [Fli16a] and evaluate it under a variety of network skew scenarios on a shared cluster, with interfering real-world applications and synthetic micro-benchmarks (§4.5). Our results show that SQUIRRELJOIN achieves speed-ups of up to $2.3 \times$ under network skew, while only adding a fixed 10 s increase in job completion time without network skew.

### 4.2 Analysis of network skew in joins

We present our formal model of a distributed repartition-based join and extend the model to include network usage. We then model the maximum speed-up a reassignment-based approach can achieve when stragglers are caused by network skew known a priori on both the receiver and sender sides. The goal of this analysis is to inform our skew mitigation approach.
4.2 Analysis of network skew in joins

4.2.1 Distributed joins

We consider a repartition join on a deployment of $n$ nodes. Each node executes a worker from a set of workers, $W$, which perform the join computation. A worker consists of a sender and a receiver process (see Figure 4.1). Sender processes read data records from the local disk and send them to the receiver process on the worker that is responsible for joining them.

The data records are split into partitions. We denote the collection of all partitions as $P = \{p_1, p_2, \ldots, p_x\}$ where $p_i$ is an individual partition. We use $P^S$ and $P^R$ to refer to partitions that are sent and received, respectively. These sets are identical, i.e. $P = P^S = P^R$, but the extended notation allows us to specify partitions that are sent from a node $i$ through use of $P^S_i$. The partitions in this set need not match the set of partitions that the same node receives, $P^R_i$. We separate the sender and receiver assignments to allow for a more robust analysis based on the unidirectionality of network skew, as mentioned in §4.1.2.

Each partition must be handled by a sender:

$$P = P^S \land \forall p \in P, \exists i \in W : p \in P^S_i \land \forall i \in W : P^S \supseteq P^S_i$$

Additionally, no partition appears on more than one sender:

$$\forall i, j \in W, i \neq j : P^S_i \cap P^S_j = \emptyset$$

Each partition is also destined for exactly one receiver, so symmetric constraints exist for $P^R$. In a traditional repartition join, each sender has one partition per receiver. Our analysis assumes the case in which data is evenly distributed across partitions:

$$\forall p_i, p_j \in P : |p_i| = |p_j|$$

and partitions are evenly distributed across senders and receivers [XKZC08]:

$$\forall i, j \in W : |P^S_i| = |P^S_j| \land |P^R_i| = |P^R_j|$$

This can be achieved by balanced partitioning approaches [RMU+14].

4.2.2 Network usage

We extend the above model for distributed joins to include network effects. Workers must send and receive partitions to and from each other over the network. In a full duplex network, sending and receiving are independent, and we model the network as two separate channels. The channel with the outgoing data is associated with the sender process of the worker, and the channel with the incoming data is associated with the receiver process. This ignores
communication between sender and receiver processes of the same worker because they communicate via inter-process communication (IPC).

Modelling the join traffic over network channels allows us to identify the location and severity of network skew. Network skew exists because other applications in the cluster may share the available network bandwidth at nodes that host workers. If there is too much traffic on a channel, it becomes a congested channel, reducing the bandwidth available to the associated worker. More congestion means less available bandwidth. We assume that other applications are not evenly distributed across the nodes, which means that not every worker necessarily experiences the same congestion.

We define the link potential \( L \) of a network channel as the amount of data that the channel can transfer over the complete duration of the join, i.e. \( L = C \times T \) where \( C \) is the capacity of the link and \( T \) is the total join completion time. We denote the link potential of the receive channel of worker \( i \) as \( L^R_i \) and of its send channel as \( L^S_i \). The network channel utilisation \( U \) is the amount of data sent over the channel during the join. The utilisation of worker \( i \) is denoted by \( U^S_i \) and \( U^R_i \) for send and receive channels, respectively. Based on this, the channel utilisation ratio is the ratio between a channel’s utilisation and its potential. For example, the utilisation ratio for the receiver channel on a worker \( i \) is given by \( U^R_i / L^R_i \), which equals the fraction of the network channel used by the join. If the ratio is less than 1, the channel is either idle for parts of the join or parts of the channel’s traffic belong to other applications.

Figure 4.1 gives an example of a single congested channel on the receiver of worker \( b_1 \). Channels send and receive the same amount of data, and the straggling worker \( b_1 \) determines the total join completion time in this scenario, so all channels have the same utilisation:

\[
\forall i, j \in W : U^S_i = U^S_j = U^R_i = U^R_j
\]
4.2 Analysis of network skew in joins

Hence, the utilisation ratio of all channels is equal to the ratio at $b_1$. There is typically a single slowest straggling worker, denoted simply by $b$, which decides the utilisation rate of every process in the system. We denote the utilisation ratio of this worker $b$ as $B$:

$$B = \min \left( \frac{U^R_b}{L^R_b}, \frac{U^S_b}{L^S_b} \right)$$

Note that this model is independent of the actual limiting resource. For example, if local storage is disk-based and the disk bandwidth is less than the network bandwidth, we can set the link capacity to match the disk bandwidth. This does not affect the model as all results are derived relative to the maximum link capacity.

Reassigning partitions from congested to non-congested channels increases the channel utilisation using the spare capacity and hence decreases join completion time. However, there is a limit to how many partitions can be reassigned before another channel exceeds its spare capacity. At this point, any additional reassignment would increase the workload at either the old or new straggling worker, so this point represents the maximum achievable speed-up. Next we calculate the maximum number of partitions that can be reassigned before the utilisation of another channel reaches its capacity.

4.2.3 Receiver-side network skew

Congested channels at receivers occur when an external application hosted on that node receives incoming traffic. We use our model to determine the partition reassignment that provides the maximum decrease in join completion time. We first assume that we have $m$ receive-side stragglers of the same magnitude, i.e. the bandwidth is limited by the same amount at each of these workers. We call the set of straggling receivers $M = \{b_1, b_2, \ldots, b_m\}$. Given that all straggling receivers are equal, we can use any one of them to determine $B = U^R_b / L^R_b$, $b \in M$. The $n - m$ other receivers and all $n$ senders can utilise the full network bandwidth for the join.

Our goal is to minimise the runtime of straggling workers caused by network skew by reassigning partitions from straggling to non-straggling receivers. We define the set of reassigned partitions as follows:

$$P^R_M = \bigcup_{i=1}^{m} P^R_{b_i} \setminus \hat{P}^R_{b_i}$$

$\hat{P}^R_{b_i}$ is the set of partitions received by $b_i$ after the reassignment. Partitions are evenly assigned among non-straggling receivers:

$$\forall p \in P^R_M, \exists i \in W \setminus M : p \in \hat{P}^R_i \land \forall i, j \in W \setminus M : |\hat{P}^R_i| = |\hat{P}^R_j|$$
Chapter 4. Traffic Balancing for Non-Aggregatable Traffic

The fraction of work reallocated from a receiver $b \in M$ is defined by

$$k = (U_R^b - \hat{U}_R^b) / U_R^b$$

As a result of the reallocation, $\hat{U}_R^b = U_R^b (1 - k)$.

Since $B$ is a constant imposed by external conditions, reducing $U_R^b$ reduces $L_R^b$ by the same factor. Given that $L_R^b = CT$, and $C$ is independent of the join, we deduce that reassigning partitions away from every straggling receiver reduces the total time of the job, $T$. This means the link potential of every channel is likewise affected.

The point at which the utilisation of another channel consumes its maximum capacity marks the maximum speed-up possible due to reallocation. To determine what value of $k$ causes this to happen, we analyse the utilisation of all other channels by determining their relative changes in utilisation. We note that, since all initial partition sizes are the same, we can implicitly substitute, e.g. $|P_R^i|$ for $|P_R^b|$, thus simplifying the calculation.

**Receivers on non-straggling workers.** Each receiver receives a fraction $(n - 1)/n$ of its partitions over the network, and the rest over IPC channels. A fraction of $k$ partitions from each of the $m$ straggling receivers is reassigned evenly across all non-straggling receivers, which increases the amount of received partitions by a worker $i \notin M$ by a factor of $km / (n - m)$.

We consider the portion of these new partitions that arrive via the network, $(n - 1)/n$. To determine the relative change in network channel utilisation, we add up the initial and additional assigned amounts and divide by the initial amount to obtain a relative utilisation increase of:

$$\hat{U}_i^R = U_i^R \left( 1 + \frac{km}{(n - m)} \right)$$

**Senders on straggling workers.** A sender on a straggling worker $b \in M$ sends partitions over the network which originally used IPC. Originally, $(n - 1)/n$ of the partitions on the worker were sent via the network, while the remaining $1/n$ of those partitions used IPC. Now an additional fraction $k$ of the locally sent partitions is also sent over the network, which increases the utilisation:

$$\hat{U}_b^S = U_b^S \left( 1 + \frac{k}{(n - 1)} \right)$$

**Senders on non-straggling workers.** All other senders shift additional partitions from the network to IPC, which decreases the utilisation of a sender on a worker $i \notin M$. Of the fraction of partitions on $i$ that were originally sent via the network $(n - 1)/n$, only a fraction have been reassigned to $i$ and now use IPC, $km / ((n - m)(n - 1))$. The remaining reassigned partitions still use network channels even as they are sent to new receivers. The relative
4.2 Analysis of network skew in joins

decrease in utilisation is thus

\[ \hat{U}_i^S = U_i^S \left( 1 - \frac{km}{(n-m)(n-1)} \right) \]

Non-straggling receivers have the highest relative increase in utilisation. Thus, to find the value of \( k \) that maximises the reduction in job completion time, we find when \( \hat{U}_i^R = \hat{L}_i^R \) for \( i \notin M \):

\[ U_i^R \left( 1 + \frac{km}{(n-m)} \right) = \frac{U_i^R}{B} (1 - k) \implies k = \frac{1 - B}{Bm - B + 1} \]  \hspace{1cm} (4.1)

It is sufficient to consider any single non-straggling receiver because they all have the same initial utilisation and increase by the same relative amount.

**Example.** Consider a 32-worker deployment in which one receive-side straggler is limited to half the channel bandwidth \((B = 0.5)\), leading to a 2\( \times \) slow-down. Equation 4.1 suggests a potential 49.2% improvement from partition reassignment. Reallocation more partitions from the original straggling receiver will slow down the new receivers of the reallocated data, while reallocating fewer causes the straggling receiver to be slower. For multiple receiver-side stragglers \((m > 1)\), the benefit only slightly decreases, e.g. for \( m = 5 \), the maximum improvement is still 45.7%. Note that an improvement of 50% would result in a join completion time equal to that of a system without skew. Achieving this is not possible because, after the reassignment, non-straggling workers must process more data than initially assigned.

It is rare for network skew to be uniform. Rather than having no traffic at non-straggling workers and the same amount of traffic at all congested channels, it is more likely that there exist different levels of congestion, i.e. each congested node has a different utilisation ratio. In that case, the maximum speed-up is bound on the lower end by the case in which all channels are congested equally to the congested node with the lowest utilisation ratio. The upper bound is the case in which all channels are congested equally to the congested node with the highest utilisation ratio.

### 4.2.4 Sender-side network skew

We now explore the case in which the join is limited by skew congesting sender channels. We start with the case of \( m \) sender-side stragglers, \( M = \{b_1, \ldots, b_m\} \), which are limited to the same utilisation ratio by background traffic:

\[ B = \frac{U_{b_i}^S}{L_{b_i}^S} = \frac{U_{b_j}^S}{L_{b_j}^S} \quad \forall b_i, b_j \in M \]
Partitions stored at specific workers cannot be assigned to other senders as they still would have to be sent out by the worker that stores it. Hence, the only way to reduce the load at a sender-side straggler $b \in M$ is to reassign partitions to the receiver process on the same worker, allowing more partitions in $P_S^b$ to use IPC. We reallocate a fraction $k = (U_S^b - \hat{U}_S^b)/U_S^b$ of work from each straggling sender $b \in M$, which reduces their utilisation by a factor of $(1 - k)$ each, i.e. $\hat{U}_S^b = U_S^b(1 - k)$. Analogously to receive-side stragglers, this also reduces the channel’s capacity by $(1 - k)$, affecting completion time and all other channel capacities. We again analyse the effect of this reassignment on all channels to optimise $k$.

**Senders on non-straggling workers.** All partitions originally sent via the network on non-straggling workers are still sent via the network, even if the destination has changed. An additional fraction $km(n-1)/(n-m)$ of the partitions originally sent using IPC is now also sent over the network. This factor captures that it is possible to reassign partitions evenly only from receivers on workers that are not in $M$, i.e. from $P^R \setminus \bigcup_{i=1}^m P^R_b$. If we reassign $p \in P_S^b$ for any $b \in M$, it would increase the sending load of $b$. Given that only $(n-1)/n$ of the partitions sent by a worker are sent via the network and $1/n$ are sent through IPC, we compute the relative utilisation increase for a non-straggling sender $i$:

$$\hat{U}_i^S = U_i^S \left( 1 + \frac{km}{n-1} \right)$$

**Receivers on straggling workers.** Each receiver on a straggling worker $b \in M$ originally receives a fraction of $(n-1)/n$ partitions over the network. After the reassignment, each $b$ receives a fraction of $k$ additional partitions, of which $(n-1)/n$ arrive from the network. Hence, the utilisation increases for these receivers:

$$\hat{U}_b^R = U_b^R(1 + k(n-1))$$

**Receivers on non-straggling workers.** Partitions are assigned evenly from all non-straggling workers $i \notin M$. In order to assign enough partitions to all workers $j \in M$, $k(n-1)/(n-m)$ of the partitions of a receiver on a non-straggling worker are reassigned to each of the $m$ stragglers. From this, we calculate the relative decrease in utilisation:

$$\hat{U}_i^R = U_i^R \left( 1 - km \frac{(n-1)}{(n-m)} \right)$$

We find that, for $0 < m < n$, the utilisation of receivers on workers with straggling senders has the highest relative increase. Hence, we solve for $k$ given $\hat{U}_b^R = \hat{L}_b^R$ for any $b \in M$:

$$U_b^S(1 + k(n-1)) = \frac{U_b^S}{B} (1 - k) \iff k = \frac{1 - B}{B(n-1) + 1} \quad (4.2)$$
Example. If we consider a 32-worker deployment and a 50% capacity decrease at a single sender-side straggler, according to Equation 4.2, the maximum reduction in job completion time is only 3.0%. The total number of partitions that must be reassigned is $P^R_b mk$. However, there are only $P^R_i (n - m)/n$ partitions available at non-congested workers to redistribute. If $mkP^R_b > P^R_i (n - m)/n$, $k$ is limited by $(n - m)/mk$, resulting in even less improvement.

We again intuitively extend to the case when congested channels are not equally congested. We are not able to improve this case any more than if there was a single straggling worker with congestion equal to the most congested channel. A lower bound is harder to determine because of the possibility that there are not enough partitions on non-congested workers.

4.3 Lazy partitioning

Next we present our lazy partitioning approach for adapting to network skew in distributed joins. We first give an overview of lazy partitions. After that, we describe how network skew is detected and rebalanced and how lazy partitioning behaves without skew. Based on the outcome of the analysis from §4.2, we focus on receiver-side skew.

4.3.1 Lazy partitions

The main idea of lazy partitioning is to buffer input data records at the sender processes in memory instead of sending them out immediately. These buffered records are assigned later in such a way as to mitigate network skew. As distributed joins are stateful and require data records with the same keys to be processed by the same workers, lazy partitioning must ensure that this property holds when assigning buffered records.

The buffered data records are partitioned into lazy partitions. Lazy partitions hold a subset of the data records determined by the join key. Initially, all data records are written to lazy partitions until a first assignment decision is made. $P^R_\emptyset$ denotes lazy partitions that have not yet been allocated to receiver processes.

Assigning a lazy partition, $p \in P^R_\emptyset$, to a receiver process $i$ removes all other partitions with the same join keys across workers from $P^R_\emptyset$ and adds them to $P^R_i$. When a lazy partition is assigned, all buffered records are sent to the same receiver process, and no future records are buffered in the assigned partition anymore. At the end of the job, all lazy partitions are assigned and sent to receiver processes, i.e. $P^R_\emptyset = \emptyset$, which guarantees correctness.

To support fine-grained assignment decisions, the number of lazy partitions is much larger than the number of workers ($|P^R_\emptyset| \gg n$), but there is a trade-off. With too many lazy partitions, e.g. one per distinct join key, it is more expensive to maintain past assignments because each sender must store the assigned receiver for each lazy partition. We found that
significant network skew requires large amounts of keys to be reassigned to be balanced effectively. Hence, very fine grained decisions are unnecessary, and a smaller number of lazy partitions is sufficient (we empirically use 3200).

Figure 4.2 shows an example of lazy partitioning. Keys $k_4$, $k_5$ and $k_6$ are the subset of join keys buffered in lazy partitions, while keys $k_1$, $k_2$ and $k_3$ are already assigned to receiver processes. Data records with keys $k_4$, $k_5$ and $k_6$ are therefore kept by the sender processes, partitioned into two lazy partitions, which can be assigned later to balance network skew. When a sender process reads a new data record, it either buffers or sends it based on whether the record’s corresponding lazy partition has been assigned. The join keys in a lazy partition are the same at each sender process, and they are withheld from all receiver processes.

Lazy partitions are assigned in two cases: (i) when network skew is detected and a rebalancing decisions is made and (ii) when the lazy partitions at a worker have exhausted the memory available to them. We discuss both cases in detail below.

### 4.3.2 Detecting and balancing network skew

Correctly deciding when lazy partitions should be assigned to mitigate network skew is critical. *Late skew detection* policies [KBHR12] make decisions after sender or receiver processes finish. Work from processes still executing may be assigned to finished processes. Such an approach prevents reacting to transient skew conditions, which balance out throughout the job duration, but it requires processes complete independently, which is not the case in a distributed join. Due to head-of-line blocking [SHCF03], processes slow down but do not finish until straggling
Algorithm 1: Detect skew and compute assignment

Input: $W$: set, $P^R$: set, $\tau_{skew}$: const, $\tau_{assign}$: const, $t$: now

1. $c_{\text{slowest}} \leftarrow \max_{i \in W} \left\{ \frac{\rho(P^R, t)}{r(i, t)} \right\}$
2. $W_{\text{finished}} \leftarrow \emptyset$
3. if $\min_{i \in W} \left\{ \frac{\rho(P^R, t)}{r(i, t)} \right\} < c_{\text{slowest}} \cdot (1 - \tau_{skew})$ then
   foreach $i \in W$ do
      $c_i \leftarrow \frac{\rho(P^R, t)}{r(i, t)}$
   while $i \notin W_{\text{finished}} \land \exists \{p_1 \ldots p_W\} \in P^R_\emptyset$ do
      $c'_i \leftarrow \frac{\rho(P^R, t)}{r(i, t)} + \left\{ \frac{\rho(P^R, t')}{r(i, t')} \right\}$
      if $c'_i > c_{\text{slowest}} \lor c'_i - c_i > \tau_{assign}$ then
         $W_{\text{finished}} \leftarrow W_{\text{finished}} \cup \{i\}$
      else
         $P^R_\emptyset \leftarrow P^R_\emptyset \setminus \{p_1 \ldots p_W\}$
         $i \leftarrow i \cup \{p_1 \ldots p_W\}$

workers have completed. As a result, all processes complete at the same time, i.e. the join at all nodes is delayed by the slowest worker. To effectively rebalance network skew, assignment decisions must be made as early as possible.

As we focus on receiver-side skew due to our earlier analysis, we need a measure of progress for each receiver process. Receiver processes continuously monitor their throughput in terms of received data records per second and use it to estimate their completion time. Significant differences in these estimates indicate skew and trigger an assignment decision.

Each worker maintains a counter $\gamma$ of the number of processed data records. We denote the number of records left to receive by worker $i$ at time $t$ in the assigned partitions $P^R_i$ by $\rho(P^R, t)$. Assuming that the total number of input records is known in advance, e.g. from statistics maintained by a query optimiser, it is possible to compute

$$\rho(P^R_i, t) = \rho(P^R_i, 0) - \gamma.$$ 

Using $\rho$, the rate $r(i, t)$ of a worker $i$ at time $t$ is

$$r(i, t) = (\rho(P^R_i, t) - \rho(P^R_i, t'))/(t - t'),$$

where $t'$ is the time of the last measurement. We denote $t - t'$ as the assignment interval $\tau_{assign}$.

The completion time $c_i$ is then given by:

$$c_i = \frac{\rho(P^R_i, t)}{r(i, t)}$$
Chapter 4. Traffic Balancing for Non-Aggregatable Traffic

Based on the estimated completion times of workers, we can now compute a lazy partition assignment. The assignment algorithm uses a packing heuristic that assigns lazy partitions to faster workers until the new estimated completion time either equals that of the slowest workers or exceeds a specified threshold. Algorithm 1 shows how to detect skew and compute the lazy partition assignment. It takes as input the set of all workers, the set of yet unassigned lazy partitions, and two sensitivity thresholds, $\tau_{\text{skew}}$ and $\tau_{\text{assign}}$. The algorithm is executed by a single master node and its result broadcast to all workers.

First, the slowest worker (line 1) is compared to the fastest worker (line 3). Repartitioning only occurs if the estimated completion time of the fastest worker is less than the skew threshold, $\tau_{\text{skew}}$, defined as a fraction of the slowest worker. $\tau_{\text{skew}}$ controls the sensitivity of the approach: smaller values cause more rebalancing decisions, while larger values cause slower reaction.

If network skew is detected, the algorithm makes an assignment based on the current completion time estimates. Since the senders use the same partitioning scheme, the algorithm assigns the equivalent partition from each sender (e.g. LP 1 in Figure 4.2) to ensure the same worker receives all records that must be joined with each other (line 6). A new estimated completion time is computed for a worker, assuming the given set of lazy partitions is assigned to it (line 7). If the estimated completion time exceeds that of the slowest worker or the cumulative increase for this round exceeds a threshold, $\tau_{\text{assign}}$, the worker is marked as finished for this round (lines 8–9); otherwise the lazy partition is assigned (lines 10–12).

Using the assignment interval, $\tau_{\text{assign}}$, as an assignment threshold makes the algorithm robust to transient skew. Fully rebalancing perceived network skew can cause an imbalance if the skew disappears before the join computation finishes. In this case, faster workers may receive too many lazy partitions relative to currently straggling workers, while consuming lazy partitions that could be used to address such later occurring skew at a different worker. Using an assignment threshold avoids this case, and by setting it to the assignment interval, we always assign enough to last at least until the next assignment decision.

The new completion time of a receiver process is estimated based on the total size of the lazy partition to be assigned and the number of data records that the receiver process must still process (line 7). The total size of the lazy partition is obtained by linearly extrapolating its current size according to the fraction of input that has not yet been seen. For example, if a lazy partition contains $l$ records and 50% of the input data was processed, the lazy partition accounts for approximately $2l$ records once the whole input data was read.

Parameter discussion

Parameter $\tau_{\text{assign}}$. $\tau_{\text{assign}}$ specifies the time between assignment decisions. Ideally, it is small to make frequent assignment decisions and react to occurring network skew as fast as possible.
4.3 Lazy partitioning

Algorithm 2: Consume lazy partitions

\textbf{Input :} \( W: \text{set}, P_R^B: \text{set}, \tau_{\text{consume}}: \text{const}, t: \text{now()} \)

\begin{align*}
1 & \quad l \leftarrow \lceil |P_R^B| \cdot \tau_{\text{consume}} \rceil \\
2 & \quad c_{\text{slowest}} \leftarrow \min_{i \in W} \{ r(i, t) \} \\
3 & \quad q = \frac{l}{\sum_{i \in W} r(i, t)} \\
4 & \quad \text{foreach } i \in W \text{ do} \\
5 & \quad \quad l_i = \lceil q \cdot r(i, t) \rceil \\
6 & \quad \quad \text{while } l_i > 0 \land \exists \{p_1 \ldots p_{|W|}\} \in P_R^B \text{ do} \\
7 & \quad \quad \quad i \leftarrow i \cup \{p_1 \ldots p_{|W|}\} \\
8 & \quad \quad \quad l_i \leftarrow l_i - 1
\end{align*}

possible. However, if decisions are made too frequently, the risk of making wrong decisions
increases as not enough data can be collected to reliably estimate completion times. Hence,
\( \tau_{\text{assign}} \) should be as short as possible to allow fast reaction times while still ensuring robust
completion time estimates. We empirically found that 5s provides a good trade-off between
both goals (see §4.5.6).

\textbf{Parameter} \( \tau_{\text{skew}} \). \( \tau_{\text{skew}} \) specifies the maximum relative difference between the slowest and
the fastest worker before network skew is detected. When setting \( \tau_{\text{skew}} \), the trade-off between
sensitivity and robustness needs to be considered. If set too small, detection becomes highly
sensitive and is more susceptible to noise unrelated to network skew. On the other hand, a
too high value can miss significant network skew that needs to be rebalanced. We set \( \tau_{\text{skew}} \) to
5% by default, which gives robust behaviour in practice (see §4.5.6).

4.3.3 Consumption without network skew

When no network skew exists, lazy partitions must eventually be assigned to receiver processes
to ensure join completion. Each worker has a finite amount of memory \( M \) to store lazy
partitions. \( S_i \) denotes the \textit{lazy partition size}, i.e. the memory required for all lazy partitions
on worker \( i \). Once a worker has filled up its memory, \( \exists i \in W. S_i = M \), it must \textit{consume} a
subset of its lazy partitions to make room for more data records. In that case, a fixed number
of lazy partitions is assigned proportionally to all receiver processes.

Algorithm 2 formalises this consumption. It first determines the number of lazy partitions
that should be consumed (line 1) and then calculates the rate of the slowest worker (lines 2).
Each worker receives a weighted number of lazy partitions according to this ratio. First,
the average weight is computed (line 3) and then, for each worker, the amount to assign is
determined (line 5) and assigned (lines 6–8).

A \textit{consumption threshold}, \( \tau_{\text{consume}} \), specifies how many lazy partitions should be consumed
when a worker exhausts its memory. The threshold is specified as a percentage over all lazy
partitions that have not been assigned yet—as more lazy partitions are assigned, the absolute amount therefore decreases.

**Parameter discussion**

**Parameter** $\tau_{\text{consume}}$. $\tau_{\text{consume}}$ specifies the percentage of the remaining lazy partitions that should be consumed when a worker runs out of buffer space. Ideally, it is set to a small value to retain as many lazy partitions as possible for later use in rebalancing. Larger values waste more lazy partitions during consumption. However, if $\tau_{\text{consume}}$ is too small, workers consume lazy partitions too frequently as less memory is freed during consumption. This increases overhead. As we show empirically in §4.5.6, $\tau_{\text{consume}} = 5\%$ balances this trade-off.

$\tau_{\text{consume}}$ also determines the number of lazy partitions required. The consumption algorithm selects a set of lazy partitions based on their combined size (see Algorithm 2, line 1). If there are too few lazy partitions, at least one lazy partition cannot be assigned to each worker. Thus the number of partitions required has a lower bound of $|W|/\tau_{\text{consume}}$ when $P_{\emptyset}$ contains all records. As more assignments are made, the size of $P_{\emptyset}$ shrinks geometrically, so it is advisable to have more lazy partitions to ensure that every worker receives at least one lazy partition during later assignments. The upper bound is one partition per key, but there is a significant overhead in tracking the assignment decision for each key separately.

For example, with 64 workers and $\tau_{\text{consume}}$ of 5%, the minimum number of lazy partitions required is 1280. In our experiments, we use approximately triple this number (3200 lazy partitions) to stay well above this calculated minimum.

### 4.4 SQUIRRELJOIN implementation

We now describe SQUIRRELJOIN, an implementation of lazy partitioning that extends the standard hash-based repartition join algorithm in Apache Flink [Fli16a]. Our implementation of SQUIRRELJOIN overcomes two challenges: (i) how to maintain and consume lazy partitions without adding significant overhead on the critical data processing path of the join; and (ii) how to handle errors in the completion time estimates robustly.

#### 4.4.1 Architecture

SQUIRRELJOIN extends the *task-based* architecture of systems such as Flink or Spark. Each worker has multiple readers that read input data records in parallel and send them to the corresponding receiver processes.
Figure 4.3: SQUIRRELJOIN architecture (solid lines represent the data plane and dashed lines represent the control plane)

Figure 4.3 shows the SQUIRRELJOIN architecture consisting of (i) a *lazy partition coordinator* to make assignment decisions, and (ii) *lazy partition managers* to manage the lazy partitioning on workers. Workers have (i) *lazy partitions* to store records in memory; (ii) *lazy partitioners* to determine if data records should be stored in a lazy partition; and (iii) *lazy partition readers* to asynchronously consume lazy partitions after assignment.

**Lazy Partition Coordinator (LPC).** The SQUIRRELJOIN architecture uses a distributed master/worker approach for coordination. The LPC is part of the Flink master node and coordinates the lazy partition assignment. It periodically contacts the LPMs of workers and collects information on the current progress rates to detect and rebalance skew. It also polls LPMs to retrieve the lazy partition sizes and makes consumption decisions as per Algorithm 2 when workers exhaust memory (see Figure 4.3). After making an assignment decision, the LPC broadcasts it to all LPMs. The placement of the coordination logic at the master reduces the overhead of SQUIRRELJOIN on the workers and also removes the need for synchronisation between workers.

**Lazy Partition Managers (LPMs).** Each worker has an LPM to report aggregate statistics from local lazy partitions to the LPC. The LPM also stores assignment decisions received from the LPC. These are queried by local lazy partitioners to identify the receiver process for a data record whose lazy partition was already assigned.

**Lazy Partitioners.** Flink exploits the parallelism of multi-core CPUs by running multiple reader tasks at a worker. In SQUIRRELJOIN, each reader is associated with a lazy partitioner,
which decides if a data record should be stored in a lazy partition or sent to a receiver. This
decision must be done consistently across all readers on all workers. Lazy partitioners can
query the LPM to determine if the lazy partition for a given record has been assigned or
not. Adding data records to lazy partitions should not impact performance. Each reader
maintains its own set of lazy partitions to which it adds new records. This avoids the need for
synchronisation between reader tasks and reduces the latency added by the lazy partitioner.

Lazy Partitions. Lazy partitions are stored in a hash table with a unique identifier per
partition as the key and the stored records as the value. Lazy partitions share the same
identifier across workers. Records are stored in byte arrays in serialised form to reduce
instantiated objects and garbage collection overheads.

Lazy Partition (LP) Readers. After a lazy partition has been assigned, an LP reader
reads and sends records from the assigned lazy partitions to the specified destinations. By
consuming lazy partitions concurrently, readers do not require extra logic for receiving and
parsing assignment decisions, reducing delay on the critical path.

4.4.2 Mitigating estimation error

The LPC requires good time estimates to make good assignment decisions. Multiple effects
create errors in the estimation, such as the measurement granularity or transient network
effects. For example, we found TCP unfairness to be a major error source: TCP implements
max-min flow fairness, i.e. each network flow on a shared link receives an equal amount of
bandwidth, but TCP converges slowly under congestion [AGM+10]. Such estimation errors
lead to wrong rebalancing decisions, which can cause degraded performance. SQUIRRELJOIN
uses three simple yet effective techniques to mitigate estimation errors:

Sliding window average. The LPC takes regular throughput measurements. Single
measurements are noisy, so the assignment algorithm uses the average rate over the preceding
\( \tau_{assign} \) interval. After an assignment, there is a back-off period equal to \( \tau_{assign} \) before the next
assignment decision to ensure the throughput is measured after the previous assignment. If
no assignment is made, the assignment algorithm executes again after the next measurement.

Friedman test. The Friedman test [Fri37] filters statistically insignificant network skew.
We observe that, when the LPC measures worker progress rates, the individual rates may
be different, with some workers appearing faster than others, resulting in an ordering of
workers. This ordering, however, should change randomly between measurements if it is due
to measurement error, and not caused by a consistent phenomenon such as network skew.
The Friedman test determines if the ordering is consistent across multiple measurements or
randomly changes. We set the detection level of the Friedman test to 1%.
Skew thresholds. In addition to the relative skew threshold $\tau_{\text{skew}}$ (see §4.3.2), an absolute skew threshold is used to avoid addressing insignificant skew. Even if statistically significant skew is detected, and the relative difference is above $t_{\text{skew}}$, reacting can be wasteful if the expected absolute difference is small—instead lazy partitions could be used to mitigate more severe future network skew. Thus the LPC only reacts to network skew that is predicted to extend job completion time by more than 5 seconds.

4.4.3 Fault tolerance

The correctness of the join result depends on a consistent view of the assignment decisions across all nodes. In case of a node failure, SquirrelJoin needs to ensure the consistency of the assignment decisions. We distinguish to types of failures: (i) failure of the master in which case the LPC becomes unavailable and (ii) failure of a worker in which case the corresponding LPM becomes unavailable.

Master failure. Data-parallel frameworks deal with master failures by provisioning multiple standby master nodes [Fli17] [Spa17]. In case the current leading master fails, any of the standby masters can replace it. As assignment decisions are broadcast to and stored by the LPMs on every worker, the new master can query any worker to request the previously made assignment decisions and resume with a consistent view.

Worker failure. There are two scenarios in case a worker fails: (i) the remaining data of the failed worker is picked up by the other workers. In that case, all workers already store the correct assignment decisions and nothing needs to be done; (ii) a new worker is spawned. This worker can then query the master to obtain the most recent assignment decisions.

4.5 Evaluation

We evaluate SquirrelJoin’s network skew mitigation in clusters with real-world applications and synthetic background traffic. We study different workloads and the robustness of SquirrelJoin’s measurements. We then explore scalability and the available parameters.

4.5.1 Experimental set-up

We deploy Apache Flink [Fli16a] on a 17-node cluster (16 nodes with 4 workers each and 1 master node) on Google Compute Engine. We use “n1-standard-16” virtual machines (VMs) with 16 CPU cores at 2.5GHz and with 60 GB of memory. We allocate 12 GB of memory to each Flink worker. We configure the VMs (using the Linux tc tool) to have virtual 1 Gbps NICs. We store input data on a RAM disk to ensure that storage I/O is not a bottleneck.
While a storage bottleneck does not change the way SQUIRRELJOIN operates, it requires more background traffic to achieve the same straggler slow-down, and hence, the same benefit, as in the case when the network is the bottleneck. With the same amount of background traffic and a storage bottleneck, SQUIRRELJOIN’s benefit decreases, depending on the severity of the storage bottleneck.

**Join workload.** Our workload is a join of the *orders* and *lineitem* tables as found in Query 10 of the TPC-H benchmark [TPC16b]:

```
SELECT o_orderkey, o_custkey, ..., 
    SUM(l_extendedprice * (1 - l_discount)) as revenue 
FROM orders, lineitem 
WHERE l_orderkey = o_orderkey 
AND l_returnflag = 'R' 
AND YEAR(o_orderdate) > 1990 
GROUP BY o_orderkey, o_custkey, ...
```

To isolate the join time, we apply the selections and compute the *revenue*, storing the results in HDFS. We use a scale factor of 1000, giving each preprocessed table a size of roughly 160 GB. Unless otherwise noted, we use a subset of 40 GB per table of the total input data to keep all join state in memory, as our SQUIRRELJOIN implementation currently does not spill data to disk.

We compare SQUIRRELJOIN to the default hash-based repartition join implementation in Flink. Each experiment is repeated 5 times, and we report the 25\textsuperscript{th}, 50\textsuperscript{th} and 75\textsuperscript{th} percentiles as errorbars.

**Background traffic.** We use four common applications found in compute clusters for generating background traffic:

1. **Log aggregation.** Log aggregation services collect logs from cluster machines and consolidate them at one or few aggregator servers. This “many-to-few” traffic is expected to generate network skew, for example, when the cluster experiences a high volume of log data due to ongoing error reporting. We use Facebook’s Scribe [TPC16a] as a log aggregator and deploy it on the Flink worker VMs with one aggregator server.

2. **Distributed search.** Similar to log aggregators, distributed search engines also follow a many-to-few pattern as already seen in Chapter 3. Incoming search queries are partitioned and processed by a number of backend servers, and the results are collected at a frontend server. Substantial network skew can occur during periods with query bursts. We use Apache Solr [Sol16] as a distributed search system and deploy it co-located with the Flink workers.

3. **Data transfer.** Large point-to-point data transfers can occur in clusters, e.g. during VM migration or large file downloads. Such transfers can be accelerated by parallelising downloads across multiple TCP connections, which generates competing receiver-side skew. We emulate a VM migration operation by transferring an 8 GB Debian image to one of the Flink workers via 30 parallel TCP connections.
(4) Data analytics. Besides Flink, a cluster may host other data-parallel frameworks such as Hadoop, Spark or TensorFlow. If a job contains few operators with large fan-in, e.g. a reducer during a MapReduce job or a parameter server for distributed machine learning, receiver-side skew is possible. We use an interfering Terasort job [HHD+10] in MapReduce with 32 mappers and 4 reducers running on the Flink worker VMs.

Each system is configured with a workload that fully utilises the available incoming bandwidth of its receivers when run in isolation, and we introduce skew 10 s after starting the join.

4.5.2 Different network skew

To evaluate the effectiveness of SquirrelJoin’s approach for the mitigation of network skew, we run it against the standard Flink join with a variety of background traffic patterns. We start off with network skew caused by background traffic from the above described real-world applications sharing the network.

Real-world applications. Figure 4.4 shows that the regular Flink join experiences an approximate $2 \times$ slow-down in the presence of background traffic. This is because the number of competing TCP network flows that each application generates is equal to the number of TCP flows that Flink generates on each worker (15). Since TCP implements max-min fairness, the available network bandwidth for Flink is halved, leading to straggling workers. The file transfer application uses 30 TCP flows, but it finishes after the 8 GB have been transferred, thus the skew does not last for the entirety of the join. SquirrelJoin correctly detects the network skew in all cases and avoids sending data to the straggling workers. As a result, its performance is consistently close to a deployment without background traffic, and it exhibits a speed-up over regular join of up to $1.7 \times$. SquirrelJoin shows a small overhead when running without background traffic, due to its start-up phase (see §4.5.5).

As background traffic is distributed evenly across cluster nodes, individual stragglers disappear as all Flink workers are affected equally, decreasing SquirrelJoin’s benefit. In the next
experiment, we increase the number of reducers for the terasort MapReduce job until it uses the entire cluster during its shuffle phase. Figure 4.5 shows that, with more reducers, SquirrelJoin’s completion time increases until, with 32 reducers, it matches the completion time of the regular join. As more reducers are used, the time of the shuffle also decreases and background traffic lasts shorter, i.e. the join is less affected. Another observation is that the regular join experiences a higher variance for 8 reducers. This is due to the fact that, with more reducers, it is more likely that MapReduce schedules multiple reducers on a single node, thus increasing the number of competing TCP flows to that node. SquirrelJoin is resilient to such issues.

**Different network skew start times.** To evaluate SquirrelJoin’s ability to retain lazy partitions when network skew appears later during the join computation, we vary the time at which synthetic network skew occurs. We use iperf, a bandwidth measurement tool, to create 30 competing TCP flows from an external machine to one Flink worker VM. We delay the background traffic by different percentages of the join completion time (without background traffic) and then let it occur until the join has completed.

Figure 4.6 shows that, when the network skew occurs early, the regular join experiences a $2.9 \times$ slow-down with 30 competing TCP flows, causing Flink’s available bandwidth to drop to roughly $1/3$. As expected, the later the network skew occurs during the join execution, the less impact it has on the join completion time until it completes before the background traffic starts ($110\%$). We observe that SquirrelJoin’s completion time is constant at around 80s, independently of when network skew occurs, yielding a maximum improvement of $2.1 \times$. The stable completion times indicate that SquirrelJoin consumes lazy partitions slowly enough to always have data available for rebalancing. SquirrelJoin cannot completely eliminate the effect of the network skew because it must send some data records over the congested network link to estimate completion times, and it introduces a start-up overhead. This means that late-occurring network skew affects SquirrelJoin longer than the regular join, explaining the difference at $110\%$. 

![Figure 4.5: Completion times for different numbers of reducers](image1)

![Figure 4.6: Completion times for different background traffic start times](image2)
4.5 Evaluation

Different network skew finish times. Next, we investigate how SQUIRRELJOIN behaves when the network skew does not last for the whole duration of the join. This shows SQUIRRELJOIN’s robustness against short periods of network skew. We repeat the same experiment as above but now always introduce network skew when starting the join and then vary its duration, again in terms of percentage of the join completion time without background traffic.

Figure 4.7 shows that the regular join experiences a linear increase in completion time with longer periods of network skew while SQUIRRELJOIN keeps completion times constant. We conclude that lazy partitions can handle transient network skew without overreacting. If the network skew is short, the impact on the regular join is low, and the start-up overhead of SQUIRRELJOIN dominates the completion time. At around 50%, which translates to approximately 30 s in this experiment, the lines cross and the regular join becomes worse. As discussed in §2.1.3, typical network skew lasts for more than 10 s, often lasting for 100 s or more. While SQUIRRELJOIN introduces an overhead for periods shorter than 30 s, it provides a benefit for most realistic longer occurrences of network skew.

Different waves of network skew. We now explore a more dynamic environment, combining the above two scenarios. We initiate different waves of transient network skew. Each wave lasts for 15 s and generates skew at a different node. We create up to 8 waves. Waves are continuous, i.e. once the previous wave has finished, the next wave starts immediately. We start the first wave after 10 s.

Figure 4.8 shows that, with more waves, the completion times of the regular join increase linearly until 7 waves when it saturates. The reason is that the join finishes before wave 7 occurs and, hence, the completion time is not further affected. SQUIRRELJOIN is stable until 3 waves and then experiences a slight increase until 5 waves, after which it is stable again at 80 s. In summary, SQUIRRELJOIN consistently outperforms the regular join and produces stable results in a highly dynamic environment. Its robustness mechanisms and gradual partition assignment prevent wrong decisions when skew affects nodes for short time periods.
Different network skew severity. SQUIRRELJOIN should detect and mitigate network skew independently of its severity. We now vary the number of competing TCP flows generated by `iperf`. Figure 4.9 shows the completion times for both regular join and SQUIRRELJOIN. It also includes the times as predicted by our model for the base case without any reassignment (model projected) and the ideal case in which perfect knowledge about network skew is available a priori and an optimal assignment can be computed before running the join (model optima).

The regular join experiences a linear increase in completion time as we increase the number of competing TCP flows. This is expected because TCP’s max-min fairness proportionally splits the available network bandwidth between different flows and, hence, the available bandwidth for the join decreases with more background flows. SQUIRRELJOIN also experiences an increase in completion time, but it is less pronounced. The gap between the regular join and SQUIRRELJOIN increases until SQUIRRELJOIN reaches a speed-up of approximately $2.3 \times$ for 40 competing TCP flows. After that, the difference stays nearly constant. The reason is that even though SQUIRRELJOIN correctly detects and rebalances the network skew, it must send a small amount of data over the shared link to obtain reliable progress measurements. The more competing network flows use that link, the longer it takes for SQUIRRELJOIN to transmit that data. If reliable measurements can be obtained faster in a more stable environment (e.g. by using DCTCP [AGM+10]), this effect would decrease.

The time predicted by model projected closely matches the observed time of the regular join. As the optimal model assumes perfect a priori knowledge of the network skew, it does not account for any data sent over the shared link and, hence, model optima and SQUIRRELJOIN diverge. However, this does not affect our findings about straggling senders (see §4.2.4) as these assume the optimal case, which is a lower bound. The figure shows that SQUIRRELJOIN always achieves at least 70% of the maximum possible improvement for 20 or more competing network flows.
4.5 Evaluation

4.5.3 Different workloads

To analyse SquirrelJoin’s behaviour under different join workloads, we first vary the size of the probe table and then consider different queries that have more than a single join.

**Different probe table sizes.** We use the pre-processed 40 GB orders table as the build table and then vary the size of the lineitem table by increasing the subset of data from the original preprocessed table to simulate longer running joins. We use three differently sized lineitem tables, 40 GB \(Q_{40}\), 60 GB \(Q_{60}\) and 80 GB \(Q_{80}\). Network skew starts randomly at 1–40 s after the join and lasts for its remainder. We rerun each query 50 times and plot the CDF of the completion times.

Figure 4.10 shows that, for larger probe tables, the CDF for both SquirrelJoin and the regular join shifts to the right by approximately 20 s as the overall join takes longer to complete. The shape of the CDF, however, does not change. This shows that SquirrelJoin scales with the size of the probe table and can retain lazy partitions long enough to also balance late-occurring network skew. Compared to the regular join, whose performance depends on the start time of the network skew, SquirrelJoin is largely unaffected by this, only deviating 10%–15% from the median.

SquirrelJoin does experience a short tail around the 95th percentile. This is due to the fact that in some cases, network skew starts right after a consumption decision was made. This means that a certain amount of lazy partitions has been evenly distributed just before they would have been needed for rebalancing and as a result fewer are available for rebalancing. This results in an increased completion time compared to when the skew starts shortly before the consumption decision and hence more lazy partitions are available.

**Different join queries.** Besides the query with a single join used so far, we investigate queries with two and three joins, respectively. The 3-way join query is taken from TPC-H Query 3 and joins the customer, orders, and lineitem tables. The 4-way join query joins part, lineitem, orders, and partsupp as is done in TPC-H Query 9. We filter again the input tables.
Figure 4.11: Completion times for different join queries

according to their respective queries. Each query is run with and without network skew of 30 competing flows, which start 1 s after the query and last for 300 s, i.e. it affects all joins in the query.

Figure 4.11 shows that SQUIRRELJOIN scales with different numbers of joins in the query. For each individual join, lazy partitioning behaves independently and successfully detects and rebalances network skew. Hence, SQUIRRELJOIN’s completion time increases only slightly when there is background traffic, while the regular join experiences a 2.9 \times slow-down. The benefit on the 4-way join query is less because, due to its increased runtime, the background traffic stops before the query completes.

4.5.4 Robustness

Next we verify the robustness and accuracy of our measurement and assignment mechanisms in the presence of real-world confounding conditions.

Uneven data distribution. Next we study how SQUIRRELJOIN handles data skew. Since TPC-H data has evenly distributed join keys, we generate a skewed synthetic dataset according to Rödiger et al. [RMU+14]. The data models urban population and consists of a \texttt{people} and a \texttt{city} relation, which are joined on the primary key of \texttt{city} and a foreign key of \texttt{people}. The foreign key is skewed and randomly drawn from a Zipf distribution to model the fact that many people live in few cities. Each relation contains 200 million records with a resulting size of 40 GB and is randomly partitioned across all nodes such that each nodes stores a total of 5 GB. We vary the Zipf factor $s$ from 0 to 1. Larger values mean heavier skew, whereas 0 represents a uniform distribution. We distinguish three scenarios: (S1) no network skew; (S2) network skew that affects the node that is most affected by the data skew; and (S3) network skew that affects a node that is unaffected by the data skew.

Figure 4.12 shows the results comparing SQUIRRELJOIN to the regular join. We observe that data skew only affects the completion time for $s = 1$. This is consistent with Rödiger et al. and
shows that the basic hash partitioning is already sufficient to balance most data skew. When $s = 1$, the regular join exhibits a slow-down of $1.3 \times$ without network skew. With network skew, we see a $2.9 \times$ slow-down for $S3$ and a $3.8 \times$ slow-down for $S2$. In the latter case, the slow-down is worse because the straggling node receives more data compared to the others and additionally, receives at a slower rate.

SQUIRRELJOIN is also affected by heavy data skew, but its performance relative to the regular join stays the same without network skew. This is because SQUIRRELJOIN does not balance data skew but evenly distributes lazy partitions across receivers. This preserves data skew but does not make it worse. When there is network skew, SQUIRRELJOIN balances it and achieves a speed-up of up to $2.6 \times$ over the regular join. As network skew is the main bottleneck, data skew does not significantly change the results. SQUIRRELJOIN also shows a slight improvement in the median for $S2$: by assigning lazy partitions from the straggler, data skew is also partially balanced. We conclude that SQUIRRELJOIN’s balancing mechanism is not affected by data skew. In addition, as lazy partitions allow a flexible key assignment, they could be combined with more advanced approaches for data skew mitigation [VBBE12, RIKN16].

Uneven CPU load. We also investigate the behaviour of SQUIRRELJOIN when stragglers are caused by imbalances in CPU utilisation. We run the join from TPC-H Query 10 and vary the number of contended CPU cores on one node using the Linux stress tool without network skew (see Figure 4.13).

The performance of both the regular join and SQUIRRELJOIN decreases linearly with more contended CPU cores. SQUIRRELJOIN performs slightly worse due to its overhead (see §4.5.5), but the relative difference remains almost constant. CPU skew is bidirectional as opposed to the unidirectionality of network skew and affects both senders and receivers evenly. Hence it is not visible in the receiver rates, and SQUIRRELJOIN does not react to it. While this shows the robustness of SQUIRRELJOIN to CPU contention, we believe that, in practice, such
Chapter 4. Traffic Balancing for Non-Aggregatable Traffic

Figure 4.14: Estimation error in each detection iteration

situations only rarely occur. Cluster schedulers such as Yarn or Mesos already isolate CPU resources to prevent CPU imbalances.

**Estimation accuracy.** Finally we assess the accuracy of SQUIRRELJOIN’s completion time estimates. We execute the join without network skew and record the estimates every second, comparing them to the actual time that it took to complete the join. Completion times are estimated for each parallel receiver running within a worker (4 per node, 64 in total for our experiments), and we compute the error for each of them and plot the minimum and maximum errors in addition to the 25th, 50th and 75th percentiles. Negative errors mean that the completion time was overestimated and vice versa.

As can be seen in Figure 4.14, initially the errors are high (we exclude the first two data points, which are above 100%). This is due to the start-up effects when senders are not yet sending at their full rates. It is important to note that the variance of the error is low, i.e. even though the completion time is underestimated, it is done equally for all receivers and, hence no skew is detected. After 6 iterations, the error is within a 10% bound. We can observe another error increase at around iteration 10. This is caused by the change between the two phases of the two-phase hash join. After the build phase has finished, data from the probe table is sent, which results in a short interruption between the phases and increases the errors. After that, errors are consistently between 0 and -10%. To avoid wrong decisions during the phase change, SQUIRRELJOIN does not perform rebalancing after the first sender and before the last sender has finished the build phase. The results show that SQUIRRELJOIN accurately estimates completion times and, combined with the robustness mechanisms from §4.4.2, prevents making wrong reassignment decisions.

4.5.5 Scalability and overhead

We analyse SQUIRRELJOIN’s scalability and overhead with different cluster and input sizes.

**Cluster size.** We deploy Flink with three cluster sizes (16, 24 and 32 nodes), comparing SQUIRRELJOIN and the regular join with and without network skew. Together with the
cluster size, we scale the input data to keep the same data-to-node ratio for all deployments. We also increase the number of competing background flows so that the straggling worker always has only 1/3 of its network bandwidth available for the join processing.

Figure 4.15 shows that, with 16 and 24 nodes, the completion times are almost equal for the four evaluated scenarios. With 32 nodes, we observe a slow-down caused by straggling workers due to other reasons than network skew. Even in this case, the network skew mitigation of SQUIRRELJOIN remains effective, and SQUIRRELJOIN with network skew performs within 3% of SQUIRRELJOIN without network skew.

Input data size. We compare the regular join and SQUIRRELJOIN without network skew as input tables grow. We use the same pre-processed tables as before and increase the subset of data from 20 GB to 80 GB per table until the join would start spilling. Figure 4.16 shows that the overhead of SQUIRRELJOIN remains fixed for 40 and 80 GB of input data at around 10 s. This fixed overhead is due to the start-up phase of SQUIRRELJOIN in which the network is not utilised until the first assignment decisions has been made. With 120 GB of input data, the difference decreases until both joins exhibit the same behaviour for 160 GB of input data. For larger partitions of the input tables, more matches are found, and the join computation shifts from being network-bound to becoming CPU-bound. Hence, SQUIRRELJOIN’s overhead is not affecting the completion time anymore.

We conclude that lazy partitioning is feasible for large joins, which are more prone to suffer from network skew due to their longer runtime, because it only adds a fixed overhead at the start. While the overhead for smaller data sizes is significant, we believe that implementations with a more optimised read path, e.g. in-memory DBMSs [RMKN15], can fill lazy partitions faster and thus reduce the overhead. If the join is CPU-bound, SQUIRRELJOIN’s overhead decreases but so does its benefit as the join is less affected by network skew.

Algorithmic overhead. SQUIRRELJOIN adds overhead on the critical path by executing the skew detection and reassignment algorithms on the LPC (see §4.4.1). Other overheads such as simple computations and data structure lookups require constant time, but the execution time
Chapter 4. Traffic Balancing for Non-Aggregatable Traffic

Figure 4.17: Completion times of the assignment algorithm for increasing cluster sizes

of the reassignment algorithm grows with the cluster size. We therefore want to investigate if the execution of the reassignment algorithm can become a bottleneck. To explore this issue, we use a micro-benchmark in which we run the detection and reassignment algorithms in isolation for different cluster sizes and measure the time taken to finish the computation. The complexity of the detection algorithm is $O(W)$ because it must scan all workers to find the slowest one. The reassignment heuristic has a worst case complexity of $O(P_R^B)$, which occurs when all lazy partitions are assigned during one step.

Figure 4.17 shows the computation time as we vary the cluster size from 64 to 16,384 workers. The results show that the algorithm is unlikely to become a bottleneck: in practice, the computation time increases super-linearly with the number of workers. This is due to the fact that, in our prototype implementation, all nodes are sorted according to their estimated completion times before the algorithm runs. However, the computation times, even for large clusters, are still only in the range of tens of milliseconds. This result demonstrates that we can efficiently compute assignments and that the computation does not limit the scalability of SQUIRRELJOIN. We further note that the reassignment algorithm runs periodically, in parallel with the join. Thus the system does not wait on the result of the reassignment at any point in time. At worst, the runtime of the algorithm increases the reaction latency. As we show in §4.5.6, an increase of tens of milliseconds does not affect SQUIRRELJOIN’s effectiveness.

4.5.6 Parameter choices

Next we perform a sensitivity analysis on all parameters used by SQUIRRELJOIN in order to establish the general applicability of our default values.

**Skew threshold.** We evaluate the effect of $\tau_{skew}$ on the join completion time by varying it from 1% to 75%. A low percentage means a high sensitivity to network skew while a high percentage represents a low sensitivity. We also vary the intensity of background traffic from zero competing flows, i.e. no network skew, to 30 competing flows. Background traffic starts 1 s after the join and lasts for the whole job duration.
4.5 Evaluation

Figure 4.18: Completion times for different skew thresholds $\tau_{\text{skew}}$

Figure 4.19: Completion times for different consumption thresholds $\tau_{\text{consume}}$

Figure 4.18 shows that values of $\tau_{\text{skew}}$ around 5% provide a good sensitivity trade-off for all intensities of background traffic. This value is not brittle in that all values between 5 and 50% perform similarly. Staying on the lower end of that range allows us to maintain more partitions to adapt to future network skew changes. SquirrelJoin is too sensitive for lower skew percentages, which can lead to wrong decisions: completion times increase for 20 competing TCP flows for $\tau_{\text{skew}}$ values of 1% and 2.5% due to falsely reacting to noisy input data. The adverse effect happens at the opposite end of the spectrum when $\tau_{\text{skew}}$ is too large: network skew is regarded as noise and no adjustments are made.

**Consumption threshold.** Next we evaluate the performance of SquirrelJoin for different values of $\tau_{\text{consume}}$. Lower percentages mean that fewer lazy partitions are consumed when a worker fills up its memory; higher percentages cause more to be consumed. We also vary the start time of the background traffic from 1 s to 30 s after the start of the join.

Figure 4.19 shows that SquirrelJoin achieves the best performance for $\tau_{\text{consume}}$ values between 1% and 5%. Higher percentages cause a slow-down of up to $1.9 \times$ due to the earlier consumption of lazy partitions, which leaves fewer for later rebalancing. This even affects SquirrelJoin when the network skew begins shortly after the start of the join because an initial consumption is required to start sending out data and collect progress measurements.

**Assignment interval.** We vary the assignment interval, $\tau_{\text{assign}}$, from 2.5 s to 15 s with and without network skew. Shorter intervals result in more frequent skew detection cycles; longer ones react more slowly but are more robust as more measurement data can be collected.

Figure 4.20 shows that the low values of 2.5 s and 5 s provide better performance with network skew as they react more quickly. This shorter reaction period permits the assignment of more lazy partitions based on rate estimates to non-straggling receivers. When the assignment interval grows longer, SquirrelJoin reacts more slowly. Hence, a larger number of lazy partitions have already been assigned evenly across all receivers as the worker memory fills up more often before the first assignment decision. This explains the increase in completion time for larger values. The assignment interval does not affect performance without background
traffic because SQUIRRELJOIN does not have to make any assignment decisions. In our experiments, we set the assignment interval to 1 s.

We observe that even the shortest intervals are not susceptible to noise in this set-up using constant, steady background traffic. This shows our noise mitigation techniques, i.e. the usage of a sliding window, the Friedman Test, and the absolute threshold (see §4.4.2) improve the robustness of SQUIRRELJOIN. Setting $\tau_{assign}$ to 5 s increases the number of measurements for the $\chi^2$ approximation in the Friedman test, making SQUIRRELJOIN more robust to transient network skew.

**Lazy partition size.** As the overhead depends on when the first consumption decision is made, we evaluate the impact of the amount of memory that is allocated for storing lazy partitions. We execute the join with and without network skew introduced by 30 competing TCP flows and vary the allocated memory from 500MB to 1.75GB.

As shown in Figure 4.21, without network skew, less allocated memory helps to reduce the start-up overhead because consumption cycles happen more frequently. A memory size of 500 MB reduces the overhead by around 4 s, which translates to 50% less overhead compared to larger sizes above 1.5 GB. On the other hand, shorter consumption cycles limit the amount of network skew that can be rebalanced: until SQUIRRELJOIN has enough information to detect network skew and make the first assignment decision, more lazy partitions are consumed as memory fills up quicker. Considering again a size of 500 MB, the benefit of rebalancing decreases by 98%. As a result, larger amounts of memory are preferable until the overhead without network skew outweighs the benefits with skew. This occurs at around 1.75 GB when the overhead is still reasonably small.

### 4.6 Summary

We presented lazy partitioning, a new technique for distributed joins that dynamically adapts to network skew in shared compute clusters. Lazy partitioning delays the assignment of data
partitions to join workers in order to permit future assignment decisions that offset network skew. By delaying partition assignment, our approach avoids both the need for a priori knowledge of the network skew and expensive state migration to rectify bad past assignment decisions. We use an analytical model to show that a lazy partitioning approach has the potential to be highly effective at addressing receiver side network skew.

We implemented lazy partitioning in SQUIRRELJOIN for Apache Flink and showed its effectiveness in reacting to network skew. SQUIRRELJOIN measures individual progress rates of workers and makes assignment decisions based on robust completion time estimates, thus maximising network utilisation and avoiding congested network paths. Its architecture minimises the overhead of lazy partitioning on the critical path by reducing the amount of required synchronisation and moving the control over lazy partitions to a centralised master, co-located with the Flink master. SQUIRRELJOIN therefore significantly improves join completion times in shared compute clusters under background network traffic with minimal overhead when no skew exists.

SQUIRRELJOIN is network-aware, as it anticipates effects from network interference that can occur in shared data centres. It actively measures worker throughput to monitor the network and exploits the unidirectionality property of network skew to effectively balance skewed network accesses. Additionally, SQUIRRELJOIN does not require state migration when repartitioning data, avoiding additional network traffic.
We now present SwiftAnalytics, an optimised object store architecture, which allows to run efficient analytics jobs. Based on a detailed benchmarking study, SwiftAnalytics overcomes the two major performance problems when executing data-parallel frameworks directly on top of an object store: read locality and write locality. SwiftAnalytics eliminates unnecessary I/O caused by these two problems by explicitly exposing the concept of locality to the decentralised object store without affecting its scalability.

5.1 Overview

SwiftAnalytics is an enhanced object-store, which allows to seamlessly integrate analytics frameworks and object stores without significant performance penalties. Its design is based on a detailed performance study (see §5.2), which revealed two major findings impeding the performance of analytics jobs:

1. Write locality: Object stores such as OpenStack Swift use consistent hashing to place and locate objects. Due to the fixed mapping between the object name and its host, the object store does not have flexibility in choosing the destination of an object. This leads to frequent remote writes and makes renaming an object an expensive operation as it requires copying of the data. During an analytics job, objects are renamed several
times (due to techniques such as speculative execution, see also §2.3.1), which adds additional latency when performed using an object store.

2. **Read locality**: As object stores store data at the granularity of objects and not blocks, large objects are concentrated on a small subset of storage nodes instead of being spread equally across the cluster. This coarser granularity limits aggregate bandwidth during reads and hence, slows down data ingress for a data-parallel query.

To address the write-locality problem, **SwiftAnalytics** uses *locality-aware writes*, which offer placement control to the object store and allow the implementation of an efficient renaming strategy. Locality-aware writes break the dependency between the object name and its hash-defined location without introducing a centralised component and hence do not affect scalability. **SwiftAnalytics** achieves this by introducing a *second lexicographic namespace* which, besides the usual hash-based placement, allows to explicitly specify the target host. The location information is then stored in a decentralised way such that it can be located efficiently with only a small overhead (on the order of milliseconds).

**SwiftAnalytics** solves read-locality with a straight-forward object chunking scheme which, similar to HDFS, chunks objects into smaller sized parts to achieve a better distribution across the cluster. In contrast to existing solutions that optimise object stores for analytics [Rot16] [VBBE12], **SwiftAnalytics** tackles the problems by changing the store itself rather than the connection layer between the storage and the data-parallel framework. This allows it to provide an effective solution without affecting scalability and usability.

### 5.1.1 Design challenges

**SwiftAnalytics** faces three design challenges, which we overcome in this chapter:

**Decentralised placement control.** The most important property of **SwiftAnalytics** is to provide placement control without breaking decentralisation and limiting scalability. Placement control is required to avoid (remote) data copies when new objects are uploaded or existing objects are renamed. To avoid introducing a centralised component that stores object locations, **SwiftAnalytics** stores the location information in a decentralised fashion with the meta-data of an object.

**Low overhead on general operations.** The added placement control should not have a significant impact on the basic operation of the object store. To achieve this, **SwiftAnalytics** only provides placement control for the *primary* replica of an object while the remaining replicas are still copied to their original destinations. As a result, an object can be located without additional effort in \((N - 1)\) cases where \(N\) is the number of replicas.

**Transparency.** Users should not have to deal with placement control and access objects unaware of any changes to the location of an object. **SwiftAnalytics** does not change the
interface that a traditional object store exposes and does not alter object names to influence placement. Hence, SWIFTANALYTICS is fully backwards compatible and does not affect existing workloads.

5.1.2 Design space

SWIFTANALYTICS is related to two main efforts: improving the performance of analytics on object stores and providing placement control in a consistent-hashing based, decentralised system. In the former category, while we are not aware of any work in the literature that deals with the discussed problems, the community around Hadoop and Spark has tried to solve the performance problems by simply avoiding renames during job completion.

The closest related effort is the Stocator project [Ver16], an optimised connection layer between data-parallel frameworks and object stores. Stocator supports writing results directly to an object store and prevents potential data loss, which may occur when speculative execution is used, by appending a unique suffix to the output file name of each single output task. Hence, competing tasks write to different objects and cannot overwrite each others changes. This makes the 2-phase commit protocol unnecessary and avoids renames. However, it changes the output file name, and thus is not a transparent change to the user as it now becomes more difficult to retrieve a result after a job has completed. While Stocator changes the connecting layer, SWIFTANALYTICS changes the underlying object store, which allows to implement a faster renaming operation using placement control instead of avoiding it. This makes SWIFTANALYTICS fully compatible with data-parallel frameworks and their optimisations.

Other work has looked into providing placement control in scalable, decentralised storage systems [HJS+03] [ZGS03] [WBMM06]. SkipNet introduced the idea of having two namespaces to place and locate objects, which inspired SWIFTANALYTICS. It uses the two namespaces to control placement in a distributed hash table and thereby enables content locality. However, the scope of both approaches differ as SkipNet targets distributed hash tables and does not use the two-namespace approach to provide fast renames.

5.1.3 Contributions

In this chapter, we make the following contributions:

1. We perform an in-depth analysis of the performance of analytics jobs running directly on top of object stores. We use a stack of Hadoop MapReduce as the data-parallel framework and OpenStack Swift as the object store. We provide a detailed break-down of the performance of a carefully chosen set of workloads with different properties and quantify the different causes of performance loss (§5.2).
2. We propose SwiftAnalytics, an optimised object storage architecture to solve the two most critical performance issues: write locality and read locality. SwiftAnalytics introduces decentralised placement control to solve write locality. Additionally, it demonstrates that object chunking, a mechanism already available in object stores such as OpenStack Swift, is necessary to match the performance of distributed filesystems such as HDFS (§5.3).

3. We introduce locality-aware writes as a mechanism to enable decentralised placement control in object stores. Locality-aware writes are based on a second object namespace, which allows to specify the storage node for an object. We use this approach to implement a fast renaming function for object stores to improve job performance (§5.4).

To evaluate the SwiftAnalytics architecture, we implement locality-aware writes in OpenStack Swift and deploy it as the storage layer of two data-parallel frameworks, Hadoop MapReduce and Spark, on our local research testbed. We assess its effectiveness using a set of workloads with different characteristics (§5.5). With locality-aware writes, we show that result data can be written up to $8.5 \times$ faster, translating to an overall improvement of job completion time by $5.1 \times$ compared to an optimised Swift deployment; object-chunking can provide an up to $3.4 \times$ improvement during the reading of input data.

5.2 Performance issues for analytics on object stores

We start by presenting experimental evidence for the MapReduce performance problems when executed directly on top of Swift.

5.2.1 Experimental set-up and workloads

We deploy Swift version 2.2.1 and Hadoop version 2.6.0 on a dedicated cluster with 16 nodes. Each node has an Intel Xeon E3-1220 CPU with 4 cores at 3.1 GHz, with 16 GB of memory, and a 500 GB hard disk, interconnected via 1 Gbps Ethernet. Each node runs a Hadoop DataNode daemon, a Hadoop NodeManager daemon, and a Swift object server and container server daemon. We provision an additional node to run Hadoop’s master services. The replication factor is set to a typical value of 3. To allow MapReduce to use Swift as its storage layer, we use a recently-developed connector [Had16b]. We deploy Swift and HDFS on XFS.

We use four different workloads, as summarised in Table 5.1. The workloads offer four possible combinations of CPU and write I/O load. By I/O, we mean only the interaction of MapReduce with the distributed storage, not including any local I/O that occurs during job processing. Write I/O intensive means that there is a large amount of result data written to the distributed storage and the ratio between output and input data is high; CPU intensive
5.2 Performance issues for analytics on object stores

<table>
<thead>
<tr>
<th></th>
<th>Wordcount</th>
<th>PCA</th>
<th>Terasort</th>
<th>ETL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data read</td>
<td>100 GB</td>
<td>32 GB</td>
<td>32 GB</td>
<td>60 GB</td>
</tr>
<tr>
<td>Data written</td>
<td>2.1 GB</td>
<td>1.3 MB</td>
<td>32 GB</td>
<td>7.9 GB</td>
</tr>
<tr>
<td>Map tasks</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Reduce tasks</td>
<td>64</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Map waves</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPU intensive</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Write I/O intensive</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: Workload properties

means that the job is computationally expensive. We have carefully chosen the workloads to cover this range of CPU and I/O behaviours because they represent the main classes of different analytics workloads, including MapReduce and also advanced analytics such as SQL processing, graph processing, and machine learning.

The first two workloads, Wordcount and Terasort, are standard MapReduce benchmarks taken from the HiBench suite [HHD+10], and include input data; the third workload, principal component analysis (PCA) is executed on a large collection of images. We use the PCA implementation from the HIPI project [HIP17] and a 32 GB subset of the 2010 ImageNet dataset [DDS+09]; as a fourth workload, we use an ETL transformation from the TPC-DI benchmark [PRJC14]. We implement a variation of the CustomerDim transformation that scans a large XML input file for New actions and extracts the required fields in CSV format for bulk loading into a data warehouse. We tune MapReduce for each individual job to maximise throughput. Unless stated otherwise, we repeat each experiment 3 times and report averages, with the minimum and maximum as errorbars. We clear file system caches between runs.

5.2.2 Overall overhead

Figure 5.1 shows the job completion times for all four workloads on HDFS, Swift in its standard deployment with a single proxy, and Swift with multiple local proxies, respectively.
For **PCA**, all three set-ups perform equally, while for the other three workloads, **HDFS** is always faster than **Swift**, and **Swift** with local proxies always outperforms **Swift** with a single proxy.

While the connector allows to read data directly from the object servers, writes still go through the **Swift** proxy, and the write I/O intensive jobs experience a major decrease in performance due to the proxy’s shared network link. In the worst case of **Terasort** where the input data size equals the output data size, tasks time out, and the job is aborted (indicated by an X).

As mentioned in §2.3.2, the proxy server is stateless and can thus be replicated. We deploy an additional proxy server on each node, and let clients by default talk to their local instance. This helps improve performance, in particular, for the write I/O intensive jobs (see Figure 5.1), but we can still observe a major discrepancy of up to 6.5× between **HDFS** and **Swift**. To better understand where the overhead originates from, we break down the job completion time into map and reduce phases. In the following, all results for **Swift** use local proxy servers.

### 5.2.3 Map phase overhead

We start by analysing the map phase. Figure 5.2 shows that, for the CPU intensive jobs, the map phase does not exhibit reduced performance on **Swift** compared to **HDFS**—for the **ETL** workload, there is even a slight improvement. This is because read I/O is not the bottleneck in those cases. On the other hand, for the non-CPU intensive jobs, there is an approximate slow-down of 1.5×.

In **Swift**, objects are not chunked when uploaded, which is in contrast to **HDFS** that splits files into smaller blocks. Additionally, an object’s location is determined through consistent hashing on the ring, i.e. its location is fixed while **HDFS** can dynamically select the storage device for a block as the mapping is stored centrally on the NameNode. This provides a more fine-grained way for **HDFS** to distribute blocks to achieve an even read I/O load across nodes.

However, in practice, **HDFS** and **Swift** exhibit a similar amount of read skew (see Figure 5.3). The reason is that our set-up consists of 16 storage nodes, and the number of input objects...
5.2 Performance issues for analytics on object stores

Figure 5.4: Sum of time spent in calls to Hadoop’s FileSystem API

Figure 5.5: Sum of time spent in HTTP requests to Swift API (Wordcount)

for a job is 64 for both Terasort and Wordcount. With 3-way replication, each object is stored on 3 different nodes because Swift avoids co-locating objects for fault tolerance. That means that the object-to-node ratio is high and hence objects are evenly distributed across all nodes.

As we will show later, with a low object-to-node ratio, Swift experiences a slow-down of up to $4.6 \times$ during reads (see §5.5), i.e. the coarse granularity of objects compared to blocks can significantly slow down jobs. However, it does not explain the slow-down in Figure 5.2 as both systems show similar skewed accesses. We currently do not know the exact origin of the observed slow-down but assume it is related to the filesystem. During our experiments, we noticed that changing the filesystem to ext4 can improve Swift’s read throughput to match HDFS’s performance during the map phase, however, it reduces its write throughput during the reduce phase.

5.2.4 Reduce phase overhead

Figure 5.2 shows the times each job spends in the reduce phase. For HDFS, these times are short, whereas, for Swift, we observe a slow-down of up to $15.8 \times$ for the ETL workload. The major work done by the reduce tasks is to read the map task outputs, sort/merge them based on their keys, and write them back to HDFS. Hence, the time is longest for jobs with large amounts of result data.

To locate the overhead, we instrument all calls that MapReduce makes to the distributed filesystem and measure their execution time for each reducer and the application master, i.e. the node that coordinates the job. Figure 5.4 reports the mean and the 5th and 95th percentiles across all tasks and runs of the experiment. While calls to HDFS are barely noticeable, calls to Swift accumulate to tens of seconds for reduce tasks and to hundreds of seconds for the application master.

When a MapReduce client on Swift makes a call such as delete or rename to Hadoop’s FileSystem interface, the connector translates it to the appropriate HTTP request and sends it to the Swift proxy. To identify the calls that cause the delays in the filesystem accesses,
we further separate them into the HTTP requests sent to Swift and sum up the times each individual reduce task and the application master spend in a call, as shown in Figure 5.5. We again report the mean and the 5\textsuperscript{th} and 95\textsuperscript{th} percentiles.

A striking observation is that the application master spends a large amount of time just in COPY calls. As explained in §2.3.1, the application master renames all output files to the user-specified folder when the job has finished. While a rename in HDFS is just a metadata update on the NameNode, Swift has to re-upload the object under its new name and delete the copy with the old name. The slow-down is amplified by the fact that an object is also replicated when uploaded, and Swift waits for at least 2 copies to be written before returning from the call. While the amount of output data in \texttt{Wordcount} is small, the added delay is still significant and becomes even worse for the write I/O intensive workloads.

We observe the same behaviour for reduce tasks when the two-phase commit is performed. First, the object is uploaded via a PUT and, if allowed to commit, renamed via a COPY. Combined with the application master, this results in 3 uploads and replications of the same object. The impact on a single reduce task is not as high because it only has to rename one object, and performs this in parallel to the other reduce tasks. However, with more output data, the effect intensifies: in \texttt{Terasort}, a reduce task spends over 60s in COPY.

Another interesting observation is the time spent in HEAD requests. For the application master, single HEAD requests usually take a few tens of milliseconds, but it needs to perform a large number of them during the job finishing stage. For \texttt{Wordcount}, for example, an application master makes over 600 HEAD calls. These are necessary to simulate the directory hierarchy that MapReduce uses to store temporary results on HDFS in the flat namespace of the object store. Unlike the application master, reduce tasks perform only few HEAD requests. However, the first includes an authentication request to Swift, which is slow (see AUTH in Figure 5.5). While the authentication problem adds a fixed overhead, the amount of HEAD requests scales with the size of the cluster and the number of outputs written, making it worse for larger jobs.

### 5.2.5 Discussion

From our analysis, we identify two classes of problems, external and internal. External problems are related to Swift’s API. Examples are the overhead of the authentication or the large amount of HEAD calls. While these are important problems, we believe that the system can be optimised by, for example, reducing the number of HTTP calls as is done by Stocator [Ver16] or storing a small amount of transient metadata in the connector during an analytics job.

Internal problems are due to the way Swift (and other object stores such as Dynamo [DHJ+07], or Walnut [CDM+12]) manage objects: (i) as described in §2.3.2, Swift uses hash-based placement of objects. To place an object, a hash over the object’s identifier (its name) is
computed and this hash value determines the target storage node. The hash computation can be done by a client independently without the need of a centralised mapping between objects and storage nodes and hence enables an algorithmic lookup. While this decentralised design provides superior scalability over centralised approaches, it creates a fixed mapping between object names and storage nodes. This fixed mapping does not allow for write-locality and incurs the observed COPY calls during renames as renaming an object most likely also changes its storage node and hence, requires a (remote) copy of the object. This causes slow-downs of up to 15.8× in our evaluated workloads; (ii) the coarse storage granularity, i.e. storing data at the granularity of objects rather than blocks, reduces read-locality as a single object is distributed across fewer nodes. This leads to overheads of up to 4.6×.

Internal problems are more fundamentally related to the architecture of an object store and hence our focus in this chapter.

5.3 **SwiftAnalytics design**

The design of **SwiftAnalytics** aims to tackle the internal problems and incorporates two major extensions to the standard functionality of an object store: (i) it supports transparent object chunking to increase parallelism during reads; and (ii) it enables efficient object renames by providing placement control to clients, which allows to write objects to specific locations in the cluster.

5.3.1 **Object chunking**

A chunking mechanism allows clients to transparently split objects into smaller parts, similar to HDFS blocks, when uploading data to the object store. However, contrary to HDFS, there is no central instance to keep a block map for an uploaded object that stores the parts and their locations for a single object. Hence, a decentralised mechanism is needed to collect and assemble all chunks of an object. Swift already offers a way to achieve this. To support arbitrary object sizes, objects larger than 5 GB are split, and the individual chunks are linked to from a manifest file, stored under the original object name. While this is typically used to manage large objects in Swift, it can also be used to transparently support small, equal-sized chunks for faster analytics.

Figure 5.6 illustrates this process. On an upload, a client can specify to chunk the object, which will cause the proxy server to split the incoming data into several parts, e.g. 128 MB blocks as in HDFS. The different parts receive internal names according to which they are placed across the cluster. Additionally, a manifest object under the original name is created, which contains an ordered list of the names of all object parts. Upon a GET request, the proxy server retrieves the manifest file, parses the list, and returns the parts in order to the client.
5.3.2 Placement control

Besides chunking, SWIFTANALYTICS provides object placement control to clients for specifying the object server on which an object should be stored (see Figure 5.6). This allows to implement an efficient rename operation. The main reason why renaming is slow on object stores is that the object location depends on the name of the object. Hence, if an object receives a new name, its location changes and in most cases that means that the data has to be copied across the network to a new object server. With placement control, the object server for an object can be specified explicitly and thereby avoid the additional copy.

The main challenge for enabling placement control in an object store is to not introduce any centralised component that would limit scalability. Distributed filesystems such as HDFS have a central metadata service which stores a mapping between all file names in the system and their corresponding locations. This allows for fast renames as a rename only means updating the name of the target file in that mapping. As a main advantage of object stores is their scalability due to their decentralised architecture, adding such a centralised service is undesirable. In the following section, we present locality-aware writes, a mechanism implemented in SWIFTANALYTICS to enable decentralised placement control.

A simple alternative to placement control for enabling fast renames is a symbolic link-like solution, which we call link files. The idea is the same as with symbolic links where a file does not contain the actual contents but rather a pointer to the original file. In Swift, link files can be realised by using 0-byte files, each with an additional metadata attribute as the pointer. When an existing object should be renamed, a new link file with a pointer to the original object is uploaded and replicated leaving the actual data untouched. When the object is requested, the object store follows the link to return the original content. We will compare SWIFTANALYTICS to link files in §5.5.
5.4 Locality-aware writes

Next we present locality-aware writes and how they can be used to provide fast renames in object stores. We then discuss the implications for object stores and describe the implementation in Swift.

5.4.1 Locality using two namespaces

Locality-aware writes use a two-namespace approach to allow clients to specify the target location of an object. This is similar to the idea used in SkipNet [HJS+03]. However, we make significant adjustments to suit the needs of analytics on object storage. In its basic operation, a consistent-hashing based object store uses the numerical hash value of an object to determine its location, i.e. the hash IDs form a numeric namespace. This namespace is agnostic to any locality constraints in the cluster as it is only a logical overlay. In order to support locality awareness, locality-aware writes use a second lexicographic namespace that explicitly captures the locations i.e. the URIs of devices in the cluster. In the example in Figure 5.7, each object server has a logical ID (a number) that determines its place on the ring but also exposes its physical ID (its hostname) that represents its location in the cluster.

On a PUT, a client can now set an additional request attribute, the location ID (LID), to specify on which node the object should be placed. If set, the LID outweighs the numeric ID and bypasses the hash placement. Figure 5.7 shows the two possible ways of uploading an object. To keep the object store decentralised, the LID is stored as a metadata attribute with the object itself and each of its replicas. Hence, no centralised service to store LIDs is required.

Since locality-aware writes break the hash placement, object retrieval can fail if an object has been stored using an LID. To be able to still locate an object, three techniques are used:
Chapter 5. Traffic Elimination for Distributed Storage I/O

1. The LID is only used to determine the location of the primary replica. All additional replicas are placed according to the hash (see Figure 5.7). By accessing one of these additional replicas and reading the LID from their metadata, the primary replica can be located.

2. To avoid losing the object in case all additional replicas fail, the LID is added to the metadata of the object’s container. A container keeps a listing of all its objects plus some additional metadata per object, e.g. a checksum or the object version. Containers are replicated the same way as objects so the LID is replicated another $N$ times.

3. On object deletion, a DELETE is sent out to all replicas simultaneously and if a replica detects that an LID has been set on its metadata, it can forward the request to the corresponding location at which the primary replica is stored.

These techniques allow the object store to offer placement control while still keeping its scalability properties.

5.4.2 Local rename and upload operations

Using locality-aware writes, SWIFT Analytics implements an efficient local rename strategy which works as follows: Swift internally uses the hash of an object’s name to not only locate the responsible object server but also to find the object locally on disk. It achieves this by using the hash as the folder name in which the actual object is stored. In case of a rename operation, the object server will locally move the source file to its destination folder, given by the new hash value, where Swift will be able to find it. The local move uses the Unix \texttt{mv} command and is entirely a metadata operation if the destination is on the same filesystem as the source.

SWIFT Analytics also supports fast uploads using the above local move approach. If a client runs co-located with an object server, as is the case for our co-located analytics cluster, and has a locally stored object, it can simply move this object to the correct folder in Swift instead of copying it to a remote destination. As no I/O is generated during such a local upload, the upload is efficient. This is useful in case the object store does not support streamed uploads, i.e. the object can only be uploaded as a whole. Without streaming, output tasks first need to write their job results to local disk and then copy it to the object store, generating additional I/O. Local uploads avoid this problem and hence can speed up analytics jobs. While Swift does support streamed object uploads, other object stores such as S3 do not natively provide that functionality.
5.4.3 Implications on object stores

Locality-aware writes come with four key implications for the functionality of object stores regarding object replication, object retrieval, consistency and security.

**Replication.** While the local move saves one copy of the existing data, all additional $N - 1$ replicas still need to be copied to their new destinations. Swift’s reliability model requires a quorum of objects persisted on disk before it reports success. Hence, in a 3-way replicated case, at least one additional copy has to be written successfully before the rename can return. However, we found that in the case of an analytics job, these additional copies are wasteful. Renames of a single object are executed shortly after each other, generating unnecessary disk and network traffic, which hurts performance. Assuming 3-way replication, we identify three possible replication schemes that trade off reliability for performance:

1. Only the primary replica will be moved locally before the rename returns (1 replica).
2. The primary replica will be moved locally and an additional replica will be copied to its new destination. This scheme fulfils Swift’s reliability requirements as on return, 2/3 of the replicas have been successfully written (2 replicas).
3. The primary replica will be moved locally and two additional replicas will be copied to its new destination. This scheme provides additional reliability (3 replicas).

Eventually, all three schemes will have all replicas persisted on disk as Swift uses an asynchronous replicator process on each node to correctly handle replication failures. This process periodically scans the filesystem and makes sure objects are correctly replicated. This happens within a specified **consistency window**, which is 30s by default.

While one replica provides the best rename performance as it does not perform any I/O before returning, it also poses the biggest risk of losing data as only one copy exists for the duration of the consistency window. We evaluate this trade-off in the following section.

**Object retrieval.** When using locality-aware writes, GET requests for the primary replica of an object can now fail, and an additional request is needed to look up the LID. The number of additional requests depends on how the object store distributes load across replicas. In Swift, the proxy server randomly selects one of the replicas to be retrieved, i.e. on average 1/3 of initial requests will fail. In case custom load balancing schemes are deployed that prioritise replicas over others, it is possible for more requests to fail. We show in §5.5 that the overhead of this additional request is small compared to the reduced completion time for analytics jobs.

**Consistency.** With placement control, consistency issues can arise in case the node storing the primary replica of a locality-aware object fails. If the object is renamed, overwritten, or deleted during the downtime of that node, the previous LID may be overwritten/removed and the primary replica becomes an orphan. In that case, additional metadata is required to
indicate that a copy of the old object still exists and where the copy is stored. This metadata can again be stored in the object’s container and picked up by an asynchronous clean-up process to ensure that the object is correctly moved/deleted.

**Security.** Another concern is the possible load-balancing issues that arise when placement control is enabled. Consistent hashing not only allows for constant time decentralised object lookups, but it also achieves an even distribution of data across all devices in the cluster. Since clients can now control where objects should be stored, skew might be generated and malicious clients can intentionally flood certain devices with objects. Hence placement control should only be made accessible to authorised clients.

### 5.4.4 Implementation in Swift

Enabling locality-aware writes in Swift only requires minor code changes. We extend the object server to provide a `MOVE` call, which can be directly accessed from a client. Reduce tasks can now talk directly to their local object server and perform the rename. `MOVE` replicates the renamed object. During a `MOVE`, the object server also updates the container servers with the new name. As Swift is fully decentralised and has a simple RESTful API, an object server on any cluster node can identify the correct container servers and perform the necessary API calls to update the object name.

In order to handle `GETs` and `HEADs` correctly, we add an additional *locality-aware* layer in front of each proxy server, which intercepts requests and forwards them to the proxies. In case of a lookup failure, the locality-aware layer attempts to find an LID and, if successful, contacts the correct object server directly. To support delete operations, we extend the object server’s `DELETE` call to forward the delete request to the primary replica’s location if an LID has been set on the object.

### 5.5 Evaluation

We now evaluate SWIFTANALYTICS and compare its performance to vanilla Swift, Swift with link files, and HDFS. We use the same experimental set-up and workloads for MapReduce as described in §5.2. To evaluate our method for a larger set of workloads, we use the SWIM workload injector tool [SWI17]. We replay a set of 500 MapReduce jobs randomly sampled from Facebook traces that are part of the SWIM tool. Additionally, we deploy Spark as another example for a data-parallel framework on our 16-node cluster.

We study five aspects of SWIFTANALYTICS: the effectiveness of its local renaming approach, the effectiveness of object chunking, the different schemes for handling replication using
locality-aware writes, the effectiveness of local uploads, and the overhead of the additional
lookups during GET requests.

### 5.5.1 Effectiveness of local renames

We start by analysing the effectiveness of local renames. Note that, for this experiment, we
use the one replica strategy, i.e. do not actively replicate after the local move. We discuss
this choice and the details of replication later in this section. While the performance of the
workloads on vanilla Swift represents the baseline we aim to improve upon, their performance
on HDFS represents a near ideal level that we want to achieve with SwiftAnalytics.

**Different workloads**

**MapReduce workloads.** For the four base workloads, SwiftAnalytics offers the most
benefit for the write I/O intensive jobs ETL and Terasort (see Figure 5.8). It matches the
performance of HDFS and considerably outperforms the baseline Swift. While Swift with
link files also matches HDFS’s performance in case of the ETL job, Terasort incurs the
heaviest I/O load during renames in which case SwiftAnalytics is able to outperform link
files by a factor of 2.4× due to its local renames. In case of Wordcount, we see that, again,
SwiftAnalytics and link files perform equally, however, they cannot reach the performance
of HDFS. This is due to the slower map phase as described in §5.2.3. For PCA, the output
data is small, and all approaches perform nearly the same.

The results reveal a trade-off between local renames and link files. Link files are able to match
the local renaming strategy in all of the workloads except Terasort. The main reason for this
difference is that link files still replicate data on upload, which gives them better reliability
guarantees at the price of slower performance. By contrast, local renames do not upload data.
However, because the link files approach only replicates data during the initial upload, which
is performed by reduce tasks in parallel, they are able to offer a 2.1× improvement compared

[Figure 5.8: Comparison of different renaming approaches for MapReduce workloads]

[Figure 5.9: Comparison of different renaming approaches for Facebook MapReduce workload]
 Chap. 5. Traffic Elimination for Distributed Storage I/O

![](https://example.com/chapter5/fig5_10.png)  Figure 5.10: Comparison of different renaming approaches for Spark workloads

![](https://example.com/chapter5/fig5_11.png)  Figure 5.11: Comparison of different replication strategies for MapReduce workloads

to vanilla Swift. This makes link files the best alternative when high reliability is required. However, if optimal performance is the primary goal, SwiftAnalytics can closely match HDFS in all cases.

**Facebook traces.** Figure 5.9 shows the CDF of job completion times for the Facebook MapReduce traces. The results show that for about 60% of jobs, all four set-ups perform equally. After that, the results start to diverge, and we observe a similar trend as for the above base workloads, i.e. SwiftAnalytics performs close to HDFS, while Swift with link files outperforms vanilla Swift. Around the 90\textsuperscript{th} percentile, SwiftAnalytics offers an improvement of 1.8× over vanilla Swift and 1.5× over link files. The results show that, in a production workload, there is a large amount of jobs that can benefit from SwiftAnalytics.

**Spark workloads.** As SwiftAnalytics is a generic solution that optimises the storage layer, which is shared across different data-parallel frameworks, we expect its benefits to also apply to other frameworks. To verify that, we deploy Spark and run Terasort. Additionally, as an example for an advanced analytics job, we also run query 3 from the TPC-H benchmark (Q3) in two variations, i.e. with and without the selection filters to vary the size of the query result.

Figure 5.10 supports our hypothesis. For Terasort and Q3 without filter, SwiftAnalytics outperforms both vanilla Swift and link files and reaches performance close to HDFS. When the output size is small (Q3 with filter), renaming does not affect performance anymore, and the completion times are nearly equal for all set-ups.

**Replication strategies**

We now discuss our choice of replication strategy. Figure 5.11 shows the completion times for the four base MapReduce workloads for each of the replication methods as presented in §5.4.3. Because I/Os are the bottleneck when the renames occur during the finishing stage of a job, we can see that job completion times increase linearly with each additional replica. While the impact of an additional replica on non-write I/O intensive jobs is small (an additional 15%
Figure 5.12: Comparison of local renames and link files under 1-way replication

for Wordcount), write I/O intensive jobs experience a major increase in completion time (up to $5.7 \times$ for Terasort).

This shows that there is a trade-off between performance and reliability. We argue that, for analytics jobs, less reliability is tolerable due to two reasons: first, we know we have one existing copy of our data and within the next consistency window, all replicas will be generated; second, if results are lost during this window, it is easy to regenerate them by rerunning the job. For example, Figure 5.11 shows that it is faster to run a Terasort job twice with only a single replica than running it once with an additional replica. For these reasons, local renames with no replication is the default mode of our approach.

**Effectiveness of local uploads**

The main reason why SwiftAnalytics outperforms link files in §5.5.1 is that link files leverage 3-way replication in Swift. To compare the two approaches under equal conditions, we deploy Swift with link files with a replication factor of 1 and run Terasort with a varying number of reducers. Figure 5.12 shows the results.

We see that SwiftAnalytics still performs better than Swift with link files. While 1-way replication improves job completion time with link files by a factor of 2 (compared to Figure 5.8), it is still 45% slower compared to SwiftAnalytics in case of 64 reducers. This discrepancy increases for smaller numbers of reducers and reaches 129% for 8 reducers. SwiftAnalytics produces stable completion times in spite of fewer reducers, whereas the variance for link files increases heavily.

The reason is that when using Swift with link files, objects still are written locally and then uploaded to the object store, creating an additional copy, whereas SwiftAnalytics can leverage its local upload mechanism to avoid that copy. The effect of the additional I/Os generated by link files is exacerbated when fewer reducers are used because a single reducer has to perform more I/Os. As previously explained, this additional copy could be avoided by streamed uploads but not all object stores provide this functionality.
Another drawback of link files is that they still rely on object placement according to its hash value. This means that output objects might create hotspots when written to the same location. Local renames do not suffer from this problem as each reducer generates and uploads the object locally.

### Overhead on regular operations

Next, we assess the overhead incurred by locality-aware writes by comparing the latency of GET requests in vanilla Swift and SwiftAnalytics. We first create input files of different sizes and then upload these files both with and without an LID. A client sends continuous GET requests to the objects, and we measure the latency. During this experiment, we turn off authentication in Swift to avoid overheads originated from the authentication process. The CDFs are shown in Figure 5.13.

For smaller objects (10 bytes and 10 KB), we can observe a slight increase in latency in the median and a more pronounced increase at the 70th percentile for SwiftAnalytics. As Swift already performs random load balancing, only 1/3 of all requests need an additional lookup which is roughly the 70th percentile mark. For larger objects (1 MB), we find that median latencies are equal to those of vanilla Swift. This is because it is the data transfer, not the request itself, that contributes the major part of the latency.

While incurring some overhead, local renames only require an additional lookup in 1/3 of all cases. Additionally, as presented in Figure 5.13, the overhead is only in the range of milliseconds and becomes negligible for larger objects.

#### 5.5.2 Effectiveness of object chunking

Finally, we evaluate the effectiveness of object chunking. To vary the object-to-node ratio, we use a smaller scale Wordcount job on 8 GB of input data and vary the object chunk size. A large chunk size produces a low number of objects and hence a low object-to-node ratio. The
smaller the chunk size gets, the better locality and hence performance, we expect. Figure 5.14 plots the completion time for only the map phase of the job.

For a large chunk size of 4GB, the input data is only stored in 2 objects in Swift, which leads to highly skewed access. In that case, SwiftAnalytics experiences a $4.6 \times$ slow-down compared to HDFS during the map phase. As the chunk size decreases, more objects are available, and the performance of SwiftAnalytics converges to that of HDFS. This confirms that chunking is necessary to achieve an even distribution of data across the cluster and to prevent skewed data accesses.

5.6 Summary

In this chapter, we presented SwiftAnalytics, an object store that serves as an efficient storage layer for data-parallel analytics frameworks such as Spark and MapReduce. In a detailed performance analysis, we have identified the major performance overheads introduced by the object store and found that an unmodified object store can cause analytics queries to slow down by up to $4.6 \times$ during reads and $15.8 \times$ during writes. This is due to the decentralised way in which object stores store and locate data. SwiftAnalytics provides mechanisms to solve these problems and improve read and write performance by transparently chunking objects and providing placement control via locality-aware writes respectively. The two techniques help queries on SwiftAnalytics achieve completion times close to their performance on HDFS.

Locality-aware writes complement consistent hashing by introducing a second, lexicographical namespace to applications. While the first, numerical namespace consists of logical node identifiers, the lexicographic namespace exposes the hostnames as physical node identifiers. Storing an object in the second namespace allows to specifically pick a storage node to achieve high data locality both during reads and writes. Locality-aware writes can be used to implement fast rename and upload operations. To not affect scalability and keep the
decentralised architecture of the object store, location metadata is stored with the existing object metadata in a decentralised way.

SWIFTAnalytics resembles a network-aware solution as it explicitly adds the concept of locality, i.e. the position of nodes/data in the network, to a location-agnostic system. We demonstrated that understanding locality helps eliminate unnecessary network traffic and also disk I/O, which significantly reduces job completion times compared to a system without control over locality.
Conclusions

The impact that Big Data had on science, industry, and our daily lives during the last years cannot be denied. It enabled major scientific breakthroughs and allowed to “personalise” the Internet. With ever denser and cheaper storage technologies and the increased connectivity of the world through mobile devices and the Internet of Things, the amounts of data will only keep increasing. However, more data is only valuable if relevant insights can be extracted from it with reasonable effort and within a reasonable amount of time.

Data-parallel frameworks such as MapReduce, Spark, and Flink have been designed to meet these requirements. Their massively parallel, scale-out architectures allow to automatically parallelise computation across a large number of machines and CPU cores. Combined with their declarative, operator-based programming interface, these systems provide a simple way of specifying and executing queries on large clusters. Users do not have to be experts in distributed systems anymore and can rather focus on the domain-specific data processing tasks. While distributed computing based on scale-out enables processing data at scale, it also means that network transfers during query execution are increasing. Some queries require to exchange a large amount of data between the different worker nodes and, with larger input datasets, network communication becomes critical for the performance of data-parallel frameworks.

In addition, the underlying network interconnect is becoming increasingly more complex as clusters are scaled out and grow larger. Data centre networks form complex topologies with
different properties such as heterogeneity or oversubscription and host a variety of different applications, which share the network and lead to unpredictable, bursty traffic. Data centre networks pose new challenges for data-parallel frameworks as these frameworks now need to be aware of the specific problems that arise when running in a data centre environment. On the other hand, new opportunities arise for data-parallel frameworks to benefit from the closed and controlled data centre environment.

In this thesis, we presented ways of achieving this network awareness for the three main types of traffic that occur during large-scale data processing: (i) aggregatable traffic, (ii) non-aggregatable traffic, and (iii) distributed storage I/O. Each of these classes has opportunities to benefit from a tighter integration of the data management application and the network. We showed that aggregatable traffic can be reduced as it flows through the network to minimise the amount of transferred data. Furthermore, we demonstrated how non-aggregatable traffic can make optimal use of the available resources by monitoring throughput and adapting to interference from other applications. Finally, we described how explicit locality control can remove unnecessary network I/O for distributed storage traffic. We implemented the above approaches in three systems that are able to outperform existing solutions by incorporating explicit knowledge about the physical network and its traffic. The following summary of this thesis provides a short overview of the three techniques.

The network is likely to remain a problem for future data processing applications. With the advent of fast storage technologies such as NAND flash [SLM16] and storage class memory [FW08], more pressure will be put on the networking resources. This means that network awareness will remain important and necessary, even as networking hardware becomes faster.

6.1 Thesis summary

This thesis started by providing an overview of the different components involved in Big Data processing: (i) the data centre network, (ii) data-parallel frameworks, and (iii) distributed storage systems.

We first discussed the structure and topology of data centre networks in detail. We introduced the problem of oversubscription, i.e. the inability of a data centre network to support communication between all end hosts at full rate. We discussed different, full-bisection bandwidth topologies to overcome oversubscription in a cost-efficient way. While the existing approaches improve the performance of data centre networks, they all come with different drawbacks and hence application support remains important. We then introduced middleboxes and SDN technologies. Using middleboxes, network functions such as firewalls or load balancers can be implemented in software; SDN on the other hand opens up the network control plane
to applications. While these two technologies allow for a certain degree of application control, the exposed interfaces are still low-level and do not support more complex, application-specific processing on different network flows. We further discussed traffic patterns and found that data centre traffic is largely unpredictable and prone to skewed accesses. This causes interference between applications and results in performance degradation.

After that, we gave an overview of data-parallel frameworks. These frameworks follow a data-parallel processing model in which different tasks operate on different input data partitions in parallel. Queries are written using an operator-based API with higher-order function operators. We discussed two types of queries in detail: partition/aggregate and join queries. Partition/aggregate queries first process the input in parallel on different worker nodes and then collect the results at one or few master node(s) for aggregation. This generates congestion at the master(s) and in an oversubscribed network core. While edge-based early aggregation approaches exist, they cannot reduce traffic enough to mitigate the congestion. Join queries repartition the input data and shuffle it across the network. They are susceptible to network interference as their completion time depends on the slowest worker node. Existing approaches rebalance skew by migrating join state but that adds additional network traffic and hence is infeasible for network skew.

Finally, we discussed distributed storage systems. We categorised them into block-based and object-based storage. Block-based storage stores large files as smaller blocks and uses a central repository to store block locations and file metadata. On the other hand, object-based storage has a decentralised architecture. While this provides higher scalability, it makes certain operations, in particular renaming an object, more expensive as it requires to copy data. This is a disadvantage when object stores are used as a storage backend for data-parallel frameworks. Queries need to rename the output data several times to ensure correctness of the results, which reduces performance. Existing approaches avoid renames to improve job performance but either cannot guarantee correctness in all cases or trade off usability.

We introduced three network-aware systems to optimise different types of traffic. We presented NetAGG, a system to reduce aggregatable traffic in the network during partition/aggregate queries. Partition/aggregate queries follow a many-to-few traffic pattern, which leads to network congestion at the few receivers and between oversubscribed network layers. NetAGG is based on the observation that aggregation functions are often associative and commutative and reduce data. Additionally, it exploits the fact that data centre networks are structured in tree topologies and contain a large number of middleboxes. NetAGG constructs in-network aggregation trees in which middleboxes constitute the tree nodes and execute the aggregation function at each physical network hop. As data flows along those trees through the network, it is aggregated and thereby reduced. By performing the aggregation in the network compared to just on end hosts, NetAGG is able to significantly speed up partition/aggregate queries.
We described the architecture of NetAgg, which allows to transparently expose its in-network aggregation service to existing deployments of data-parallel frameworks. This is achieved by shim layers that intercept traffic at the network socket layer and forward it to the nearest middlebox for aggregation. Additionally, we explained the architecture of the aggregation middleboxes, called Agg boxes. Agg boxes perform aggregation at high rates due to their task-based, parallel design, which allows them to execute an aggregation function across multiple CPU cores and can achieve processing rates of up to 9.2 Gbps, enough to saturate a 10 Gbps link. Agg boxes can be shared by different applications. We used an adaptive weighted fair scheduling algorithm to split the resources of the Agg box between applications according to a predefined ratio. NetAgg can improve queries to the distributed search engine Solr by 9.3× and Hadoop MapReduce jobs by 5.2×.

Second, we described SquirrelJoin, a novel join algorithm that dynamically reacts to network skew. Network skew can occur in shared data centre networks when some links carry more flows than others. This leads to interference and creates stragglers, which can cause a significant increase in the join completion time. SquirrelJoin mitigates the stragglers and balances processing by dynamically changing the data partitioning and sending more traffic to non-straggling workers. To detect network skew, SquirrelJoin constantly monitors the throughput of all receiving workers and estimates their completion times based on their current throughput. If the difference between the fastest and slowest worker is beyond a certain threshold, SquirrelJoin attempts to balance the skew by computing a new partitioning.

To avoid having to migrate existing join state when data is repartitioned, we introduced lazy partitioning. Lazy partitioning initially buffers all records at the sending workers and thereby prevents receiving workers to accumulate state for the keys of these records. As a result, these keys can be arbitrarily distributed across the nodes while still guaranteeing correctness. We presented the architecture of SquirrelJoin, which minimises the overhead of lazy partitioning by avoiding synchronisation between worker nodes and decoupling the balancing logic from the workers. We then described how SquirrelJoin uses basic data smoothing techniques and statistical testing to become robust against noisy completion time estimates of the workers. We also presented a formal model of a distributed join to analyse the potential of balancing sending and receiving workers and found that balancing senders can only yield a small improvement. SquirrelJoin can speed up distributed joins by up to 2.3× under network skew conditions.

Third, we presented SwiftAnalytics, a novel object store architecture which is designed to efficiently support data-parallel frameworks as a storage layer. When data-parallel frameworks are reading to/writing from an object store, they experience a decrease in performance due to the different, decentralised architecture of object stores compared to distributed filesystems. In a detailed performance study, we identified two major problems, which are read and write locality. We introduced the design of SwiftAnalytics, which solves these problems by
6.2 Future work

During the process of writing this thesis, we identified several directions for future work:

**Application-specific network functions.** NetAgg demonstrated that application-specific in-network aggregation can significantly improve query completion times in data-parallel frameworks. However, other network functions, such as load balancing or multi-cast, can also benefit from application knowledge. For example, an application-specific multi-cast service could reduce traffic during a query when the same data needs to be sent to multiple nodes. Hence, a general framework that allows to easily implement and deploy arbitrary application-specific network functions that can be executed at high rates is an interesting future direction. Flick is one step towards that direction and provides a domain-specific language and efficient runtime for implementing network functions [ACM+16]. Going forward, we plan to look into automatic deployment mechanisms, e.g. based on SDN, to re-route application data through the correct middleboxes without the need for shim layers.

**Balancing sending workers during network skew.** We showed that lazy partitioning is effective in balancing network skew across receiving workers while sending workers cannot benefit from such an approach. However, sending workers are also prone to becoming stragglers due to network skew, and SquirrelJoin does not have the means to handle this case at the moment. We plan to develop SquirrelJoin further to also support sender balancing. In particular, we are looking into exploiting data redundancy that is usually present in distributed storage systems via replication. If input data is present more than once, it allows us to read and send the same input data from different sources and thereby increase the sending load of non-straggling senders while reducing the sending load of stragglers.

transparency chunking objects into smaller parts to improve the aggregate read throughput for large objects, and by providing placement control to the object store. Placement control allows to explicitly specify the storage node for an object to eliminate additional data copies that occur when objects are renamed.

We introduced locality-aware writes as an approach to provide placement control in an object store without affecting its scalability. Locality-aware writes use a second object namespace in which users can specify the exact location of an object. This location information is stored in a decentralised way to avoid introducing a central component that could become a scalability bottleneck. We used locality-aware writes to implement fast rename and upload operations for the object store that avoid copying data across the network by only locally moving it on a single host. SwiftAnalytics is able to speed up data-parallel jobs by a factor of up to $5.1 \times$ due to its fast rename and upload functions while its object chunking mechanism can provide a speed-up of $3.4 \times$ for reads.
A general system for skew mitigation based on lazy partitioning. We have successfully used lazy partitioning to balance network skew during distributed join processing. However, lazy partitioning is currently unable to balance other types of skew such as CPU skew or data skew. We are planning to investigate if lazy partitioning can be extended to other types of skew. Additionally, we want to consider offering skew mitigation as a service, similar to NETAGG’s aggregation service. This will help any data-parallel framework robustly detect and efficiently balance any type of skew, increasing the overall utilisation of the cluster while delivering faster query processing times. This is especially interesting for long running queries, such as in stream processing, as the environment/data distribution may change significantly over time.

Optimising cloud-based object stores for analytics. With SWIFTANALYTICS we showed, how an object store can be optimised to serve as a more efficient storage layer for data-parallel frameworks. However, many existing deployments do not have the flexibility of changing the object store itself as they are using proprietary cloud offerings such as Amazon S3 [Ama16] or Google Cloud Storage [Goo17]. These deployments can still benefit from faster analytics over the data stored in the object store, and we are planning to extend our study to investigate the problems that arise in such a set-up, i.e. when the object store is remote and under the control of a third party. We then want to propose solutions that do not require changing the object store while still yielding performance benefits.

Object stores as a hybrid backend. Another interesting question regarding object stores is their primary use case. Some object stores, such as Facebook’s Haystack [BKL+10], are used as an image store and serve images to the Facebook web application. Other use cases include cloud and archival storage. Each case has different requirements in terms of objects and access patterns. For example, an image store has many small objects with frequent reads and writes while archival storage may host large objects with mainly writes. In a cloud environment, different tenants have different use cases and workloads might span a large variety of object size distributions and access patterns. We plan to investigate the trade-offs of different use cases when analytics should also be supported efficiently by the same object store deployment. For example, one potential problem could be object granularity, as analytics work best on large objects but do not handle a large number of small objects well. Another problem might be interference of read intensive analytics with read intensive object store workloads.
References


References


References


[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter


References


[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed


References


