Imperial College London
Department of Computing

Stateful Data-Parallel Processing

Raul Castro Fernandez

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, September 2015

Abstract

Democratisation of data means that more people than ever are involved in the data analysis process. This is beneficial—it brings domain-specific knowledge from broad fields—but data scientists do not have adequate tools to write algorithms and execute them at scale. Processing models of current data-parallel processing systems, designed for scalability and fault tolerance, are stateless. Stateless processing facilitates capturing parallelisation opportunities and hides fault tolerance. However, data scientists want to write stateful programs—with explicit state that they can update, such as matrices in machine learning algorithms—and are used to imperative-style languages. These programs struggle to execute with high-performance in stateless data-parallel systems.

Representing state explicitly makes data-parallel processing at scale challenging. To achieve scalability, state must be distributed and coordinated across machines. In the event of failures, state must be recovered to provide correct results. We introduce *stateful data-parallel processing* that addresses the previous challenges by: (i) representing state as a first-class citizen so that a system can manipulate it; (ii) introducing two distributed mutable state abstractions for scalability; and (iii) an integrated approach to scale out and fault tolerance that recovers large state—spanning the memory of multiple machines. To support imperative-style programs a static analysis tool analyses Java programs that manipulate state and translates them to a representation that can execute on SEEP, an implementation of a stateful data-parallel processing model. SEEP is evaluated with stateful Big Data applications and shows comparable or better performance than state-of-the-art stateless systems.

Acknowledgements

I want to thank Dr. Peter Pietzuch for making this thesis possible: for giving me the opportunity to work in his group; for his careful supervision at different levels over four intense years; and for his contagious enthusiasm in research, that motivated me day after day. I am proud to have learnt what I know about research from him.

I want to thank Matteo, Eva and Paolo, who helped in countless ways. Our off-topic conversations challenged and motivated me, and ultimately helped me to become a better researcher. I also want to thank Richard Mortier and Cristian Cadar who carefully read and helped me to improve this document after a great discussion.

I thank my parents and my brother for trusting every decision I made, they have helped me to become the person I am today.

I am proud of having Jorge, Christian, Victor, Patri and many other wonderful people by my side. They have always been and will always be there. Thanks to the London crowd for their support during my time in that fantastic city. They made everything easier. Special thanks to Laura, with whom I shared most of these four years. She listened every time to whatever I had to say, helped every time and her smile made everything possible. We suffered together and she always understood and supported my goals and passion. For that I will always be grateful.

I want to thank to the fantastic people with whom I shared office all these years. They managed to make working in office 346 a pleasant experience. Special thanks to Ioannis, Andreas, Lukas, April, Luo, Alexandros, Dan and Matthias for the work done together (Matthias, Alex), the off-topic conversations, and the post-deadline whiskies that eased all the moments of hard work

I want to thank Mike, Sara and Alvaro with whom I shared most of my time during my undergraduate studies. I have learnt invaluable skills from all of them, and I am lucky to have opportunities to keep doing so in the future.

I also want to thank the LinkedIn team with whom I spent a fantastic Summer in 2014. Dong, Joe, Chris, Jun, Neha, Jay and Mammad all helped me with their invaluable advice and their technical expertise.

The idea of pursuing a PhD stems from goals and passion. I want to thank those who have motivated me fueling these. Sometimes they have done that over years, other times over coffee conversations, but they all mean something if included below:

Alvaro Agea started everything, he is the reason why I am doing what I do today. Peter Pietzuch is responsible for recovering my hope in research. Vittorio challenged and motivated me in ways he probably does not even imagine. Kevin Cuturi, Peter Alvaro, Martin Kleppmann, Anand Rajaraman, Chris Re, Itxaso and Dave all contributed in one way or another to this thesis.

To you black gold, for always accompanying me, during the highs and the lows of this adventure. Forever thank you.

Declaration of Originality

This thesis presents my work in the Department of Computing at Imperial College London between October 2011 and September 2015. I declare that the work presented in this thesis is my own, except where otherwise acknowledged.

Copyright Declaration

Contents

List of Figures

Chapter 1

Introduction

Today, most digital services and products depend on the analysis of large volumes of data. We find online content through search engines that use gigantic indexes of all the addressable web pages in the Internet to produce results [Eco]. Online recommendations which suggest to us content that we may find interesting, are built from the aggregated data of many users. Other examples are natural language interfaces that permit us to send voice commands to mobile devices such as Microsoft Cortana [Mic], Apple Siri [App] or Google Now [Goo]. The list of services that use data to increase user-perceived value goes on and on. Yet, these are just a few examples of cases where data analysis is useful. Internet companies have based their business models on the commercialisation of user data, for example, with the use of sophisticated ad-targeting platforms. In a broader sense, science benefits from new sources of data to access new insights that permit to make progress faster. Examples can be found in bioinformatics [Sav14], social sciences [CHKS14] and, more generally in *data-driven science* [HTT09] that emphasises data analysis as an integral part of the scientific process.

As data analysis becomes part of the day-to-day activities of data scientists, the demand for more sophisticated analysis platforms increases. The need to analyse large volumes of data triggered several breeds of systems that scale their distributed infrastructures to clusters or public clouds. These systems, however, do not support imperative programming languages. The main reason is that they all build on top of stateless data-processing models that make representing state explicitly difficult. In this thesis, we present stateful data-parallel processing, a new processing model that permits an explicit representation of state and therefore enables analysis programs in imperative languages.

1.1 The Big Data Era

Many organisations have relied on data analysis to provide services and products for decades. These analyses have traditionally been performed in *data warehouses* that store data and facilitate its analysis [CD97]. However, the unprecedented volume of data generated and collected in the last decade has outpaced the capabilities of traditional data management technology. *Big Data* encompasses new problems in data management: (i) it introduces scalability challenges in the data management solutions of traditional organisations; and (ii) it creates new opportunities that spark new, innovative ways of uncovering hidden value. This new angle to an old problem has produced a renewed interest in data management technology. One particular example is the appearance of *data-parallel processing systems* that leverage distributed shared-nothing architectures to permit the analysis of ever-growing amounts of data. The design of these systems in particular, and of data analysis technology in general, is motivated by the changes in the data management landscape that occurred over the last decade.

We identify three trends that have converged to shape modern data analysis systems: (i) the combined storage cost decrease and the global adoption of the Internet allowed to store large volumes of data cheaply; (ii) the development of new hardware and software systems to facilitate Big Data analysis; and (iii) the availability of cost-efficient pools of computing resources enabled by the commercialisation of cloud computing. These three trends have fostered innovation and enabled new products and services based on data. This, in turn, has attracted more people—with new, more complex applications in mind—to the data analysis life-cycle, thus creating more and more data. This creates new challenges that hinder the full extraction of value from data analysis, as summarised below.

Data becomes cheap

The Internet evolved from a read-only platform used to navigate content through hyperlinks to a write-intensive platform where millions of people upload all types of content, such as posts in social networks or multimedia content such as music or video. Unlike in the read-only era where the content hosted in the Internet grew slowly, the write-intensive era generates petabytes of information daily [hkb]. This data is valuable for companies that are motivated to acquire more and more data. By carefully analysing this data, it is possible to detect user-behaviour patterns, which can increase the quality of services and products. This reinvestment of value further increases user engagement, which in turn, boosts data growth.

These large volumes of data can be stored due to the decrease in storage prices. There are two factors that contribute to this cost reduction: (i) reduced hardware costs makes terabyte devices a commodity; and (ii) the development of distributed file systems, which work on cheap off-the-self high-storage servers, enable the aggregation of the storage capacity of multiple machines.

Data becomes cheap due to the storage cost reduction, and companies are willing to store more data for increased future value. In this situation, the bottleneck in data analysis shifts from data generation and storage to data processing. This need motivates the development of sophisticated ***data-parallel processing systems*** *designed to process an always increasing amount of data.*

Quick evolution of data-parallel processing systems

In order to cope with this new processing bottleneck, organisations must find scalable data processing technologies that allow them to process large volumes of data. Additionally, such solutions must account for future data growth, and for this reason, it is crucial that they are cost-efficient. In particular, scaling up—adding more powerful hardware—is not a viable solution, as data grows faster than Moore's law [JGL+14] with costs rising quickly. One cost-efficient alternative to scaling up is relying on shared-nothing architectures composed of off-the-self servers [Sto86]. These aggregate the computational resources of many machines, thus providing a powerful and cost-efficient platform.

New hardware infrastructure demands new software that can take advantage of the new characteristics. This software has two crucial requirements. One is scalability: it must scale to aggregate the throughput of distributed machines. Second, the software must be fault-tolerant. With an increasing number of machines running in parallel, there is a higher probability of hardware failure [GGL03, GJN11, PWB07], so software must avoid data loss in the event of failure. **Data-parallel processing systems** appear as a scalable and fault tolerant solution to process large volumes of data in shared-nothing architectures, with MapReduce [DG04] as one of the early examples. However, with new organisations that want to analyse large volumes of data, new complex applications and workloads appear that accelerate the development of more sophisticated data-parallel processing systems.

Continuous data growth fosters the development of cost-efficient data-parallel processing systems that run on shared-nothing architectures. The main requirements for these systems are ***scalability*** *and* ***fault-tolerance****. The broad range of workloads and application use cases give rise to many specialised systems.*

Large-scale computing platforms at our fingertips: Cloud computing

Analysing Big Data with data-parallel processing systems requires large clusters of machines to provide results in a timely manner. These clusters are difficult to operate and expensive to maintain, which used to limit Big Data analysis to organisations with the resources to build and operate large clusters. The appearance of *cloud computing* [AFG+10] opened the possibility for small organizations and individuals to analyse Big Data. Cloud computing allows users to provision computing resources on demand and only pay for consumed resources, avoiding the costs of operating and maintaining large clusters. Instead of building data centres, organisations can buy resources from public cloud vendors, such as Amazon AWS [Amaa] or Rackspace [Rac], and deploy their data analysis solutions to perform analysis at scale. An additional advantage of the cloud model is its *elasticity* property: users have an initial pool of resources that they can grow or shrink, adjusting the used resources to current demand. The vision for computing as an utility is as old as computing [CK09], but it is not until the past decade that vendors have created a product out of that vision, and brought it to the general public.

The abundance of data and the appearance of cloud computing creates new opportunities to extract value from data. This causes the appearance of applications with more complex requirements, and encourages more people—with varied backgrounds—to become part of the data analysis process. With all these new stakeholders involved in data analysis, there is a race to find data-parallel processing systems that better capture the requirements of new applications, e.g. that efficiently support iteration as used in many machine learning applications. With the involvement of domain experts in the data analysis process, the programming models of data-parallel processing systems evolve to make programming these systems more accessible.

Easy and cost-efficient access to large pools of computing resources and data makes Big Data analysis accessible to more people. As a result, domain experts demand more complex applications. Under this scenario, data-parallel processing systems must evolve to provide support for more complex applications, and the programming languages of these systems become more important.

However, current data-parallel processing models hinder analysis opportunities because they are stateless. In a stateless model, dataflow graphs are composed of nodes that represent computation and edges that represent the data communication between nodes. This simple model facilitates achieving fault tolerance and scalability—properties fundamental to these systems. However, the lack of state precludes the straightforward representation of algorithms that are stateful, thus limiting the number of applications that can be expressed efficiently. Most importantly, a lack of state means that these models do not support high-performance execution of applications that require fine-grained updates.

Consider the case of an application that trains a machine learning model to classify emails. Additionally, the application requires to classify on-demand, i.e. with low-latency to not affect user experience, new incoming emails. Such an application must access state—the trained model in this case—with low latency, and be able to update it with fine-granularity (for the required user) as soon as possible to maintain a *fresh* view of data.

The stateless models of current data-parallel processing platforms are a hurdle towards the *democratisation of data* [AAA$^+$], the trend of more people getting involved in the generation, collection and analysis of data. When the analysis requires domain expertise, experts find it difficult to represent stateful algorithms in the stateless models supported by current systems.

In this thesis, we propose a new data-parallel processing model called **stateful data-parallel processing** that allows explicit state, therefore permitting a wide range of applications with fine-grained access to state to execute with scalability and fault tolerance in distributed clusters of machines. In addition, stateful data-parallel processing makes it possible to incorporate imperative-style languages.

Next, we give a summary of the technical challenges that give rise to the aforementioned hurdles: *how to provide scalability and fault tolerance in stateful data-parallel processing*. After that, we state the research contributions and conclude with an outline of this thesis.

1.2 Current Challenges of Big Data Processing

This section provides an overview of data-parallel processing systems structured along two different aspects. First, it summarises how the stateless dataflow representation used by these systems has evolved to incorporate new features such as branching and loops, as demanded by more complex applications. Second, it walks through the evolution of dataflow models towards higher-level constructs. Overall this section emphasises modern system requirements and exposes the shortcomings of their stateless processing models.

Data-parallel processing: Systems

Data-parallel processing systems designed to run on shared-nothing architectures have evolved over the last decade to incorporate new features for new use cases and workloads. All these systems receive as input applications that are represented as a dataflow graph—a directed acyclic graph (DAG), where nodes perform computation and edges represent the flow of data across the nodes. Dataflow graphs are a good representation to identify elements that can run in parallel and to recover from faults as explained below.

The MapReduce model [DG04] proposed by Google in 2004 offers a computation model formed by *map* and *reduce* tasks. Map tasks apply a function to the input data and sort the output results, and reduce tasks aggregate the map task outputs. The main benefit of this model is that map and reduce tasks can run in parallel by partitioning the

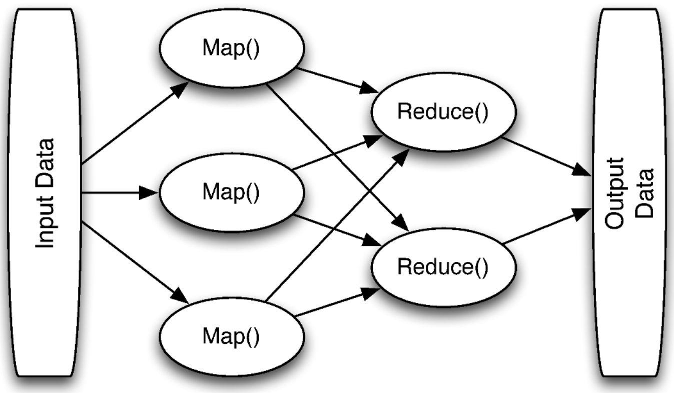

Figure 1.1: MapReduce dataflow model composed of map and reduce tasks.

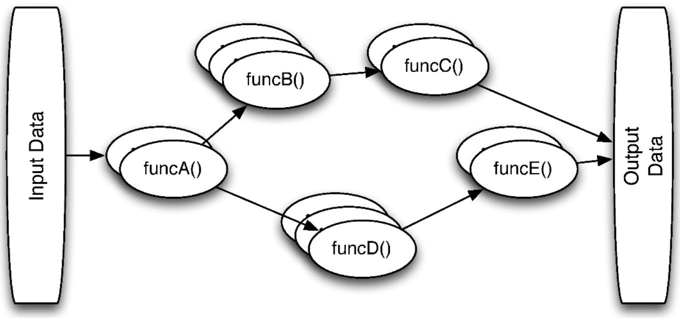

Figure 1.2: Dryad dataflow model forming a DAG.

input data, which increases the processing rate by aggregating the resources of multiple machines. The dataflow graph that expresses MapReduce computation is simple, as shown in Fig. 1.1, where *map* tasks receive input data that is sent downstream to *reduce* tasks that output the final result. It is indeed due to its simplicity and suitability to run over large datasets that the model has been adopted by many organisations that need to do large scale data analysis.

Dryad's dataflow model [IBY$^+$07] allows to represent more complex applications than MapReduce. Fig. 1.2 shows a dataflow that can be expressed with Dryad, where the nodes are not constrained to only map and reduce tasks. Instead, Dryad permits a fixed number of higher-order functions with known parallelisation strategies, which are depicted in the figure as different tasks interconnected in arbitrary ways, forming a DAG. In this way, Dryad's dataflow model allows to write applications in a declarative fashion [YIF$^+$08], while enabling scalable processing by partitioning each of the dataflow operators.

To achieve fault tolerance both MapReduce and Dryad follow a simple approach—they

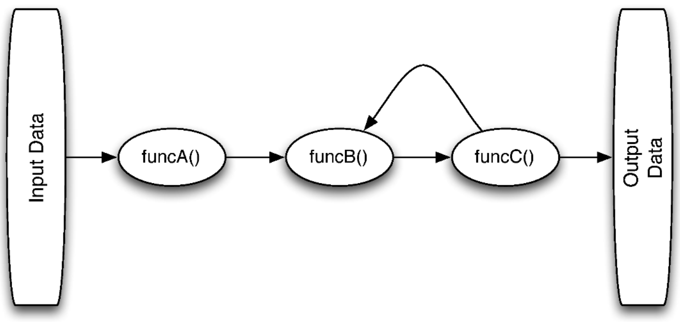

Figure 1.3: Spark dataflow model with iteration.

materialise all intermediate results produced by the processing nodes. Under the assumption that computation is deterministic—which is the common case—this approach reprocesses failed tasks by replaying the materialised data, therefore recovering and producing the correct results.

One common challenge of data-parallel processing systems is ***scalability****. Systems must process large volumes of data and support new, more demanding workloads.*

More recently, Spark [ZCD+12] and Naiad [MMI+13] have extended the expressiveness and performance of data-parallel processing systems. First, they both perform in-memory computation when possible, which results in throughput improvements of up to two orders of magnitude for certain workloads with respect to MapReduce. In addition, both systems represent iteration—commonly found in machine learning algorithms—in the dataflow graph, which permits more efficient loop execution. Fig. 1.3 shows an example of a *logical* Spark dataflow graph with a cycle in its graph. Note that during execution, loops are unrolled by Spark's scheduler. The reason why loops are more efficient in Spark is that computation happens in-memory, unlike MapReduce or Dryad that must materialise intermediate results for fault tolerance. For efficient fault tolerance, Spark maintains the lineage of transformations applied to data. In the event of failure, Spark recomputes such transformations in parallel by reapplying the operations recorded in the lineage.

Another important factor that has contributed most to the development of new systems evolution is the rise of new workloads. MapReduce was designed to work on top of distributed file systems where it reads data in a coarse-grained fashion and performs batch-processing. This is a common scenario for many organisations that dump data into scalable distributed file systems, where MapReduce runs periodically to execute batch-oriented applications. Batch-oriented processing systems focus on high throughput instead of low latency, i.e. an application may take hours to complete. Over the last years, many applications appeared that require low latency processing, or increase their value with lower latency. For example, applications such as fraud detection, online recommendation and monitoring alert generation, require low latency processing of large volumes of data to be useful.

Another challenge of data-parallel processing systems is to offer more general dataflow models and abstractions that allow them to adapt to ***varied workloads****.* ***One size does not fit all*** *means that different systems and architectures are proposed to cope with applications that demand, for example, low-latency processing.*

This shift in the workload requirements triggered the development of new systems based on stream-processing [AAB+05, CCD+03a] that process data in a fine-grained fashion, permitting to achieve low-latency results. Similar to batch-oriented processing systems, scalability and fault tolerance are the two key properties that drive their designs. For this reason, they also represent applications with dataflow graphs, such as Twitter Storm [TTS+14a], Apache Samza [Apac] or Google Millwheel [ABB+13]. However, the techniques required to achieve scalability and fault-tolerance in stream-processing differ from batch-oriented systems due to the different processing model. For example, materialising all intermediate results as in MapReduce would introduce too much overhead, precluding low latency processing.

Data-parallel processing systems have evolved to respond to two new trends. First, they adapt to new applications by offering more expressive models that allow arbitrary DAGs and loops. Second, they adapt to support application workloads that require latencies ranging from hours to seconds.

Data-parallel processing: Languages

The other major aspect of data-parallel processing systems that has experienced strong evolution is the programming language for writing applications. The main reason is again related to the desire for including better support for more complex applications. For example, the MapReduce model offers a Java template that requires implementing a map and reduce task per job, where inputs and outputs are fixed.

There have been many approaches to abstract this low-level Java interface. For example, Apache Pig [ORS+08] introduces a limited set of operators that facilitate the composition of different map reduce jobs. Apache Hive [HCG+14] offers a SQL interface on top of the MapReduce model: declarative SQL queries are rewritten as multiple MapReduce jobs. Similar efforts exist for Dryad [YIF+08] and Spark [XRZ+13, AXL+15]. All these approaches offer different interfaces, but they are fundamentally limited by the native stateless processing model of the underlying platforms, i.e. they cannot execute stateful algorithms efficiently—all state is represented as data that must flow through the dataflow graph. In addition, expressing a stateful algorithm through a stateless programming interface is not natural for most developers.

In parallel with the evolution of the stateless processing models, domain experts and data scientists have relied on frameworks such as Matlab or R, and imperative programming languages such as Java, Python or C++ to write data analysis algorithms. All these frameworks and languages have one important characteristic in common: they all represent and manipulate state with fine granularity, which permits domain experts to write algorithms concisely. However, the shortcoming of these frameworks is that they are not designed for large scale data analysis.

Despite an evolution of the interfaces exposed to developers on top of stateless dataflow models, the lack of state in data-parallel processing systems constrains the range of applications that can be executed with high-performance. This slows down the development of more complex applications that would benefit from explicit state.

1.3 Problem Statement

In summary, the shortcomings of current data-parallel processing systems are:

- **No support for state.** Stateless processing models prevent implementation of stateful applications that can be executed with high performance. Most stateless dataflow models were proposed for the first breed of batch-oriented data-parallel processing systems. As new workloads demand low-latency results, there is a need to support fine-grained updates to state, which is not supported by current models.

- **No support for imperative languages.** The lack of support for mutable state prevents the implementation of imperative-style programming languages on top of data-parallel processing systems. Such a programming model would allow a broader range of domain scientists to analyse large amounts of data.

In spite of the benefits of a stateful model, there are two reasons why current data-parallel processing systems do not support state. State makes it difficult to achieve scalability and fault tolerance because it has to be managed to not become a bottleneck and it has to be recovered after failures for correct results. For this reason, a stateful data-parallel processing system must address these two issues, which are the subject of this thesis:

- **Scalability.** Data-parallel processing systems must scale out to aggregate the throughput of multiple machines. The presence of state makes scaling out more challenging because the system must know how to handle the state (for example, partition it) in a way that preserves the application semantics, yet aggregates throughput.

- **Fault-Tolerance.** To achieve fault-tolerant data-parallel processing, a system needs to keep track of how much data has been processed, and recover intermediate results that may be lost in the event of failure. Adding state means that a system needs to control not only how much data has been processed, but it also needs to account for state. When updates to state occur in a fine-grained way, the information that the system must preserve increases, which further complicates the problem.

1.4 Research Contributions

We introduce **stateful dataflow graphs (SDG)**, a new dataflow abstraction that, in addition to data and computation, represents state explicitly in the dataflow. SDGs therefore can: (i) represent stateful algorithms concisely, for example, those that perform fine-grained updates; and (ii) capture the state used in imperative-style languages, therefore allowing the implementation of algorithms in imperative languages. We refer to this as **imperative big data processing**.

To achieve scalability and fault tolerance in the presence of state, we introduce two different distributed mutable state abstractions that allow the distributed execution of stateful tools. To deal with failures and scalability, we introduce an integrated approach for scale out and fault tolerance that relies on a set of state management primitives.

1.4.1 Stateful Dataflow Graphs

Stateful dataflow graphs separate data, state and computation and represent them explicitly in a dataflow graph. A data-parallel processing platform that implements SDGs must provide high-throughput and low-latency data processing. To achieve these properties, state is exposed as a first class citizen to the system, which allows to define state management primitives that can manage the state directly. For example, there are primitives to checkpoint the state—making it external to the system—or to partition it according to a partitioning function.

We introduce an integrated approach to scale out and fault tolerance that relies on the observation that a node failing and a node scaling out are equivalent from a state management perspective. The integrated approach then uses the state management primitives to achieve fault tolerance, recovering the state after a node fails. Additionally, the system uses the state management primitives to provide elasticity, which we define as dynamic scalability—the system scales out dynamically to adapt to varying workloads or algorithms requirements.

Stateful Dataflow Graphs expose state as a first class-citizen to the system. This makes it possible to introduce state management primitives that can be used to achieve scalability and fault tolerance, as required by data-parallel processing systems designed to run on shared-nothing architectures.

1.4.2 Imperative Big Data Processing

Making state explicit in SDGs provides an opportunity to represent the state found in imperative programming languages. We introduce an approach that transforms imperative Java programs into SDGs via static code analysis. The idea is that developers can write Java code, where they define state as attributes and different methods that receive input and use the state and logic to implement algorithms. A task then takes this Java class as input and produces an SDG, which works as an intermediate representation between user programs and the execution platform.

The SDG is then executed by the stateful data-parallel processing system. The compute nodes always access state locally, i.e. they do not perform remote state access to keep latency low. To allow compute nodes to scale out, two distributed mutable state abstractions are provided that permit the state to scale. To leverage opportunities for parallelism, users can annotate the state in a Java program with *partitioned* or *partial* annotations, indicating the distributed mutable state abstraction to use in each case. These annotations depend on the semantics of the algorithm: partitioned means that the state can be partitioned, while partial means that the state must be replicated.

Imperative big data processing allows the concise representation of algorithms by allowing the use of state and permits algorithms to be written in widely adopted languages, such as Java, C/C++, Python or Matlab.

1.5 Publications

During the course of the thesis, the following related publications have been authored:

- **Liquid: Unifying Nearline and Offline Big Data Integration**. Raul Castro Fernandez, Peter Pietzuch, Joel Koshy, Jay Kreps, Dong Lin, Neha Narkhede, Jun Rao, Chris Riccomini, Guozhang Wang. In 7th Biennial Conference on Innovative Data Systems Research, (**CIDR**), Monterey, CA, USA, (2015) [FPK+15]

- **Making State Explicit for Imperative Big Data Processing**. Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki and Peter Pietzuch. USENIX Annual Technical Conference (**USENIX ATC**), Philadelphia, PA, USA, (2014) [FMKP14]

- **Grand Challenge: Scalable Stateful Stream Processing for Smart Grids**. Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch and Avigdor Gal. 8th ACM International Conference on Distributed Event Based Systems (**DEBS**), Mumbai, India, (2014) [FWPG14a]

- **Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management**. Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki and Peter Pietzuch. ACM International Conference on Management of Data (**SIGMOD**), New York, NY, (2013) [CFMKP13]

- **Towards Low-Latency and In-Memory Large-Scale Data Processing**. Raul Castro Fernandez and Peter Pietzuch. Doctoral Workshop of the 7th ACM International Conference on Distributed Event Based Systems (DEBS), 06/2013, Arlington, Texas, USA, (2013) [FP]

1.6 Thesis Structure

The remainder of the thesis is organised as follows:

Chapter 2 presents background and related work. This chapter introduces a number of dimensions by which different systems and methods proposed in the literature are classified. We justify both the choice of these dimensions as well as the classification of the systems. This section acts both as a technical background of content and concepts used in the thesis as well as a related work section that puts the advances of past decades into perspective.

Chapter 3 describes Stateful Dataflow Graph (SDG), the new abstraction proposed to bridge program expressiveness and system performance. This chapter explains the characteristics of the model in depth, as well as the assumptions on which it is based.

Chapter 4 introduces the state management primitives and the integrated approach for scale out and fault tolerance. This chapter then focuses on system aspects and presents different techniques to achieve fault-tolerance under varying assumptions, and explains how to implement these techniques efficiently to maintain high-throughput and low-latency processing.

Chapter 5 focuses on the programming model of stateful data-parallel processing. We introduce an imperative programming model based on Java, and describes *Java2SDG* which is a compiler that takes annotated Java programs as input and produces SDG that execute in a stateful data-parallel processing platform. We explain the translation process performed by Java2SDG based on static code analysis as well as the code synthesis stages required to generate executable code. We also evaluate the efficiency of the programs translated by the compiler.

Chapter 6 is the evaluation chapter. It describes SEEP, a prototype implementation of a stateful data-parallel processing system, and evaluates its performance with several applications and benchmarks. This chapter answers practical questions related to the scalability, fault-tolerance and feasibility of stateful data-parallel processing and shows real usages of the model for smart grid analytics and in the context of a data-driven Internet company.

Chapter 7 summarises the contributions of the thesis and presents some future research lines.

Chapter 2

Background

We classify approaches aimed at overcoming the deluge of data into two families of Big Data processing systems: data-parallel processing systems, which are the focus of this thesis, and distributed programming models (DPM), such as distributed shared memory [TSF90] or message passing models [GLDS96], that have characteristics closely related to our contributions.

Data-parallel processing systems are a response to large and ever-increasing volumes of data [DG04]. They are designed to work on shared-nothing clusters formed of off-the-self servers, which are cheaper to maintain and operate [HB09]. Distributed programming models have seen a major adoption in HPC deployments [HDF11] that are built from specialised hardware with tailor-made software to achieve high performance. Typical applications of HPC deployments are scientific applications [Nat, Sch10, KEGM10].

The rise of the Internet and data volumes have accelerated the adoption and innovation of data-parallel processing systems to a point where the differences with the goals of DPM become blurry. The fast-paced evolution of these systems in both industry and academia uncovers new opportunities and challenges. This thesis studies one such opportunity—with our contribution of stateful data-parallel processing—and we use this chapter to present background on the area.

The focus of this thesis is on data-parallel processing systems. The adoption of these systems has exploded across new domains other than the original Web companies [WESM+10, LYYZ10], due to the reasons that we introduced in §1: (i) new domains generate data that needs analysis to capture its value; (ii) data-parallel processing systems can run on commodity servers without specialised hardware; and (iii) cloud computing means that more people can cheaply access large computational resources for intensive data processing.

To help domain experts that need the processing capabilities of data-parallel processing systems, these systems must: (i) be scalable to harness distributed computational resources; (ii) be fault-tolerant to continue operation when failures occur; and (iii) offer a stateful processing model that permits writing stateful algorithms and execute them with high-performance.

In this chapter, we highlight an important mismatch between data-parallel processing and DPM models. On the one hand, stateless data-parallel processing systems are scalable and fault tolerant, but they cannot execute stateful algorithms with high performance; on the other hand, distributed shared memory support stateful algorithms, but they rely on developers to capture the parallelisation opportunities, and their support for fault tolerance is limited.

We structure this chapter as follows. First, we start with an overview of basic concepts and definitions that are used throughput the thesis §2.1. Then, we survey how modern Big Data technology achieves scalability (§2.2), and fault tolerance (§2.3), and to describe their programming models (§2.4). We finish the chapter with a summary §2.5.

2.1 Big Data Landscape

Stateless data-parallel processing systems and DPM models are similar in that they both need to be scalable and fault tolerant [BBC$^+$02]. They, however, differ in the techniques employed to achieve these properties and, most importantly, in the programming models they expose to developers.

It is possible to differentiate Big Data processing models based on how they achieve scalability—i.e. how they expose parallelism to developers—and based on whether they expose computation state or not. Processing models have traditionally been divided

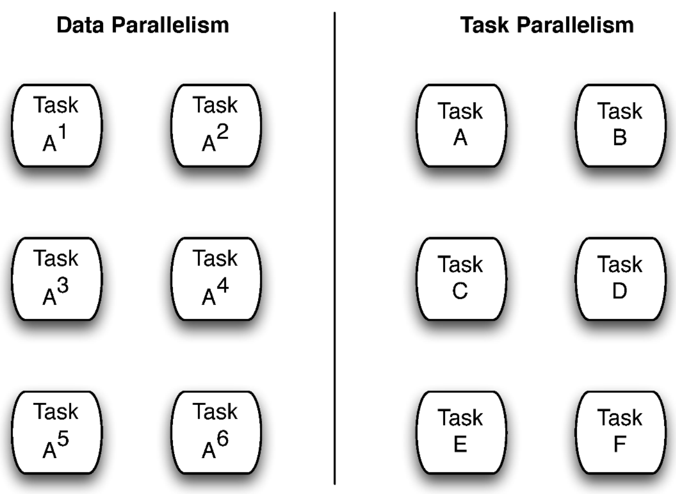

Figure 2.1: Task and data parallel models.

into two general classes: *task parallelism* and *data parallelism* [SSOG93]. According to whether they expose state or not, they can be *stateless*—when the computation state is not explicitly seen by developers—and *stateful* when developers can freely manipulate the state. We explain these characteristics in more detail:

Parallel processing models. According to how the processing model coordinates distributed resources for parallel execution, we say that it performs task parallelism or data parallelism. Task parallelism consists of tasks performing different computation over the same or different data, while data parallelism refers to performing the same computation over different chunks (i.e. partitions) of the data. Figure 2.1 illustrates both models. In a pure task parallel model, developers choose the granularity of the tasks, i.e. assign computation to each task. Coordination between dependencies between tasks is also explicit, for example, through communication primitives. Instead, in a data parallel model developers must choose how to partition the data, which is usually done according to the number of available resources, but they do not need to choose the tasks.

When employing data parallelism, the data is partitioned as illustrated by the cloud in Fig. 2.2 so that each machine in a cluster processes an individual chunk of small size, with all machines processing the dataset in parallel [DG92, DGG$^+$86]. With more computational resources, data can be partitioned further, therefore reducing processing time.

Typically, DPM models provide interfaces to perform both task- and data-parallelism [DM98], allowing flexibility when writing programs. The way in which these facilities are pro-

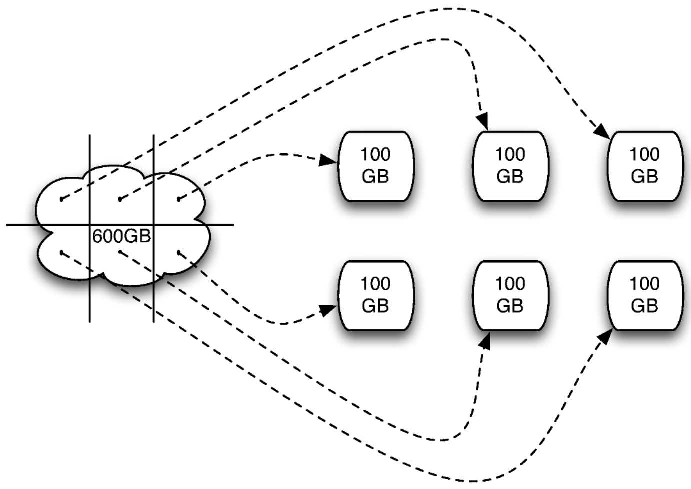

Figure 2.2: Data Partitioning for scalable processing.

vided, however, differs across different systems and domains. For example, in High Performance Computing (HPC) environments, most workloads are CPU- or memory-bound, and therefore a processing model that allows developers to split computation or add more resources with maximum control is more appropriate. In contrast, Big Data processing workloads are traditionally IO-bound [DG04, ZCD^{+}12], due to the large volumes of data required to process. For this reason, data-parallel processing systems designed for Big Data processing expose dataflow-based abstractions that facilitate data parallelism, while inherently extracting the necessary task parallelism from the program definition, i.e. with minimal user intervention. These dataflow models permit the run time system to scale throughput with the number of resources available. We survey techniques and systems in §2.2.

Stateless and stateful processing models. According to whether a processing model exposes computation state or not—mutable data structures that are modified by programs—data-parallel processing systems can be stateless or stateful, (see Fig. 2.3). Exposing state facilitates, for example, stateful programs and their performant execution properties. However, state has to be managed for fault tolerance and scalability by developers. Stateless models do not expose state, which facilitates achieving these properties without further user involvement.

One consequence of the processing model of a system is the programming interfaces that it can support. For example, a stateless processing model can offer a stateful, imperative interface to programmers if state is represented as data, which prevents the

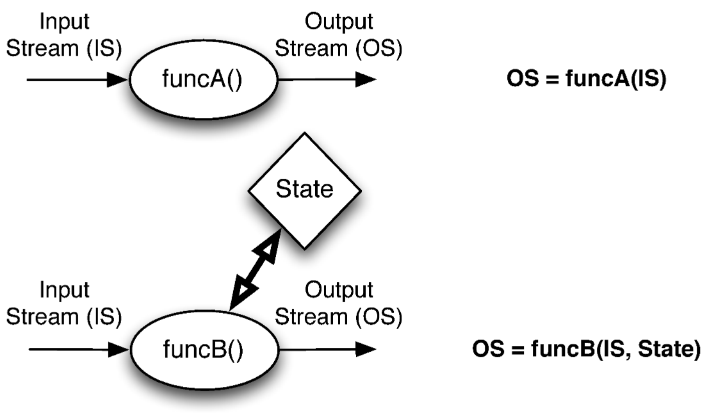

Figure 2.3: Stateless and stateful processing models.

high-performant execution of the algorithm because all updates to state must be propagated, even when they are small. We survey the programming interfaces of different systems in §2.4.

Fault tolerance approaces. Differences in the previous two dimensions mean that different techniques are required to make Big Data processing fault tolerant. Stateless data-parallel systems can implement fault tolerance techniques based on *logging data*, which means that the system internally keeps track of the data that has already been processed and coordinates automatically its storage and recovery. For DPM models that expose state, the typical approach consists of taking *checkpoints* of that state, which happens independently of data processing. We survey different approaches to fault tolerance in §2.3.

These three dimensions introduce different trade-offs. Task parallelism allows to decide on a fine-grained basis about the tasks that compose the computation. These independent tasks can all access shared state, therefore offering a stateful programming interface where it is easy to express many algorithms concisely. However, developers must explicitly write their programs to exploit data parallelism, they must coordinate access to the shared state to avoid concurrency problems, and the constructs offered to control the tasks are low-level and thus error-prone.

Stateless data-parallel processing systems extract task parallelism automatically as part of the *dataflow* that describes the application. This means that they handle task parallelism for developers that do not need to choose tasks manually. This is achieved at the cost of permitting developers only to write applications in stateless dataflow models.

Ideally a processing model for Big Data processing should offer the freedom of DPM models, i.e. with explicit access to state and imperative style code instead of constrained functions. It also should handle data parallelism, task definition and fault tolerance for users.

2.2 Scalable Big Data Processing

With increasing amounts of data and application complexity processing becomes a bottleneck that leads to longer job completion times. One way of addressing this is through more powerful hardware. This *scale up* strategy is however bounded by Moore's law [Sch97] and economies of scale: a small increase in hardware power may mean much higher costs.

There are two trends to combat limitations of Moore's law: multi-core architectures and distributed systems. The first approach seeks to develop software that can leverage the parallel resources of multiple available cores. Today it is easy to find off-the-self machines with tens of cores, and the number is only expected to rise [EBSA+11]. The massive parallelism gains of these architectures can help achieving higher performance for certain workloads.

The other approach is to use distributed systems. Instead of *scaling up* hardware for increased throughput, it is possible to add more machines of similar hardware characteristics and write software that can aggregate their resources—this is a *scale out* approach.

The focus of this thesis is on distributed systems. It is important to note, however, that all advances towards better software for multi-core and distributed systems contribute towards performance gains—these two approaches are orthogonal and complementary.

In the context of data-parallel processing systems, one can distinguish two different modes for achieving data parallelism. One applies in the case of *scheduled* systems, where a scheduler is responsible for assigning a task and a pointer to data to machines in the cluster [DG04, IBY+07, ZCD+12]. The other mode applies to *materialised* systems, where tasks are scheduled only once and stay on the same machine, continuously processing incoming data [PLS+06]. We found this classification useful because each mode requires different techniques to achieve scalability.

We next discuss the techniques to achieve scalability in scheduled and materialised systems as well as those employed by distributed programming models.

2.2.1 Scheduled Systems

In a scheduled data-parallel processing system, a logically centralised scheduler assigns tasks from the dataflow to nodes in a cluster. Any node in the cluster can receive any task and a pointer to the input data for execution. To avoid costly network, however, the scheduler assigns tasks following the principle of *data locality* [ZBSS$^+$10], i.e. it sends tasks to where the data resides.

Schedulers of data-parallel processing systems are data-driven [ZCD$^+$12, IBY$^+$07, MSS$^+$11]. The scheduling plan is given by the data dependencies expressed in the dataflow. Each node in the graph becomes a logical unit of scheduling, and executes in parallel in the cluster through tasks that implement such logic.

One assumption of scheduled systems is that input data is sharded across multiple machines [DG04, IBY$^+$07, ZCD$^+$12]. This is the usual setting in data-intensive scenarios, where data is stored in a distributed file system (DFS) [GGL03], such as HDFS [HDF]. Batch-oriented workloads benefit in such scenarios because tasks can run in parallel on each of the shards of data. For this reason, scheduled systems are typically employed in batch-oriented systems.

Batch-oriented data-parallel processing systems such as Spark or Flink, are designed for high-throughput processing. These systems do not satisfy particular latency requirements, but rather focus on minimising completion time [KTGN10]. To maximise throughput, they process data in a coarse-grained fashion [DG04, ZCD$^+$12], therefore optimising the computation-to-communication ratio. Costs due to network transfers as well as serialisation and deserialisation of data are amortised when data is delivered in *batches*—collections of individual data elements.

Dataflow systems for batch-oriented data parallel processing evolved from the simple, constrained model of MapReduce—composed of only *map* and *reduce* tasks—to the arbitrary DAGs of Dryad [IBY$^+$07], Spark [ZCD$^+$12] or Stratosphere [ABE$^+$14], which permit

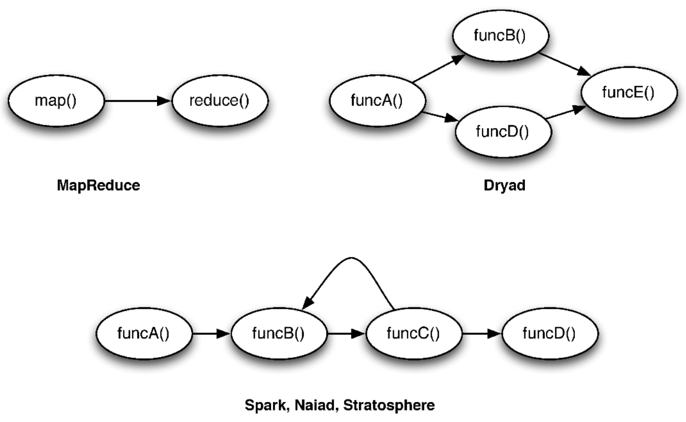

Figure 2.4: A glimpse of the evolution of dataflow graphs for expressing processing models.

the use of higher-order functions.

Fig. 2.4 depicts how dataflows have evolved from the simple MapReduce model to arbitrary DAGs introduced by Dryad and followed by Spark. The major innovation of Spark [ZCD+12] is that it permits fault tolerant execution even when the processing occurs in-memory. One immediate advantage of in-memory processing is the efficient execution of iterative jobs; the per-iteration result is stored in-memory and therefore the execution becomes efficient. This is in contrast to systems such as MapReduce and Dryad that need to materialise results after each processing step to enforce fault tolerance properties.

Spark is not the first system that permits iterative processing. Before, systems such as Haloop [BHBE10], Twister [ELZ+10] or CIEL [MSS+11] permitted the execution of iterative jobs. Later, systems such as Naiad [MMI+13] or Stratosphere [EST+13] improved on iterative processing by supporting incremental processing even in the presence of loops, i.e. maintaining the differences across iterations.

Scalability.

There are two main considerations when designing a scheduled system that is scalable. One is the scheduling overhead, and the other is the appearance of *stragglers* [ZKJ+08, MMI+13], which are nodes that underperform—for example due to an overloaded machine—slowing down the entire job.

The scheduling overhead must be amortised by the task execution lifespan for the system to be efficient [OWZS13]. As long as this happens, the scheduler can manage more machines to further partition the job. With very short-lived tasks, the overhead of the scheduler can dominate, becoming a bottleneck and limiting the scalability of the system.

Stragglers are common in large-scale deployments. They can be caused by skewed workloads, interference by other tasks in the same machine or software deficiencies [ZKJ+08, MMI+13]. To mitigate the negative effects of stragglers, schedulers use techniques such as work-stealing [DLS+09] that dynamically re-balance the load across the cluster, reassigning work from a straggler node to other nodes.

One fundamental assumption made by scheduled systems is that the dataflow graph is stateless, so that the scheduler can assign tasks without keeping state across scheduling decisions. This is also a basic requirement of work-stealing techniques, which must reassign tasks as needed to balance the work in the cluster.

2.2.2 Materialised Systems

We refer to materialised systems as those that perform a single scheduling stage, or *placement* during deployment where a task is assigned to a machine [PLS+06]. In normal circumstances, tasks await to receive data of a fine granularity that is processed with results sent immediately downstream [NRNK10, TTS+14b]. Each task is a vertex in the dataflow graph, and tasks communicate according to the data dependencies in the dataflow. The *sources*—i.e. tasks in the dataflow without input edges—receive data from some external system and the *sinks*—i.e. tasks in the dataflow without output edges—produce the final results to some external system. Although there is no scheduler, a logically centralised service performs the allocation and the monitoring of the deployment.

To achieve data-parallelism, a node in the dataflow graph is materialised on multiple machines that execute in parallel. Since all tasks are materialised, it is necessary to provide tasks in the cluster with *routing information* so that they can communicate with other tasks.

In contrast to scheduled systems, materialised systems excel at achieving low latency

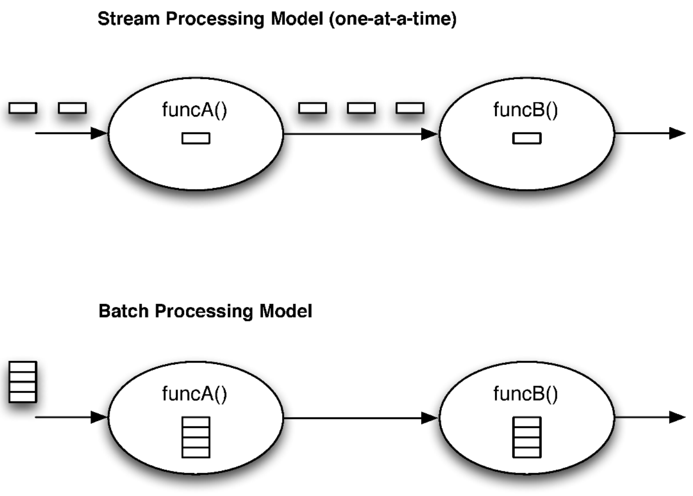

Figure 2.5: Stream versus batch processing models.

processing. First, there is no scheduling overhead during the data computation. Second, because all tasks are materialised, the execution of data permits *pipeline parallelism*, further reducing the latency per data tuple.

Low-latency processing is important for applications that need to process continuous streams of data, e.g. when analysing network data [CJSS03] or click-streams [MLSL04]. For this reason, stream processing systems are typically based on materialised designs.

Stream processing systems process data in a fine-grained fashion, or *one-tuple-at-a-time*. Fig. 2.5 shows both batch-oriented and stream processing models: the top processing nodes receive one tuple at a time, which is forwarded immediately after processing to downstream nodes. At the bottom, batch-oriented computation operates over batches of tuples: when a tuple arrives it is buffered until the batch is full. The computation-to-communication ratio of materialised systems is lower than that of batch-oriented ones, which usually means lower processing throughput per CPU core compared to batch systems, but it also yields lower latency.

The first generation of stream processing systems focused on supporting the relational operators in a streaming fashion [ACc+03, ABB+03, CCD+03b, CDTW00, AAB+05, GAW+08, BGAH07]. To achieve this, they introduced the concept of a *window* that bounds an otherwise infinite incoming stream to permit its processing by blocking operators such as `aggregation` or `join`. These stream processing designs led to implementations in industry such as Microsoft Streaminsight or IBM Infosphere Streams [Str, IBM].

A second generation of systems put an emphasis on distributed execution. Systems such as S4 [NRNK10] or Storm [TTS+14b] use a dataflow abstraction to achieve scalability, as explained previously. These systems were built by web companies (Yahoo and Twitter, respectively) that needed the scalability and fault tolerance due to their large deployments.

Last, systems such as Millwheel [ABB+13], Samza [Apac] and Chronostream [WT15] are examples of stateful systems that permit the use of state in their applications.

Scalability.

Materialised systems that use stateless dataflow graphs achieve data-parallelism by running multiple tasks in different nodes and propagating the necessary data between downstream and upstream tasks [NRNK10, TTS+14b, AAB+05]. Partitioning the stream typically has lower overhead, as this only involves performing round-robin assignment or look-ups in a data structure with routing information.

However, materialised systems suffer some of the same problems of scheduled systems, such as stragglers, which are more of a concern in materialised settings, where techniques relying on work-stealing are less applicable. Materialised systems are well suited for applications that can exploit pipeline parallelism, e.g. jobs where the computational cost of each task is high. When this is not the case, a scheduled system may be more efficient because it always uses all resources available in the cluster.

2.2.3 Dynamic Scalability: Elasticity

In the previous discussion, we assumed that both scheduled and materialised systems decide on a number of nodes that perform data-parallel computation statically. In some scenarios, dynamic scalability, referred to as *elasticity*, is needed instead.

Elasticity is the ability of a system to scale computational resources dynamically, i.e. without affecting system availability. This property is important for applications that execute on public clouds, where computation is sold as utility. An elastic system can adapt to changes in the workload, avoiding costly over-provisioning of resources. Elasticity is not a new concept in systems design [KC03], but it is a less explored property of data-parallel processing systems which only recently started to gain more atttention [MBF14].

Elasticity can also be beneficial for data-parallel processing systems in data centres. In such deployments, systems usually share clusters through some form of resource management, as offered by YARN [VMD+13], Mesos [HKZ+11], or Omega [SKAEMW13]. Applications running on shared resources come and go. For the cluster to be fully utilised—i.e. used in a cost-effective manner—it is necessary to assign idle resources to applications. Elasticity can help accommodate idle resources of applications with varying needs. Achieving elasticity is challenging from an operational point of view. It has implications on both fault tolerance and application semantics.

The mechanisms for achieving elasticity differ according to the data-parallel processing model. In scheduled systems, it is necessary to repartition the underlying data to the new available machines and then schedule more instances of the processing tasks on the new nodes. It is more challenging to achieve the same in materialised systems: they need additional information on how to route data to new nodes, while keeping the system operational and available.

2.2.4 Distributed Programming Models

We structure this section according to two relevant distributed programming models: message passing and distributed shared memory.

Message passing. In message passing systems, different processes—potentially distributed—communicate exclusively via messages. On receipt of a message, a process chooses to perform an action. Developers must define the task granularity and define and enforce the dependencies between tasks. This model supports explicit state in the computation, but it requires to specify all the tasks and their dependencies beforehand.

The two major examples of message passing models are Parallel Virtual Machine (PVM) [Gei94] and Message Passing Interface (MPI) [GLDS96]. More modern implementations of the message passing model are based on communicating sequential processes (CSP) [Hoa78] and actors, which are embedded in many modern languages such as Scala, Erlang or Go.

Orleans [BGK$^+$11], a recent system, uses the actor abstraction and provides an infrastructure that encapsulates state to provide fault tolerance.

Other variants of message passing restrict the model to offer additional guarantees. Pydron [MAAC14], extends the Python language with an additional API with communication primitives to permit one-sided communication where a master process coordinates a set of workers. The Julia programming language [BEKS14] embeds in its standard library a one-sided communication library and distributed collections to facilitate distributed computing.

Distributed shared memory. In distributed shared memory (DSM) systems, the system offers developers the abstraction of a single memory address space [Gei94]. This allows programs to access remote memory as if it was local. Updates to the logical memory view may need to be sent to underlying physical memories that comprise the shared abstraction. In this case, the system handles all necessary remote communication.

Unlike message passing systems, DSM systems hide communication issues for developers. However, they must still control and protect concurrent access to shared memory. Some systems that implement the DSM model are Treadmarks and Kerrighed [KCDZ94, MLV$^+$04].

Tuple spaces [Gel85] are an abstraction on top of DSM systems. Linda [Gel89] is a representative example of structured DSM systems. It offers developers the abstraction of a *tuple space*, with the aim of helping coordinate access to shared memory, thus reducing the necessary concurrency control and facilitating programmability. There are also commercial systems that provide a tuple space abstraction, such as Javaspaces [ss] and Tibco Activespaces [Tib].

Unfortunately all these systems suffer scalability problems due to the overheads of performing remote memory accesses over a network [NL91]. Some recent systems, such as Piccolo [PL10] or Oolong [MPL12], implement the DSM abstraction as a highly scalable key/value store, and they show that it is possible to implement scalable systems based on this paradigm.

There are more recent proposals to overcome the scalability limitations of DSM systems. For example, Partitioned Global Address Space [CEGNY03] optimises DSM so that

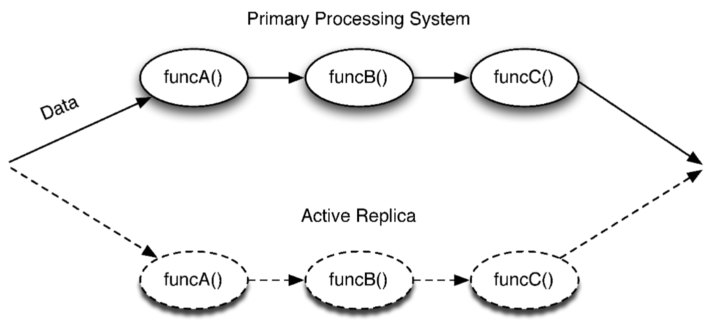

Figure 2.6: Fault tolerance through redundant computation.

the underlying memory is logically partitioned in a way that maximises access locality. Chapel [CCZ07] and X10 [CGS$^+$05] are two languages that implement the PGAS concept natively.

2.3 Fault Tolerant Big Data Processing

To operate at high-throughput despite the failures that happen in large-scale deployments of shared-nothing architectures—caused by software bugs or misconfigurations—systems must be fault tolerant [DG04, GJN11, PWB07]. There are different notions of fault tolerance according to the provided semantics after a failure. *At-most once semantics* does not guarantee data will be processed, only that the system will continue processing data after a failure. *At-least once* semantics guarantees all data is processed, but does not prevent possible duplicates. *Exactly-once* semantics is the only model that guarantees transparent fault tolerance, i.e. all data is processed exactly once despite failures.

There are different techniques to provide fault tolerance. One straightforward approach is to provide fault tolerance through *redundant computation*, also called *active replication* [GS97]. This consists of having N systems performing the same computation over the same data in parallel. As an example Fig. 2.6 shows this approach with $N = 2$. If one system fails, the system hands over control to any of the other replicas. Although simple and effective, this approach is usually infeasible in large-scale scenarios due to its costly operation: doubling the resources of a deployment is expensive. For this reason, redundant computation is usually used only for safety critical applications.

Instead, fault tolerance methods for data-parallel processing systems rely on reprocessing some data after failures. This introduces an overhead in the data processing path, but it does not require redundant computation, which makes it feasible for large scale systems.

Fault tolerance methods for data-parallel processing systems must also be designed for low runtime overhead. They usually follow two methods called *log data* and *log operation* [RG02]. *Log data* methods store intermediate results so that these can be recovered in the event of a failure. In contrast, *log operation* methods keep provenance information—i.e. the operations applied on the data—so that these can be replayed on the input data after a failure. Both methods have different trade-offs and are used in data-parallel processing systems.

We structure the discussion in this section along these two methods for fault tolerance. For *log data*, we describe systems that materialise intermediate results (§2.3.1), and systems that perform checkpoints (§2.3.2). For *log operation*, we describe lineage-based fault tolerance (§2.3.3).

2.3.1 Log Data: Materialising Intermediate Results

Systems such as MapReduce [DG04], Dryad [IBY$^+$07] or Samza [Apac] materialise results after computation. MapReduce is designed to operate on a distributed file system (such as HDFS) from where it reads the input data. Once map tasks finish processing results, they materialise the output results to their local disks before the scheduler schedules the reduce tasks. This means that throughput suffers as reducers cannot start processing output data until it has been correctly materialised to disk.

In general, materialising results has a large cost in terms of performance, as results must be stored in reliable storage, typically hard disks, which have restricted write bandwidth. For this reason, data-parallel processing systems often deploy RAID disks configurations to reduce the performance impact of the strategy.

How do these systems react to a failure?

After a crash failure, a central component—for example, a scheduler in the case of scheduled systems—re-schedules the failed task on a new node, where it starts reading its input data again. Instead of recomputing everything, the newly re-scheduled task can read the materialised results of the previous scheduled task. The central scheduler controls where all data is stored, and therefore coordinates this process.

In large deployments with frequent machine failures, it is advantageous to materialise results—in spite of the impact on performance—as this avoids recomputing all data from scratch. Note that the output of a long-running batch job can be connected to the input of a downstream job, further complicating the structure of long-running applications, which makes recomputation even more expensive.

2.3.2 Log Data: Independent Checkpoints

Similar to the materialisation of intermediate results, checkpointing backs up computed results to some form of storage from where they can be recovered after a failure [ZSK04]. The main difference between both approaches is that, while materialisation is a sequential process that occurs on the critical data processing path, checkpointing occurs asynchronously with data processing. One benefit of checkpointing is that, due to its asynchronous nature, the overhead on the runtime is typically smaller than in the case of materialisation. However, its implementation and operation are more complex, so there is normally a design tension between both approaches.

During normal operation, a data processing system consumes data that may update some state as part of the processing. Independently, a *checkpoint* process takes *snapshots* of the state and performs a *backup* to storage, from where it can be recovered in case the node fails.

According to how often checkpointing happens, it can be classified as follows:

Active replication. The data processing on the critical path is replicated to a parallel system that performs exactly the same computation [GS97]. This means that at least two times the resources (in some cases, it could be replicated x times) are used to compute the results. After detecting a failure, the system hands over the processing to the other active system. In this case, there is always a backup of the data that is continuously maintained by replicating the processing. As mentioned earlier, active replication is typically not used in data-parallel processing systems due to its high resource cost.

Passive replication. In passive replication, a checkpoint process takes snapshots of the state, and they are backed up periodically to persistent storage or maintained in memory on a different machine [Bir85]. The checkpointing process occurs concurrently with data processing. For this reason, it is necessary to enforce that snapshots are taken atomically—without concurrent modifications of the state.

In addition to maintaining a backup of stored snapshot, it is necessary to track "in-flight" data when implementing a passive replication approach. In-flight data are updates to state and produced results that are not part of any checkpoint yet, i.e. the updates that arrive between checkpoints. In the event of a failure, this in-flight data must be either discarded completely—so that all computation restarts from the last checkpoint—or accurately tracked so that it can be incorporated in the latest checkpoint, while keeping correct results without data lost.

Due to the need to track in-flight data, a checkpointing mechanism may follow a stop-the-world approach [MMI+13] where all output buffers of a processing node are flushed before a snapshot of the node state is taken. This approach increases processing latency, but it enforces that no in-flight data exist that can complicate recovering a failed process.

Perhaps the most well-known and widely implemented protocol to take coordinated and correct checkpoints is Chandy-Lamport [CL85] (also called the *snapshot algorithm*) that is implemented by systems such as Piccolo [PL10] or Flink [Apaa].

How do these systems recover after a failure?

When a failure occurs, the failed process is instantiated elsewhere by the recovery mechanism. After initialisation, it reads the latest available checkpoint and handles the potential in-flight data as necessary. For example, it can first reconstruct the state from the checkpoint, and then request replay of in-flight data, i.e. any data that was produced after the last checkpoint was taken. It is also part of the recovery mechanism to coordinate the communication with the failed node—which must stop during the recovery process.

2.3.3 Log Operation: Lineage-Based Failure Recovery

The main approach of lineage-based recovery in the context of data-parallel processing systems is implemented by Spark [$\text{ZCD}^{+}12$], through its Resilient Distributed Dataset (RDD) abstraction. RDDs keep lineage information, i.e. the operations that were necessary to compute the RDD. Maintaining the lineage has no impact on performance in the case of coarse-grained operations because it only requires adding the operations to the lineage graph—however, this model becomes expensive for fine-grained updates.

After a failure, the system discovers the failed RDDs and checks their lineage graphs to start the re-computation. At this point, the recovery process tracks which previous RDDs are available, if any, so that it can apply the smallest number of operations to recover the failed RDDs.

The aspect that makes this process feasible for Spark, in terms of both runtime performance and fast recovery, is the fact that re-computation can happen in parallel—it is not necessary to recover RDDs to the same node that failed. Instead, once the system knows the operations that must be reapplied, it can do so on partitioned chunks of the input data.

2.3.4 Discussion

The choice of a fault tolerance model depends on two main factors: (i) runtime overhead, which refers to how much of an impact a fault tolerance mechanism has on the normally running system; and (ii) recovery speed, which refers to how fast the system can get back to normal operation after a failure. Typically choosing between these two factors requires making a trade-off. Logging data has a higher runtime overhead than logging operations but provides better recovery times when small input data leads to much output data. Logging operations has a small runtime overhead, but the complexity is higher, and it is more difficult to achieve short recovery times. One particular situation that is beneficial for logging operation approaches is when operations are coarse-grained because keeping lineage is inexpensive.

A recent paper presents a detailed study of the trade-offs of both approaches [ZTKL14] in the context of Silo, a multi-core in-memory database system. Such analysis seems necessary in order to understand the best technique for the problem at hand.

Among systems that perform data logging, there is also the decision of whether to materialise results or checkpoint state. Data-parallel processing systems today are, as discussed before, stateless, and are designed in general for batch-oriented computation, which typically takes longer to complete. For these reasons, systems typically employ a strategy of materialising intermediate results, as the additional overhead of the operation is amortised by the long running tasks.

Overall, there is a large design space from which to select a fault tolerance approach, which is typically determined by the driving design decisions made on the target system.

2.4 Data-Parallel Processing Languages

The ideal programming language would permit developers to write algorithms in the most natural form and not worry about runtime considerations, such as parallelisation or fault tolerance. Unfortunately, automatic parallelisation [VRDB10] is a hard problem and there is no approach to date that fully achieves this goal.

Instead, programming languages for data processing systems make a trade-off: they choose between expressiveness and ease of use. It is possible to place existing programming languages on a spectrum in which, at one extreme, are languages that constrain the use of state but capture opportunities for data parallelism—these are the stateless programming languages of data-parallel processing systems; on the other end of the spectrum, are languages of distributed shared memory systems, in which algorithms make explicit use of state, but require more input from developers to achieve scalability and fault tolerance.

We do not provide a detailed description of all programming languages for data processing. Instead, we focus on both sides of the spectrum, with emphasis on the differences between stateless programming models of data-parallel processing systems and stateful models of distributed programming models.

2.4.1 Stateless Programming Models

Stateless dataflow graphs expose functional and declarative interfaces. Developers can use higher-order functions that the system can partition [BEH+10]. This permits to write concise code and yet it can leverage data-parallelism without explicit intervention by developers.

SQL-Based Programming Models

Relational databases—which are by far the most widely used data processing systems—have relied for decades on the expressiveness and ease of use of SQL, a declarative language that permits developers to describe the desired results, without expressing the transformations on data.

The last decade has witnessed several efforts to offer SQL-like interfaces on top of stateless dataflow models. For example, Hive [HCG+14] is one of the first approaches for writing SQL code that is translated into MapReduce programs, therefore exploiting the scalability and fault tolerance properties of the paradigm. Shark [XRZ+13] and Spark SQL [AXL+15] do the same on top of Spark, and Impala [KBB+15] provides its own stateless data-parallel processing system.

Hadapt [BPASP11] brings the expertise of relational query optimisation to the data-parallel processing domain. HAWQ [CWM+14], Redshift [GAT+15] are further examples of similar approaches. HAWQ is similar to Hadapt, and the differences of both engines are mostly in language features. Redshift departs from the previous in that its target deployment infrastructure is cloud-hosted services, where users can connect and write their queries.

Imperative-Style Interfaces

In parallel to the previous SQL-based programming languages, another breed of models appeared that provide structures that resemble those found in imperative-style programming languages. Pig Latin [ORS+08] is a domain specific language that introduces special operators that can be written as part of statements. A Pig program consists of a series of statements that are executed in order and permit developers to write ad-hoc queries, which are then translated to the MapReduce platform. A major application of Pig is to write complex extract-transform-load (ETL) oriented programs. Another language with a similar goal to Pig is Sawzall [Gri08].

An approach that takes a different direction is FlumeJava [CRP$^+$10]. It offers special collections—included as a library of the Java programming language—that gives developers the illusion of manipulating state when in fact this is translated into stateless MapReduce jobs.

A different approach is the proposed by CIEL [MSS$^+$11] with its Skywriting language. In Skywriting, users indicate the engine to execute tasks synchronously or asynchronously, within the dataflow graph of operators. This flexibility of controlling task creation, yet relying on data-driven scheduling provides is in the middle of the aforementioned spectrum.

2.4.2 Distributed Programming Models (DPM)

Unlike data-parallel processing systems that choose to provide an easy-to-use language at the expense of constraint expressiveness, DPM takes the opposite approach. They allow access to state, which is typically represented through a distributed shared memory abstraction, and permit fine-grained imperative operations.

The drawback of these models is that they also require users to indicate which parts of the code are meant to execute in parallel. In some cases, users must also be explicit about communication patterns, e.g. indicate when a shuffle is necessary. Advances in DPM are towards removing some responsibilities from the users, and we discuss relevant examples.

Piccolo [PL10] offers an imperative programming model in which users can write stateful programs. Piccolo's main abstraction is a key/value store that realises a distributed shared memory model. Unlike data-parallel processing systems, Piccolo requires developers to specify the tasks that form the computation, and it is not clear how a developer can write a pipeline.

Recently there are languages that embed distributed programming abstractions to facilitate scalable processing. Chapel [CCZ07] provides the expressiveness of DSM systems without requiring users to provide explicit hints about which parts of the code can be parallelised safely. X10 [CGS+05] is another language that implements the PGAS abstraction: developers split their computation into *spaces* where different data and functions live, to improve access locality.

2.5 Summary

We surveyed techniques to achieve scalability and fault tolerance as well as the programming models exposed by two families of Big Data processing systems: stateless data-parallel processing systems and distributed programming models.

Data and task parallelism are the two main methods to build scalable systems. The first is the preferred choice of data processing systems, while the second is typically offered in distributed programming models. We discussed techniques for scalable processing in both scheduled systems—optimised for batch-oriented workloads—and materialised systems, that are the preferred choice for streaming workloads.

Logging data and logging operations are the two main strategies for achieving fault tolerance in Big Data processing. Data-parallel processing systems usually materialise results; distributed programming models—that permit the modification of state—offer checkpoint-based solutions instead.

Programming languages for data processing form a spectrum between: (i) ease of use but constrained programming model; and (ii) general programming models but with the need to be explicit about parallelisation opportunities.

Distributed programming models and stateless programming models provide different tradeoffs to developers. Writing stateful algorithms is easier with distributed programming models than with stateless data-parallel models because they permit access to mutable state. Despite this, they are not the preferred choice for Big Data analysis because they require developers to choose the right task parallelism level and manually handle fault tolerance. In contrast, data-parallel processing systems capture parallelism and provide fault tolerance automatically, but they do not permit the high-performing execution of stateful programs.

Chapter 3

Stateful Dataflow Graphs

This thesis describes a stateful data-parallel processing model, where access to state is explicit, which has two implications: (i) writing stateful algorithms becomes easier as it is possible to maintain state; and (ii) it is possible to execute stateful programs, even when updates are fine-grained, with high throughput. Stateful data-parallel processing requires a new abstraction called **Stateful Dataflow Graphs** (SDGs), which is the subject of this chapter.

The goal of a stateful dataflow graph is to represent a stateful algorithm in a dataflow graph by introducing state as a new element in the dataflow (see Fig. 3.1). The SDG must retain the scalability properties of modern stateless dataflow models by permitting data-parallel processing. In order to do this in the presence of state, SDGs introduce two distributed mutable state abstractions that define how state is manipulated.

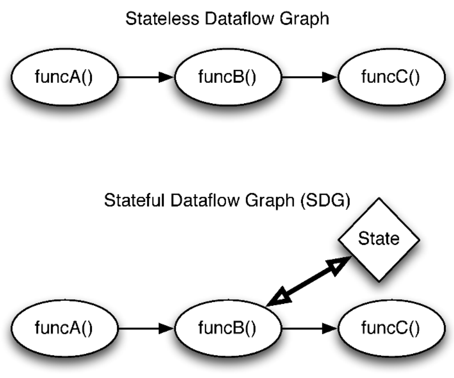

Figure 3.1: An SDG in comparison to traditional stateless dataflow graphs.

In this chapter, we present the SDG model and we explain in detail the distributed mutable abstraction. The next chapter (§4) focuses on the particular techniques used to achieve scalability and fault tolerance, and we then present the translation of imperative programs in §5.

We first provide intuition for Stateful Dataflow Graphs and introduce a running example that is used throughout the chapter (§3.1). We then define SDGs more formally and focus on their components, that is, task elements that perform computation (§3.2) and state elements that embody the mutable state (§3.3). Although the primary focus of this chapter is the SDG model, we conclude with a discussion of some runtime considerations that affect the deployment of SDGs and are important for the subsequent chapters (§3.4).

3.1 The Stateful Dataflow Graph Model

A stateful dataflow graph is formed of three different entities. First, it has *task elements* (TE), that express the computation. These are represented as ovals in Fig. 3.1. TEs receive data through input dataflows that carry streams—the edges in the dataflows in the figure—which are, informally, collections of data. TEs process the data and produce output data, which is, in turn, sent downstream through output dataflows. SDGs also have *state elements* (SEs), which represent the state of the computation. An SE is represented in the bottom dataflow of Fig. 3.1 as a square. State access is represented in SDGs through access edges (see Fig. 3.1). When a TE has access to an SE, it can access it after receiving input data. For example, it can produce some data used to update the state, or use both input data and state to produce output data, etc.

When reading data from input dataflows, a TE can choose with which granularity to read data. For example, it can read one tuple at a time or a batch of tuples at a time. It must be possible to order data so that, at a given point in time, there is a notion of how much data has been consumed and how much data has been produced. This is necessary for fault tolerance, as explained in the next chapter.

Next we formally define an SDG and introduce an example that we use along this chapter to explain the remaining concepts.

```
 1 Matrix userItem = new Matrix();
 2 Matrix coOcc = new Matrix();
 3
 4 void addRating(int user, int item, int rating) {
 5   userItem.setElement(user, item, rating);
 6   Vector userRow = userItem.getRow(user);
 7   for (int i = 0; i < userRow.size(); i++)
 8     if (userRow.get(i) > 0) {
 9       int count = coOcc.getElement(item, i);
10       coOcc.setElement(item, i, count + 1);
11       coOcc.setElement(i, item, count + 1);
12     }
13 }
14
15 Vector getRecommendation(int user) {
16   Vector userRow = userItem.getRow(user);
17   Vector rec = coOcc.multiply(userRow);
18   return rec;
19 }
```

Algorithm 1: Collaborative filtering in Java.

Formal definition of SDG

An SDG is a graph that has two types of vertexes: *task elements*, $o \in O$, transform input to output dataflows; and *state elements*, $s \in S$, represent the state in the SDG.

Access edges, $a = (o, s) \in A$, connect task elements to the state elements that they read or update. To facilitate the allocation of task and state elements to nodes, each task element can only access a single state element, i.e. A is a partial function: $(o_i, s_j) \in A$, $(o_i, s_k) \in A s_j = s_k$. Dataflows are edges between task elements, $d = (o_i, o_j) \in D$, and contain data items.

Example. To illustrate an SDG, we describe the example shown in Algorithm 1, which is a Java implementation of the collaborative filtering (CF) algorithm [SKKR01] (we omit the implementation of certain functions that are not relevant for the example). It outputs up-to-date recommendations of items to users (function `getRec` line 18) based on previous item ratings (function `addRating` line 7).

An item-by-item collaborative filtering algorithm relies on information about the ratings that a population of users have given to a set of items. This information is typically represented as a matrix, which we call *userItem*. The intuition behind the recommendation algorithm is that similar users are more likely to be interested in similar items. For example, if user A has rated action movies highly, and B has done the same, a new item that is rated highly by user A is also to be of interest to B, and thus the algorithm offers it as a recommendation.

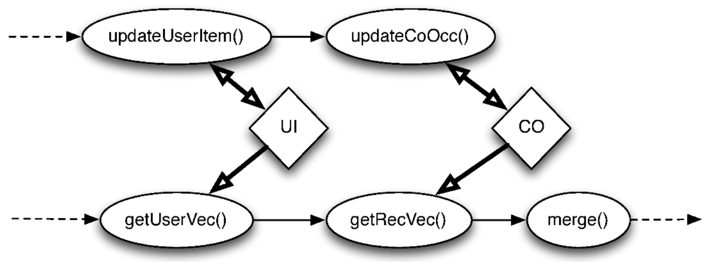

Figure 3.2: Example of SDG for the CF algorithm.

To capture the two-fold relationship between users and items, the algorithm relies on a *cooccurrence* matrix that results from multiplying *userItem* by itself. A subsequent multiplication of *userItem* by *cooccurrence* results in the recommendation matrix, where items with higher value per row, i.e. per user, are the ones that should be recommended.

We implement this algorithm in an online fashion, i.e. it maintains an updated cooccurrence matrix and only computes the recommendation vector for the user to whom it offers a recommendation.

Note that one important characteristic of this algorithm is that the task of keeping *cooccurrence* up-to-date and the task of providing recommendations to users have different performance goals. Maintaining the *cooccurrence* matrix updated requires high-throughput. Providing recommendations to users demands low-latency for good user experience, for example, when recommendations are included in dynamically generated web pages.

Fig. 3.2 shows the SDG for this algorithm. It consists of five TEs: the `updateUserItem` and `updateCoOcc`. TEs realise the `addRating` function from the algorithm; and the `getUserVec`, `getRecVec` and `merge` TEs implement the `getRec` function.

The algorithm maintains state in two data structures: the matrix `userItem` stores the ratings of items made by users (line 4); the cooccurrence matrix `coOcc` records correlations between items that were rated together by multiple users (line 5). Using these two data structures permit a concise representation of the algorithm: it is only necessary to consider the functions that read and update the structures to achieve a given goal.

The function `addRating` first adds a new rating to `userItem` (line 8). It then incrementally updates `coOcc` by increasing the cooccurrence counts for the newly-rated item and existing items with non-zero ratings (lines 10-15).

The function `getRecommendation` takes the rating vector of a user, `userRow` (line 19), and multiplies it by the cooccurrence matrix to obtain a recommendation vector `userRec` (line 20).

Note that both functions require `userItem` and `coOcc` to be mutable, with efficient fine-grained access. If this algorithm were to be redesigned using a stateless dataflow model, all state would be represented and forwarded as data in the dataflow graph. For fresh results—results that include the latest information available—this should happen every time the state changes, i.e. every time a new update occurs, therefore leading to inefficient processing, specially when state grows large and fault tolerance mechanisms require materialising intermediate results.

3.2 Task Elements (TE)

In this section, we define task elements. First, through a generic data model, we explain how TEs process data, and then we explain the relationship between SEs and data.

Data model. A stream is a possibly infinite series of tuples. A tuple has a key, a payload, and a timestamp that is assigned by a monotonically increasing logical clock. When the clock is in a logical TE, the timestamp expresses *system time*. When it is assigned by external sources it expresses *application time*. Tuples in a stream are ordered according to their timestamps. This is an important requirement, as TEs depend on it to track how much data they have processed at a given point in time. Keys are not unique, and they are used to partition the stream. They can be computed, for example, as a hash based on the payload. In the case of the collaborative filtering algorithm, for example, the key can be the `uid` as illustrated in Fig. 3.3.

TE model. Tuples are processed by TEs. A TE o takes n input streams, processes tuples in the input streams and produces one or more output streams. For ease-of-presentation, we assume that a TE emits only a single output stream (unless the downstream TE is partitioned).

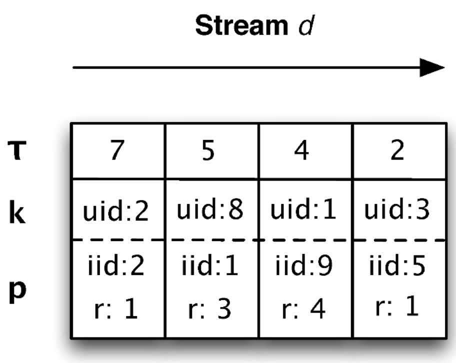

Figure 3.3: Data stream of tuples. Tuples are ordered according to a timestamp and contain a key and a payload.

Note that a TE can process data in streams in a stream-oriented fashion—one tuple at a time—or it can process data in a batch-oriented fashion, processing batches of tuples delimited by either timestamps or size. The first mode yields lower latency and the second higher throughput. The trade-off between latency and throughput depends on the particular computation that the TE performs.

A TE function f_o defines the processing of TE o on input tuples from the streams. At each invocation of f_o, the TE accepts a finite set of non-processed tuples from the streams. After processing, the TE advances to new positions in the input streams and updates a vector of timestamps of size n—same as the number of input streams—that keeps track of the last tuple processed per stream.

Stateful TEs. TEs can be stateless, such as in the case of a filter or a map operation, or stateful, as it is the case for a join as an aggregate, where state is internal to the operators. In other cases, the state is external, for example in the TEs that access `userItem` and `coOcc` in the collaborative filtering example. In other cases, state consists of a partial view of the data.

More formally a stateful TE has access to a SE s_o, which can be updated after processing. We assume that TEs are deterministic and do not have other externally visible side-effects. There is a vector of timestamps associated to each SE to specify the last tuple from each stream that affected the state. This vector of timestamps must be smaller or equal than the one used to keep track of last processed tuples. A stateless TE does not have any SE.

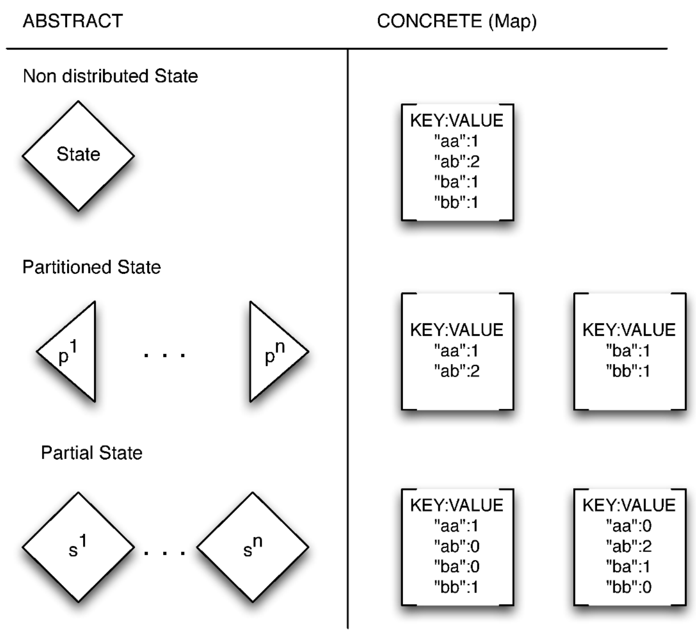

Figure 3.4: Different distributed state abstractions.

3.3 State Elements (SEs)

Explicit state in a dataflow graph has two consequences: it increases the expressiveness of the model and it permits high performance execution of stateful algorithms, even when updates to the state occur in a fine-grained fashion, e.g. when every tuple updates the state. For scalability, it is possible to scale out TEs, as other systems have done previously (e.g. MapReduce, Spark or parallel database systems). In order for this strategy to be effective, it is necessary to handle state to avoid remote state accesses, which could limit the scalability of SDGs.

It is possible to scale state in different ways, each with its own advantages and disadvantages. In stateful data-parallel processing, the SEs are responsible for encapsulating distributed mutable state in a way that enables data-parallelism. If the state becomes too large to fit in the memory of a machine, it must be distributed across multiple nodes allowing to aggregate their memory. Distributing an SE s_i leads to multiple SE instances $s_{i,j}$. In a typical scenario, SDGs distribute SEs as many times as TEs, so that each TE can access an instances of the SE locally.

The distribution of SEs is constrained by the algorithm semantics. We propose two abstractions for algorithm state, which are shown in Fig. 3.4: a *partitioned* SE splits its internal data structure into disjoint partitions. This can be used when algorithms access state only by key—such as in the case of the `userItem` matrix in the collaborative filtering algorithm—and leads to an efficient distribution. When this is not possible, for example, because an algorithm accesses the SE randomly, a *partial* SE replicates its data structure, creating multiple copies that are updated independently.

The collaborative filtering example from Fig. 3.2 illustrates these two cases. First, the `userItem` SE may grow larger than the main memory of a single node, for example, when the user population or the available items grow. Second, the data-parallel execution of the CPU-intensive `updateCoOcc` TE leads to multiple instances, each requiring local access to the `coOcc` SE for scalable processing.

3.3.1 Partitioned State

For algorithms for which state can be partitioned, SEs can be split and their instances placed on separate nodes. In this case, access to the SE instances occurs in parallel.

Developers can use regular data structures for SEs (e.g. vector, hash tables, matrices) or define their own. Different data structures support different partitioning strategies: e.g. a map can be hash- or range-partitioned; a matrix can be partitioned by row or column. To obtain a unique partitioning, which is necessary to partition the SEs in a consistent way, TEs cannot access partitioned SEs using conflicting strategies. We define a conflicting strategy as one that changes the key during subsequent accesses to state. An example would be an algorithm that accesses a matrix by both row and column.

In addition, the dataflow partitioning strategy must be compatible with the data access patterns by the TEs. For example, multiple TE instances with an access edge to a partitioned SE must use the same partitioning key on the dataflow. This is necessary so that the TEs can access SE instances locally. For example, in the collaborative filtering algorithm, the `userItem` SE and the `newRating` and `rec` request dataflows must all be partitioned by row, i.e. the users for which ratings are maintained. If an algorithm requires access from different TEs with different partitioning strategies, it is possible to simply instantiate two *logically* different SEs.

Partitioned SEs permit to achieve high throughput as all SEs can be accessed concurrently. The collaborative filtering example illustrates a clear use case: when the `userItem` matrix grows, it can be partitioned across nodes based on user identifier (uid) as an access key. In this way, multiple concurrent requests can access state in parallel.

The assumption under which our partitioning strategy for SEs operates is that TEs will access one element of the SE at a time, and that there are no issues arising due to concurrent accesses from several TEs. For example, when two different TEs access the same SE, different orderings of such accesses happen. The consequence of different orderings for the type of analytical algorithms—the target of data-parallel processing systems—is that we may get a result that is slightly stale. This situation, however is recovered as soon as the next update occurs. In other words, we target algorithms for which different orderings do not modify results, or in case there is modification, results are still correct.

Applications that present such concurrency issues are not the target of data-parallel processing systems. Databases with transactional support are a better fit for these type of problems.

3.3.2 Partial State

In some cases, the data structure of an SE cannot be partitioned because the access patterns of TEs are arbitrary. For example, in the collaborative filtering algorithm, the `coOcc` matrix has an access pattern in which the `updateCoOcc` TE may update any row or column. In this case, an SE is distributed by creating multiple partial SE instances, each containing the entire data structure. Partial SE instances can be updated independently by different TE instances, which means that there are situations in which different SE instances store different data.

When a TE accesses a partial SE, there are two possible types of accesses based on the semantics of the algorithm: a TE instance may access (i) the local SE instance on the same node; or (ii) the global state by accessing all of the partial SE instances, which introduces a synchronisation point (see Fig. 3.5).

When accessing all partial SE instances, it is possible to execute computation that merges their values, thus reconciling the differences between them. This is necessary when, for

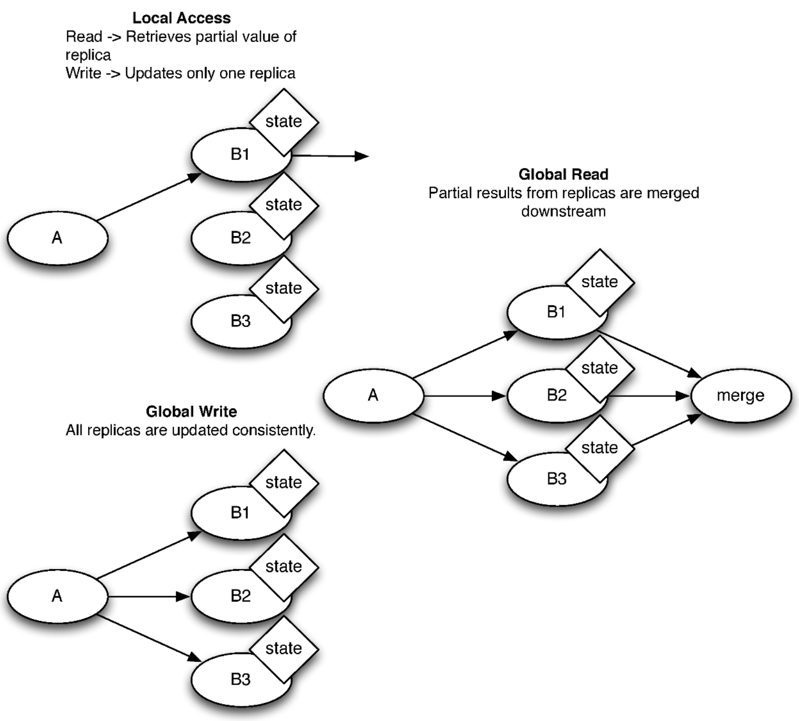
Local Access
Read -> Retrieves partial value of replica
Write -> Updates only one replica
Global Read
Partial results from replicas are merged downstream
Global Write
All replicas are updated consistently.
A
B1
B2
B3
state
merge

Figure 3.5: Types of partial state accesses.

example, the application is reading data from partial instances: users do not expect a collection of partial results, but a unique one. Such view of the results is achieved by a merge TE that computes a single global value from partial SE instances. Merge computation is application-specific and must be defined by the user. In the collaborative filtering algorithm, the `merge` function takes all partial `userRec` vectors and computes a single recommendation vector.

In the collaborative filtering example, the `coOc` matrix requires arbitrary access patterns. It cannot be partitioned but only replicated on multiple nodes in order to partition updates. In this case, results from a single instance of `coOcc` are *partial*, and must be *merged* with other partial results to obtain a complete result. For example, when multiplying the `userVec` by the `coOcc` matrix, this must be performed on all instances of `coOcc`, leading to multiple partial recommendation vectors. These partial vectors must be merged to obtain the final recommendation vector `rec` for the user (line 21).

Merging occurs in SDGs by introducing a TE that receives all the partial results from the upstream TEs and applies a *merge* function that reduces those partial results to a single one, which is then the output of the *merge* TE. Users are required to provide the *merge* function. Writing one is trivial when the previous TEs perform operations that are commutative and associative. In other cases, even when operations do not comply with those properties, it is still possible to merge results in meaningful ways, such as in the case of merging machine learning models that were trained with an stochastic method.

3.4 Execution of Stateful Dataflow Graphs

The SDG model is general and can be realised as part of a real system in different ways. We explain some of the choices that we have made when implementing SDGs as part of SEEP, our stateful data-parallel processing prototype. We relate these decisions to the dimensions presented in §2. Subsequent chapters explain some of these decisions in more detail.

Scheduled vs materialised tasks. Data-parallel processing systems have two main modes of execution, they can schedule or materialise tasks, as introduced in §2. In the context of SDG, these tasks are the TEs of our dataflow model. We make the decision of materialising the entire SDG, instead of scheduling individual TEs. One consequence of this decision is that it is unnecessary to generate the complete output data of a TE before it is processed by the next TE. Data tuples are processed with low latency, even across a sequence of TEs, without any scheduling overhead, and fewer tuples are processed during failure recovery.

Degree of parallelism. For data-parallel processing, a TE can be instantiated multiple times to handle parts of a dataflow, resulting in multiple TE instances. We use the distributed mutable state abstraction to maintain the local access to SEs in the case of stateful TEs. The number of instances for a given TE can be chosen a priori, or adapted dynamically as the workload changes or stragglers occur. We provide more details about dynamic parallelism, or *elasticity* in the next chapter.

Iteration. In iterative algorithms, SEs are accessed multiple times by TEs. There are two cases to be distinguished: (i) if the repeated access is from a single TE, the iteration is entirely local and can be supported efficiently by a single node; and (ii) if the iteration involves multiple pipelined TEs, a cycle in the dataflow of the SDG can propagate updates between TEs.

With cycles in the dataflow, SDGs do not provide coordination during iteration by default, i.e. TEs in the cycle process data as soon as it is available. This is sufficient for iterative machine learning and data mining algorithms that can converge from different intermediate states even without explicit coordination, such as those based on stochastic training. A strong consistency model for SDGs could be realised with per-loop timestamps, as used by Naiad [MMI+13] with its implementation of a *timely* dataflow.

3.5 Summary

This chapter presented SDGs, which are a new dataflow model that enables stateful data-parallel processing. By explicitly including state in a dataflow graph of task elements, it is possible to represent a broad number of stateful algorithms.

Stateful Dataflow Graphs and the distributed mutable state abstractions—*partitioned* and *partial*—permit efficient distribution of state. This allows to achieve scalability, even when updates to state occur in a fine-grained fashion. These properties are key to translate imperative programs written in Java to SDGs, a process that we describe in Chapter §5.

Chapter 4

Scalability and Fault Tolerance of Stateful Dataflow Graphs

In the previous chapter, we introduced Stateful Dataflow Graphs that represent state explicitly in the dataflow model, allowing it to realise stateful data-parallel processing. Different distributed mutable state abstractions enable data-parallelism by defining how state can be distributed across a cluster of machines.

This chapter focuses on the system aspects when making the SDG abstraction scalable and fault tolerant. Achieving scalability and fault tolerance—the defining properties of stateless processing systems—is more challenging in a stateful data-parallel processing system. This chapter focuses on the system aspects of making SDGs scalable and fault tolerant.

- **Data-parallel processing.** A data-parallel implementation of SDGs must scale when more resources are provided, even in the present of state. To do so, it is necessary to introduce new techniques and protocols that manage the state maintained by the system.
- **Elasticity.** A modern data-parallel processing systems must be *elastic*. The system must support dynamic provisioning of new resources to increase its processing throughput, for example, when addressing sudden spikes in the workload. One important consequence of an elastic system is its cost-effectiveness when deployed

on a public cloud. By acquiring resources on demand, a data-parallel processing system can reduce costs in public cloud environments, such as Amazon EC2 [Amaa] and Rackspace [Rac].

- **Resource-efficient failure recovery.** The system must be fault-tolerant, so that it can recover in the presence of failures. Fault tolerance cannot impact performance. For this reason, a fault tolerant solution must incur low runtime overhead. Additionally, due to the presence of streaming workloads where data arrives continuously, failure recovery should be quick to prevent losing data arriving during the time the system is down.

In this chapter, we introduce the mechanisms to realise stateful data-parallel processing through SDGs. The main principle of a system that implements stateful data-parallel processing is to represent state explicitly as a first-class citizen in the system §4.1. We then introduce an integrated approach to achieve scalability and fault tolerance in §4.2, explaining the details of both scalable data-parallel processing §4.2.2 and fault tolerance §4.2.3. Finally, we present SEEP [CFMKP13], a stateful data-parallel processing prototype that supports the SDG model and implements the techniques described in this chapter. We conclude the chapter with some practical considerations that influence the previous mechanisms when deploying a system in a real public cloud scenario §4.3.

4.1 State as a First-Class Citizen

When designing a data-parallel system that supports techniques that deal with state, such as scaling out or recovering state, we make the decision of treating state as an abstract entity that the system is aware of and hence can manage. In particular, we: (i) make SEs externally visible to the system; and (ii) define primitives for the system to manage state in a generic fashion. Based on these primitives, we define more complex operations such as scale out and failure recovery.

Treating state as a first-class citizen is unusual in the context of data processing systems because they typically avoid exposing state. For example, stream processing systems use state internally for some TE implementations, e.g. an aggregate function that maintains

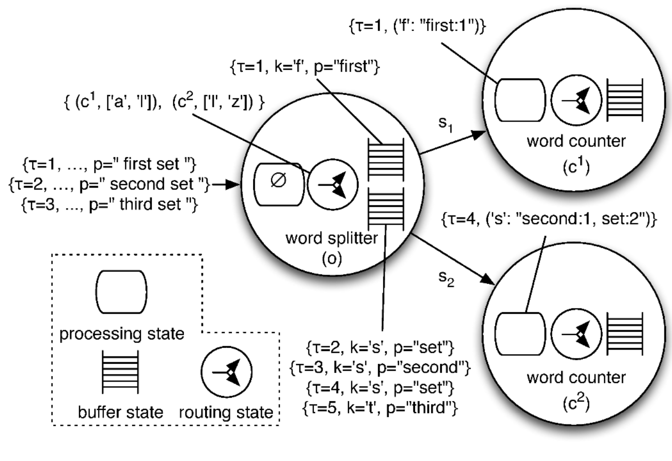

Figure 4.1: State is formed by processing state (SE), routing state and buffer state.

a partial aggregation value while scanning the input data. Often state is only managed for specific purposes, such as persistence or overload handling. For example state is spilled to disk when a node is about to run out of memory [LZR06]. Instead, we aim to define a set of primitives that permit a more general treatment of state. We should be able to implement different scale-out algorithms with these techniques if necessary.

There are several reasons to make state a first-class citizen. First, it enables the system to recover stateful TEs more efficiently after failure. Instead of re-processing all data necessary to recreate the SE, the system can restore the state directly from an SE checkpoint, as we explain later in this chapter. Second, it allows the system to distribute the SEs across a set of new instantiated TEs to support scaling out. Such distribution can occur according to one of the supported distributed mutable abstractions, partitioned or partial.

In the previous chapter, we defined state as the SEs in a stateful dataflow graph. Here we generalise our understanding of state, and the state—in terms of what the system must manage—consists of three different types: processing state—which is equivalent to SEs—buffer state and routing state, as shown in Fig. 4.1. The figure shows a dataflow with three TEs for collecting word frequencies in a text stream every minute: a stateless word split TE o tokenises a stream of strings into words, and two partitioned stateful word count TEs c_1 and c_2 maintain a windowed frequency count of words. We explain the three types of state below and use the query as a running example.

Processing state. Input and output data of TEs is formed by tuples. The state of stateful

TEs depends on the input tuples and potentially the history of past tuples. TEs typically maintain an internal summary of this history of input tuples, which is the SE with the processing state. The current processing state θ_o is computed from all past processed tuples.

We define the processing state of a TE o as a set of key/value pairs, $\theta_o = (k_1, v_1), ..., (k_n, v_n)$, where keys are unique. The value v stores the portion of processing state that the TE requires when processing tuples with a given key. In addition, the processing state is associated with a vector of timestamps that maintains the last tuple from each input stream that is reflected in θ_o.

In Fig. 4.1, we give an example of processing state for the word frequency TEs. To simplify presentation, keys are assumed to be the first letter of a word. The upstream word split TE sends the word "first" to the word count TE c_1 at timestamp $\tau = 1$, resulting in the processing state $\theta_{c1} = ('f','first:1')$ and timestamp $\tau_{c1} = (1)$. The words "set", "second" and "set" are processed by c_2, instead, which at $\tau_{c2} = (4)$ holds processing state $\theta_{c2} = ('s','second:1,set:2')$.

A TE can maintain state using efficient data structures internally and only translate it to key/value pairs when requested by system. To expose its processing state, the developer of a TE o implements a function `get-processing-state(o)` $\rightarrow (\theta_o, \tau_o)$. This function is invoked by the system and takes a snapshot of the state and records the timestamp τ_o of the most recent tuples that affected the state. Recording this timestamp is useful to understand how much input data is represented already as part of this state, which is used by fault tolerance, as explained later in the chapter.

Buffer state. A system typically interposes output buffers between TEs that are placed in different physical nodes, which are used to buffer tuples before sending them to downstream TEs (see Fig. 4.1). Buffers compensate for transient fluctuations of stream rates and network capacity. They are also used to batch data for more efficient network transfers.

Tuples in output buffers contribute to the state that the system must manage: (i) output buffers store tuples that have not yet been processed by downstream TEs and therefore must be re-processed after failure; (ii) after dynamic TE scale out, tuples in output buffers

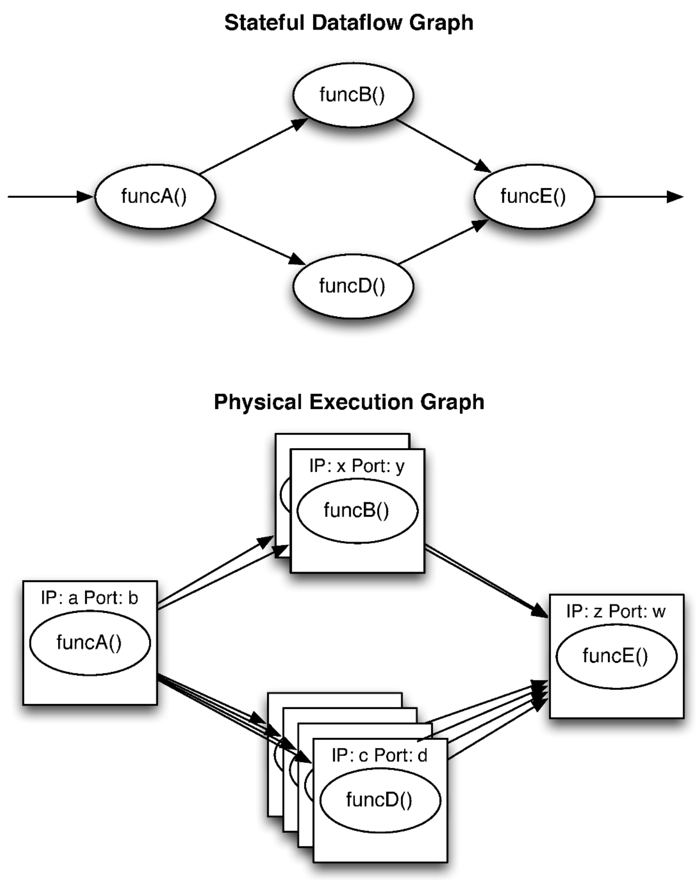

Figure 4.2: Stateful dataflow graph and execution graph.

must be dispatched to the correct partitioned downstream TE.

We model the output buffer of a TE o to a downstream TE d as having buffer state $\beta_o = (d_1, t_1, \ldots), \ldots$ with $t_1 \in (o, d_1)$. It stores a finite number of past output tuples. The notation $\beta_o(d_i)$ refers to the output tuples for a downstream TE d_i.

Note that output buffers maintain what we refer to as *unprocessed* data. This data has been processed by some but not all of the TEs in the dataflow, so it is still needed to produce results. Efficient handling of buffer state is key to achieving the right fault tolerance semantics. The general intuition is that tuples in an output buffer are discarded after they are no longer needed for recovery. Removing tuples is necessary to reduce resource consumption. A TE trims tuples from an output buffer by removing tuples with timestamps older than τ when it executes the function `trim(o, τ)`.

Routing state. If TEs were scheduled, the scheduler would know about their placement. However, when TEs are materialised, as it is the case in our implementation of SDGs, the TEs must know where to route tuples downstream. A TE o in the SDG may correspond to multiple partitioned TEs $o_1, ..., o_\pi$ in the *execution graph*, which is the physical realisation of a SDG. Fig. 4.2 shows the difference between a *logical* SDG and a concrete *execution graph*. An upstream TE u has to decide to which TE o_i to route a tuple. In the case of partitioned SEs, the problem is more challenging as one desired property is to allow dynamic scale out—i.e. elasticity—of TEs. This means that the routing state that a TE is using to route tuples must be restored after a failure.

Formally, for a TE o, we define the routing state as $\rho_o = (d_1, [k_1, k_2]), ..., (d_\pi, [k_{\pi 1}, k_\pi])$, which maps the keys $k \in [k_i, k_j)$ to a partitioned downstream TE d_i. For example, the word split TE in Fig. 4.1 has $\rho_o = (c_1, ['a', 'l']), (c_2, ('l', 'z'])$. It sends words starting with letters up to 'l' to TE c_1 and with letters from 'l' to c_2.

4.2 Integrating Scale Out and Fault Tolerance

We define primitives to operate on the state. These primitives work as building blocks that can be composed together to create more complex protocols.

The main idea of our approach to provide scalability and fault tolerance to stateful data-parallel processing lies on the following observation: ***a TE failing and a TE scaling out—redistributed to more physical nodes—are equivalent from a state management perspective***. This observation is key for an efficient implementation of mechanisms that permit state management.

The integrated approach for fault tolerance and scalability is a protocol that defines the interactions between *worker* nodes, which host TEs and SEs of the SDG, and a *master* node, which coordinates the deployment and execution of applications. We define the protocol in a generic way based on the collection of state management primitives.

In this section we first introduce and discuss the different primitives for state management §4.2.1. We then explain how the integrated approach achieves scalability §4.2.2 and fault tolerance §4.2.3. We finish with some practical considerations §4.3 for elasticity and fault tolerance in stateful data-parallel processing systems.

4.2.1 Primitives for State Management

We introduce a set of state management primitives. We define a small set of primitives to cover our goals—i.e. elasticity and fault tolerance, which are: `checkpoint`, `backup`, `restore`, `partition` and `merge`, as explained below.

Checkpoint state. The system can obtain a representation of the processing state θ_o and the buffer state β_o of a TE o in the form of a checkpoint. Note that routing state is not included in the state checkpoint because it only changes in case of scale out or recovery and not during regular tuple processing. Instead, routing state is maintained by the logically centralised *master* node.

A checkpoint is taken by the function `checkpoint-state(o)` $\rightarrow (\theta_o, \tau_o, \beta_o)$. It obtains the processing state θ_o by calling the user-implemented function `get-processing-state()`, which also returns the timestamp τ_o of the most recent tuples in the streams from the upstream TEs that affected the state checkpoint. This permits the system to discard tuples with older timestamps, which are duplicates, during replay—a necessary condition to achieve exactly-once semantics.

The function `checkpoint-state` is executed asynchronously and triggered every checkpointing interval c, or after a user-defined event, e.g. when the state has changed significantly. A short checkpointing interval results in a smaller number of tuples that would be required to replay and reprocess to bring the processing state up-to-date, but it incurs a higher overhead.

In the case of TEs that perform some computation over streams of data, it is common to define *windows* that split the seemingly infinite stream into chunks of data that are given to the application. The checkpointing interval should be shorter than the window size of a TE when a window is defined. If checkpoints are taken more rarely, they contain processing state that is superseded by tuples that must be re-processed. To reduce the size of checkpoints, it is also possible to use incremental checkpointing techniques [HXCZ07], which checkpoint only changes to the state since the last checkpoint.

Backup state. The SE and buffer state, as returned by `checkpoint-state`, can be backed up to a node hosting a different TE in anticipation of a `restore` or `partition` operation.

```
TE identifier function: $id : \mathcal{O} \rightarrow \mathcal{N}$,
upstream TE of $o$: $up(o) = \{o_1, \dots, o_i, \dots, o_m\}$
previous backup TE: $backup(o) = o_j$ or $\perp$ if undef.
function backup-state($o$)
1     $(\theta_o, \vec{\tau}_o, \beta_o) \leftarrow$ checkpoint-state(o)
2     $i = hash(id(o)) \bmod |up(o)|$
3     store-backup($o_i, o, \theta_o, \vec{\tau}_o, \beta_o$)
4     for $u$ in $up(o)$ do trim($u, \vec{\tau}_{oj}$) : $s_j = (u, o)$
5     if $backup(o) \neq \perp \wedge backup(o) \neq o_i$ then
6         delete-backup($backup(o), o$)
      end
7     $backup(o) \leftarrow o_i$
end
function restore-state($o, \theta, \vec{\tau}, \beta, \rho$)
8     set-processing-state($o, \theta, \vec{\tau}$)
9     $\beta_o \leftarrow \beta, \rho_o \leftarrow \rho$
end
function replay-buffer-state($u, o$)
10    for $t$ in $\beta_u(o)$ do send o: t
end
```

Algorithm 2: SE backup and restore.

After the state was backed up, already processed tuples from output buffers in upstream TEs can be discarded because they are no longer required for failure recovery—as they are already incorporated in the state that is tolerant to failures.

Algorithm 2 defines function `backup-state(o)` for backing up the state of TE o. To backup the SE, the system first needs to create a checkpoint of the state, i.e. to externalise the SE. The checkpoint is created in line 1. Once the checkpoint is available, the system must choose a target node to store it. Since many stateful TEs can be part of the SDG and they all need to checkpoint and backup their state, it is important to choose the node that will host the checkpoint in a way that spreads the load evenly across the nodes. This is performed by using a hash function in line 2.

When a machine hosting a TE has many TEs that use it to keep their state backups, the backup operation incurs significant overhead. In that case, TEs should balance the backup load across all of the available nodes. This can be achieved by reusing the scale out approach that we introduce later.

The state is backed up in line 3. When a TE receives the checkpointed state from a downstream TE, it can trim the output buffer that points to the downstream, as this may contain data already reflected in the state. In particular, it is safe to trim all tuples in the output buffer that are older than the timestamp associated to the backup, which occurs in line 4.

When new nodes join the system to host TEs—such as in the case of a scale out operation—or when they leave the system after hosting a TE, such as after a failure or merge, the choice of `backup(o)` may change. This is, if TEs were to reapply the hash function of line 2, the TE chosen to host the SE backup will likely change. When such change happens line 5, the old backup TE is released (line 6).

Restore state. A backed up SE is restored to another node to recover a failed TE or to redistribute state across partitioned TEs. For example, when a stateful TE fails and loses its SEs, the TE that was hosting the failed SE must restore it to continue operation. The function restore-state(o,θ,τ,β,ρ), defined in Algorithm 2, restores the state of TE o. It then initialises the processing state using the function `set-processing-state` (line 8) and also assigns the buffer and routing states (line 9).

After the state was restored from a checkpoint, the function `replay-buffer-state(u,o)` is used to replay unprocessed tuples in the output buffer from an upstream TE u to bring the SE up-to-date. Before TE o emits new tuples, it resets its logical clock to the timestamp τ. This permits downstream TEs to detect duplicates—those tuples that are already part of the state according to the state—and discarding them.

Partition state. When a stateful TE scales out, its processing state must be distributed across the new nodes. It can be partitioned when the SE is of *partitioned* type or replicated when it is of the *partial* type. This is done by repartitioning the key space of the tuples processed by the TE or by copying them in the case of partial state. In addition, the routing state of its upstream TEs must be updated to account for the newly allocated TEs. Finally, the buffer state of the upstream TEs is partitioned to ensure that unprocessed tuples are dispatched to the correct downstream partitioned TE.

In Algorithm 3, we define the function partition-processing-state(o, π), which partitions the state of TE o for π new partitioned TEs $o_1,...,o_\pi$. The partitioning is performed from the state saved by `backup(o)` to allow partitioned TEs to recover in the case of failure. First, the key range processed by o, as specified by the routing state of the upstream TE u, is split into π intervals (lines 1-2). The key space can be distributed evenly using

```
1  key interval : $(k_l,k_h)$ : $(o,[k_l,k_h]) \in \rho_u \wedge u \in up(o)$
2  key split : $(k_1, \dots, k_\pi)$ : $k_l = k_1 < \dots < k_{\pi+1} = k_h$
   function partition-processing-state($o, \pi$)
3      $(\theta, \vec{\tau}, \beta) \leftarrow$ retrieve-backup($backup(o), o$)
4      for $i \leftarrow 1$ to $\pi$ do
5          $\theta_i \leftarrow \{(k,v) \in \theta_i : k_i \le k < k_{i+1}\}$
6          $\vec{\tau}_i \leftarrow \vec{\tau}$
7          if $i \neq 1$ then $\beta_i \leftarrow \emptyset$ else $\beta_1 \leftarrow \beta$
8          store-backup($backup(o^i), o^i, \theta_i, \vec{\tau}_i, \beta_i$)
       end
   end
   function partition-routing-state($u, o, \pi$)
9      $\rho_u \leftarrow \rho_u \setminus \{(o,[k_l,l_h])\}$
10     for $i \leftarrow 1$ $to$ $\pi$ do
11         $\rho_u \leftarrow \rho_u \cup \{(o^i,[k_i,k_{i+1}])\}$
       end
12     store-routing-state($u, \rho_u$)
   end
   function partition-buffer-state($u$)
13     for $(o, T)$ $in$ $\beta_u$ do
14         for $t = (\tau, k, p)$ $in$ $T$ do
15             for $(o',[k_1,k_2])$ $in$ $\rho_u$ do
16                 if $k_1 \le k < k_2$ then $\beta(o') \leftarrow \beta(o') \cup \{t\}$
               end
           end
       end
17     $\beta_u \leftarrow \beta$
   end
```

Algorithm 3: SE partitioning.

hash partitioning, or the key distribution can be used to guide the split, for example to perform range partitioning. The state is retrieved from `backup(o)` (line 3) and is split by partitioning the processing state (line 2). The timestamps associated with the processing state are copied for each partition, and the buffer state is assigned to the first partition (lines 6-7). Finally, the state for each partition is stored in `backup(`o_i`)` in order to provide an initial backup for each partition; afterwards `backup(o)` is removed safely from the system (line 8).

The function `partition-routing-state(u, o, π)` is used to update the routing state of each upstream TE u. The entry for the old key interval is removed and key intervals for the newly allocated TEs are added (lines 9-11). The routing state is then stored at the master node to be recovered in case of failure (line 12).

An upstream TE u can repartition its buffer state β_u according to the updated routing

```
function scale-out-TE($o, \pi$)
1       partition-processing-state($o, \pi$)
2       $\rho \leftarrow$ retrieve-routing-state($o$)
3       for $i \leftarrow 1$ to $\pi$ do
4           $o^i \leftarrow$ get-new-VM-with-TE()
5           $(\theta_i, \vec{\tau}_i, \beta_i) \leftarrow$ retrieve-backup($backup(o), o^i$)
6           restore-state($o^i, \theta_i, \vec{\tau}_i, \beta_i, \rho$)
7           for $d$ in $down(o)$ do  replay-buffer-state($o^i, d$)
        end
8       stop-TE-and-release-VM($o$)
9       for $u$ in $up(o)$ do
10          stop-TE($u$)
11          partition-routing-state($u, o, \pi$)
12          partition-buffer-state($u$)
13          for $i \leftarrow 1$ to $\pi$ do  replay-buffer-state($u, o^i$)
14          start-TE($u$)
        end
end
```

Algorithm 4: Integrated fault tolerant scale out.

state ρ_u using the function `partition-buffer-state(u)`. It iterates over the tuples in β_u (lines 13-14), assigning each tuple to a partition according to the ρ_u key intervals (lines 15-16).

Merge state. In addition to partitioning the state, it is possible to define a *scale in* operation that merges different partitions of state. The scale in process is similar to the scale out. A function *merge* is defined that takes different state partitions and returns a single merged one. Upstream TEs must modify their buffer and routing state accordingly to ensure that all pending traffic in the buffer is sent to the newly allocated machine and that new traffic is routed correctly.

4.2.2 Elasticity: Scaling Out Dynamically

To scale out queries at runtime, the system partitions TEs on-demand in response to bottlenecks. Bottleneck nodes prevent the system from increasing processing throughput, and we discuss heuristics for identifying them in §4.3. After scaling out a TE that was the bottleneck, the TE processing load is shared among a set of newly provisioned TEs—that live in different physical nodes—thus increasing available resources. Our scale out mechanism partitions SEs and streams dynamically while guaranteeing correct results.

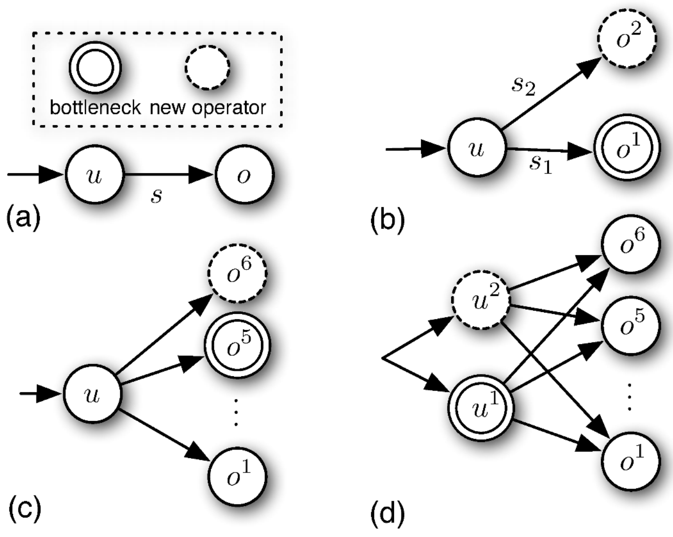

Figure 4.3: Example of scale out of stateful TEs.

In this section, we first give an example of the effect of scaling out a TE, and then we explain Algorithm 4, which is our integrated approach to scale out and fault tolerance. First we focus on how the system performs a scale out and then on how it recover from failures.

The effect of scale out on the execution graph. We give an example of the scale out process in Fig. 4.3, which shows four versions of an execution graph during scale out. When first deployed (Fig. 4.3a), the execution graph has one TE instance for each (logical) TE in the SDG. A TE o is connected through stream s to an upstream TE u. We assume that TE o is the bottleneck TE. Fig. 4.3b shows how the upstream TE u can partition its output streams into two streams. The two partitioned TEs, o_1 and o_2, share the processing load and alleviate the bottleneck condition. In the same way, additional TEs can be added to the execution graph for further scale out (Fig. 4.3c). When the upstream TE u becomes the new bottleneck (Fig. 4.3d), it is also partitioned and its output streams are replicated.

Using the integrated approach algorithm to achieve elasticity. The system has two versions of the TE's state, which can be partitioned for scale out: the current state, maintained by o, and a recent state checkpoint stored by the node that hosts `backup(o)`. The system partitions the most recent state checkpoint.

Algorithm 4 shows the steps for scaling out TE o to a parallelisation level π—that indicates the number of instances of TE in the execution graph. First, the system executes the function `partition-processing-state` to partition o's processing state located on `backup(o)`, backing it up to survive failure (line 1). It also retrieves o's routing state from the master node (line 2). It then creates π newly partitioned TEs o_i to replace o (line 4). After o's processing and buffer state were retrieved from `backup(o)` (line 5), they are restored to the new partitioned TEs o_i (line 6). Tuples yet unprocessed by the downstream TEs are replayed from o_i's buffer state (line 7). After that, o is stopped, and the node that was hosting it is released (line 8).

The next step is to update the execution graph. The system first signals o's upstream to stop processing data TEs (line 10). It updates its routing state to reflect the new incoming TE in the system (line 11). Finally, it repartitions the buffer states (line 12), so that data that was buffered is sent to the new correct downstream TE. After that, the system replays tuples that are not reflected in the state checkpoint from the output buffers (line 13). The final step is to restart the upstream TE (line 14).

Scaling out the most recent checkpoint instead of the current state has several benefits: (i) it means that the scale out mechanism can be used to recover TE o after failure; (ii) it avoids adding further load to TE o, which is already overloaded, by requesting it to checkpoint or partition its own state; and (iii) it makes the scale out process itself fault-tolerant: if it fails or is aborted, TE o can continue processing unaffected.

4.2.3 Fault Tolerance

To be fault tolerant the system must recover the state of a failed node so that computation can continue. We use the integrated approach algorithm to achieve this, by using the state management primitives and keeping *unprocessed* data—that is being currently processed or awaiting in output buffers.

Achieving fault tolerance is just a special case of the scale out mechanism explained in the previous section. To recover from the failure of a stateful TE o, the system simply executes `scale-out-TE(o, 1)`, which restarts processing from o's last checkpoint.

It is also possible to use the scale out capability to reduce recovery time. The system can partition the failed TE state by executing `scale-out-TE(o,2)`. In this case, each restored TE only has to process half of the replayed tuples in `replay-buffer-state`, thus reducing recovery time. We call this approach *parallel recovery*, which is evaluated in Chapter 6.

Some applications require state that grows larger than the memory of a single node. In these applications, the TEs must scale out to aggregate the memory of multiple nodes. The fault tolerance presented cannot handle these scenarios efficiently.

To address these applications, we introduce a new mechanism that is geared towards reducing runtime overhead even when checkpointing *large state*. Next we introduce a generalisation of the fault tolerance algorithm presented in this section that is suitable for all state sizes.

Large State

Designing a fault tolerance mechanism to address the case of applications with *large state* presents the following challenges: (i) it must scale to save and recover the state of a large number of nodes with low overhead, even with frequent failures; (ii) it must have a low impact on the processing latency; and (iii) it must achieve fast recovery time even when recovering large SEs.

We achieve these goals with a mechanism that has the following properties. First, it combines local checkpoints with tuple replay, thus avoiding both global checkpoint co-ordination and global rollbacks. Every TE backs up their SEs independently, with the backups containing a timestamp indicating which tuples are already incorporated in the state. Since timestamps are uniquely assigned by the system, it is possible to track at every point what data has been processed by a TE, and therefore there is no need to coordinate the checkpoint in a global manner.

Second, the mechanism divides state of SEs into consistent state, which is checkpointed, and dirty state, which permits continued processing while checkpointing. While the SE is being checkpointed—note that this process can take a long time when state is large—the SE is frozen, no updates are allowed to such state. Instead, a dirty state maintains all new updates to the state, therefore it permits to continue processing, avoiding an impact on the critical data processing path.

Third, while taking the checkpoint, it will partition the SE and backup each partition to a different node, enabling parallel recovery. Next, we describe the solution in more detail:

Approach. Our failure recovery mechanism combines local checkpointing with tuple logging. Nodes periodically take checkpoints of their local SEs and output buffers. Dataflows include increasing TE-generated scalar timestamps, and a vector timestamp of the last tuple from each input dataflow that modified the SEs is included in the checkpoint. Once the checkpoint is saved to stable storage, upstream nodes can trim their output buffers of data items that are older than all downstream checkpoints.

After failure, a node recovers its SEs from the last checkpoint, replays its output buffers and reprocesses data items received from the upstream output buffers. Downstream nodes detect duplicate tuples based on the timestamps and discard them. This approach allows nodes to recover SEs locally beyond the last checkpoint, without requiring nodes to coordinate global rollback, and it avoids the output commit problem.

State checkpointing. We use an asynchronous parallel checkpointing mechanism that minimises the processing interruption when checkpointing large SEs with GBs of memory. The idea is to record updates in a data structure—that we call *dirty state*—that is compatible with the original one, while taking a checkpoint. For each type of data structure held by an SE, there must be an implementation that supports the separation of dirty state and its subsequent consolidation.

For example, when the SE is implemented as a dictionary, updates to keys in a dictionary are written to the dirty state, and reads are first served by the dirty state and, only on a miss, by the dictionary. Removals are also kept separately.

Checkpointing of a node works as follows: (1) to initiate a checkpoint, each SE is flagged as dirty and the output buffers are added to the checkpoint; (2) updates from TEs to an SE are now handled using a dirty state data structure: (3) asynchronously to the processing, the now consistent state is added to the checkpoint; (4) the checkpoint is backed up to multiple nodes (see below); and (5) the SE is locked and its state is consolidated with the dirty state, e.g. updates to the dirty state are propagated to the original one, and removals are made effective.

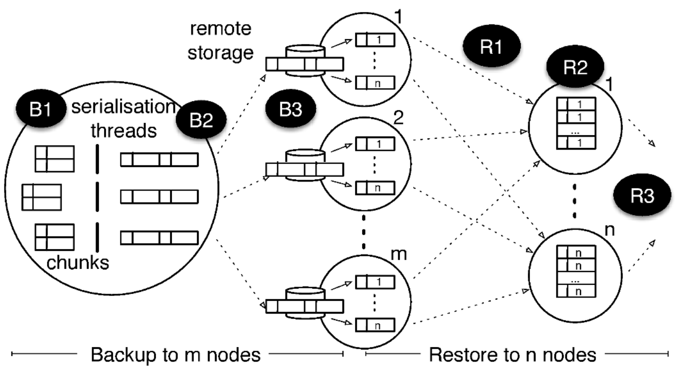

Figure 4.4: Parallel, m-to-n state backup and restore.

State backup and restore. To be memory-efficient, checkpoints must be stored on disk. We overcome the problem of low I/O performance by splitting checkpoints across m nodes. To reduce recovery time, a failed SE instance can be restored to n newly partitioned SE instances in parallel. This $m-to-n$ pattern prevents a single node from becoming a disk, network or processing bottleneck.

Fig. 4.4 shows the distributed protocol for backing up checkpoints. We use the term checkpoint chunk to refer to hash-partitioned checkpoint data that is divided into partitions, or chunks. In step B1, checkpoint chunks are created, and a thread pool serialises them in parallel (step B2). Checkpoint chunks are streamed to m nodes, selected in a round-robin fashion (step B3). Nodes write received checkpoint chunks directly to disk.

After failure, n new nodes are instantiated with the lost TEs and SEs. Each node with a checkpoint chunk hash-partitions it into n partitions, each of which is streamed to one of the recovering instances (step R1). The new instances reconcile the chunks, reverting the partitioning (step R2), and consolidating them as the new SE. Finally, data items from output buffers are reprocessed to bring the recovered SE state up-to-date (step R3).

4.3 Cloud Deployment

In this section, we describe how our integrated approach for scale out and recovery can be realised in system implementation with a focus on a cloud deployment. One benefit of a cloud deployment is that it allows to provision and decommission resources elastically, exploiting the elastic properties of the system. However, achieving this entails

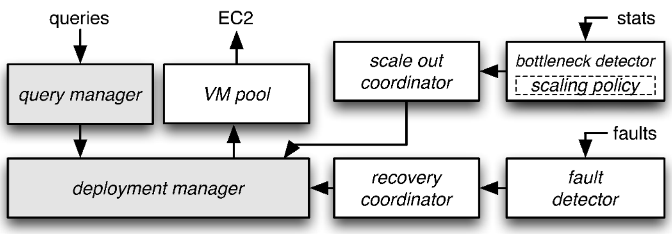

Figure 4.5: Integration with stream processing system.

several challenges: the system must (i) have heuristics for identifying TEs that become a bottleneck; (ii) have a policy as to when to scale out TEs; and (iii) handle the delay when new nodes are provisioned.

Fig. 4.5 shows the architecture of our stateful data-parallel processing system after integrating the scale out and fault tolerance mechanisms.

We assume that both *master* and *worker* nodes are deployed in a cloud environment. A client can connect and interact with the master node to submit and start applications. In particular, an SDG is submitted to a query manager, which performs a mapping of TEs and SEs to nodes or virtual machines (VMs), to obtain an execution graph. The execution graph is used by a deployment manager to initialise VMs, deploy TEs, set upstream communication and start application processing.

To support dynamic scale out, a bottleneck detector identifies the bottleneck TEs in the query based on system statistics, as explained in the next section. In the system, every worker emits statistics—both system and application level—to the master node.

According to a scaling policy, the bottleneck detector invokes the scale out coordinator. For example, after retrieving metrics from the workers, the bottleneck detector may emit an alarm, e.g. after the CPU utilisation of the cluster increases beyond a threshold. When an alarm is produced, the scale out coordinator makes the decision of scaling out a particular node, e.g. a node that hosts a TE that is utilising all CPU available.

For failure recovery, a failure detector notifies the recovery coordinator to recover a failed TE. The deployment manager can request new VM instances from a VM pool. The VM pool masks the delay of cloud platforms when provisioning new VM instances, as described next.

4.3.1 Bottleneck Detection

We adopt a simple yet effective scaling policy that triggers alarms based on a bottleneck detection mechanism that is based on measured CPU utilisation of nodes hosting TEs. Every r seconds, VMs hosting TEs submit CPU utilisation reports to the bottleneck detector, which records the user and system CPU time of each TE executed. This accounts for *stolen* CPU time when other VMs on the same physical node are scheduled. When k consecutive reports from a TE are above a user-defined threshold δ, the bottleneck detector notifies the scale out coordinator to partition the TE.

To determine the parameters r and k depends on the application, resources available and workload. For example, r is increased when the application performance is expected to stay stable. For workloads that present many spikes, it is necessary to both report more often, by reducing r, and adjust k to avoid false alarms by making the policy less aggressive.

4.3.2 VM Provisioning

The time taken to deploy a new TE is a critical issue. When scaling out, VMs must be allocated quickly in order to alleviate a bottleneck that will reduce application throughput otherwise. When recovering a failed VM, fast recovery minimises the disruption caused by the failure. Current Infrastructure-as-a-Service cloud platforms, however, require on the order of minutes to provision new VM instances. This makes it impractical to request new VM instances on-demand when they are required by the system.

Our solution is to decouple the request for a new VM from the provisioning of the VM. We preallocate a small pool of VM instances of size p ahead of time. New VM instances for TEs are requested from this VM pool, which can happen in seconds. Asynchronously the pool is refilled to size p by requesting VM instances from the cloud provider.

A challenge is to decide on the optimal VM pool size. A VM pool that is too small may get exhausted when multiple VMs are requested in short succession. A large VM pool incurs an unnecessarily high financial cost because pre-allocated VMs are billed by the cloud provider.

We make two observations regarding the VM pool size p: (i) with real-world failures, preallocating 1-2 VMs is sufficient, which means that p is primarily determined by the scale out requirement; (ii) p can be adjusted to the scale out behaviour over time. For example, p may be kept larger while the system scales out aggressively to adapt to an open loop workload—one where excess incoming data is dropped. After the rate of new VM requests decreases, the VM pool can shrink to support steady-state operation.

4.4 Summary

In this chapter we addressed the two major challenges when realising stateful dataflow processing: scalability and fault tolerance. First, we explained how making state a first-class citizen permits the definition of primitives to handle state. We make the observation that in a system, a node that fails and a node that scales out are equivalent from a state management perspective. This is a key idea to propose an integrated approach to fault tolerance and scale out, which facilitates the efficient implementation of both properties.

We finished the chapter by considering practical aspects, such as bottleneck detection, that permits the system to trigger scale out decisions and practical aspects to consider when deploying the system on a cloud environment.

Chapter 5

Imperative Big Data Processing

The state in SDGs enables the representation of stateful programs, as we have shown in the two previous chapters. One remaining question is how this can be exposed in a familiar manner to developers. We focus on a particular segment of users who are used to write stateful programs in imperative-style programming languages, such as Java, Python, C++ or Matlab.

This chapter describes a method to translate stateful Java programs to SDGs for parallel execution. We do not attempt to be completely transparent for developers or to address the general problem of automatic code parallelisation. Instead, we exploit data and pipeline parallelism by relying on source code annotations that developers can include in their programs.

The chapter starts with an overview of the translation process (§5.1). We explain that the information that we cannot extract from the input program as it depends on the algorithm semantics, and introduce a number of annotations that developers use to choose the right distributed mutable state (§5.2). We then describe the actual translation process through static code analysis and bytecode engineering (§5.3), limitations of the approach (§5.4) and conclude with a summary (§5.5).

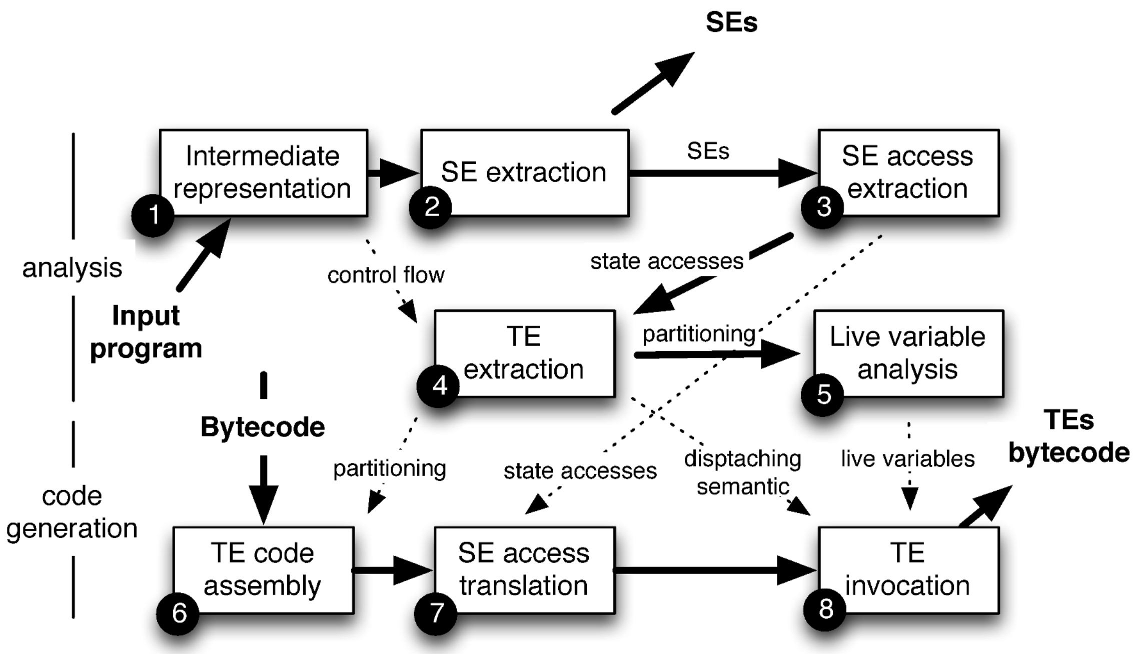

Figure 5.1: Translation of an annotated Java program to an SDG.

5.1 Overview of the Translation Process

In this section we give an overview of the translation process by explaining the intuition behind each of the stages shown in Fig. 5.1. The process is divided into two parts: *analysis* and *code generation*. During *analysis*, the translation process detects the elements of the SDG and the necessary information to construct it. The *code generation* part takes as input the SDG created during *analysis* and generates code that can execute on the targetted stateful data-parallel processing system. We introduce Java2SDG, a tool that implements the *analysis* and *code generation* tasks.

Analysis. The analysis activity encompasses the first five stages in Fig. 5.1. It first represents the input Java program in an intermediate representation (IR)—between Java source code and bytecode—that is amenable for static code analysis (stage 1). The intermediate representation is JIMPLE, which represents source code in a single static assignment (SSA) form. The key properties of the IR with the SSA property is that variables are assigned exactly once, which enforces that they are defined before use. This simplifies the static code analysis later.

During stage 2, the analysis activity extracts the SEs from the code by inspecting its attributes and accesses to such SEs (stage 3). Stage 4 consists of a TE extraction strategy. The strategy extracts TEs in such a way that reduces the number of accesses to SEs from the TEs. Its intuition is that different SEs will be collocated with exclusive access from different TEs, which provides more parallelism opportunities.

Finally, once all TEs have been extracted, live variable analysis extracts the variables that are alive across TE boundaries (stage 5). This is useful to understand the amount and type of data that must be transmitted through the dataflow.

Code generation. During the first stage (stage 6) of code generation, the bytecode of the Java program is logically divided into TEs according to the information provided by the *analysis*. Once all code for the TEs is available, the process adds the necessary API calls to translate state access to the appropriate system functions. For example, it introduces communication API calls at TE boundaries to propagate the necessary data through dataflows. This stage depends on both partitioning information and the type of the mutable state abstraction, as these impose different restrictions on the API calls used.

Finally, in stage 8, the assembled TE code with the translated SE accesses is combined to form an SDG. To perform this, it is necessary to use information that is implicit in the SDG extracted during the *analysis* part.

5.2 Annotations to Disambiguate State Semantics

The translation process explained so far is a pipeline of stages where a Java program is received as input and an executable program is produced as output. This occurs automatically except for certain information. In particular, the two distributed mutable abstractions provided by SDG for scalability and fault tolerance are chosen depending on algorithm semantics.

For this reason, when the analysis stages discover an SE access, it is not possible to automatically determine whether the state must be accessed in a partitioned or partial fashion, or whether it is a local or global access. This information is crucial for the translation process to produce a correct SDG that honours the algorithms semantics.

To resolve this ambiguity, we introduce annotations that are used by developers to indicate what type of distributed mutable state abstraction to use. Therefore this provides all necessary detail to carry on the translation process.

We revisit the example of Algorithm 1, this time introducing all necessary annotations required by the translation process to generate an application that can run on a stateful data-parallel processing system. Algorithm 5 shows the complete example with the annotations required by the translation process.

When defining a field in a Java class, a developer can indicate if its content can be partitioned or is partial by annotating the field declaration with `@Partitioned` or `@Partial`, respectively.

`@Partitioned`. This annotation specifies that a field can be split into disjoint partitions. A reference to a `@Partitioned` field always refers to a single partition. This requires that accesses to the field use an access key to infer the partition.

In Algorithm 5, rows of the `userItem` matrix are updated with information about a single user only, and thus `userItem` can be declared as a partitioned field.

`@Partial`. Fields are annotated with `@Partial` if distributed instances of the field should be accessed independently. Partial fields enable developers to define distributed state when it cannot be partitioned.

In collaborative filtering, matrix `coOcc` is annotated with `@Partial`, which means that multiple instances of the matrix may be created, and each of them is updated independently for users in a partition (lines 13–14).

`@Global`. By default, a reference to a `@Partial` field refers to only one of its instances. While most of the time computation should apply to one instance to make independent progress, it may also be necessary to support operations on all instances. A field reference annotated with `@Global` forces a Java expression to apply to all instances, denoting global access to a partial field, which introduces a synchronisation barrier in the SDG.

Java expressions deriving from `@Global` access become logically multi-valued—the variable exists in every partial instance SE and therefore contains multiple values—because they include results from all instances of a partial field. As a result, any local variable that is assigned the result of a global field access becomes partial and must be annotated as such. This avoids errors in the program, as developers must deal with these partial variables explicitly.

In collaborative filtering, the access to the `coOcc` field carries the `@Global` annotation to compute all partial recommendations: each instance of `coOcc` is multiplied with the user rating vector `userRow`, and the results are stored in the partial local variable `userRec` (line 20).

`@Collection`. Global access to a partial field applies to all instances, but it hides the individual instances from the developer. For global reads, it is necessary to access all partial values so that they can be reconciled in a single value, meaningful for the application. The `@Collection` annotation exposes all instances of a partial field or variable as a Java array after `@Global` access. This enables the program to iterate over all values and merge them as necessary into a single value.

In collaborative filtering, the partial recommendations are combined by accessing them using the `@Global` annotation and then invoking the `merge` method (line 21). The parameter of `merge` is annotated with `@Collection`, which specifies that the method can access all instances of the partial `userRec` variable to compute the final recommendation result.

5.3 Translation from Java to SDG

We detail now the analysis and code generation parts of the translation process. For both processes to work we expect Java programs to conform a specification:

1. The input Java program must have a `driver` method that coordinates and manages the execution of the program. Users specify the input (i.e. *sources*) and output (i.e. *sink*) of data as part of this driver program. For example, they can indicate the data source as well as its format, i.e. schema, and serialisation format. Java2SDG simply exposes a minimal API that permits users to write such configuration.
2. Each method that is part of the SDG must be public and declared in the main Java class. Methods that return a value must be connected to a *sink* that handles results.

3. The class can contain as many private methods as required by the program logic and the state must be declared as attributes of the class, and be accessible from all the public methods.

SE generation. The class is compiled to Jimple code, a typed intermediate representation for static analysis used by the Soot framework [soo] (stage 1). The Jimple code is analysed to identify SEs with partitioned or partial fields and partial local variables (stage 2). Based on the annotations in the code, access to SEs is classified as local, partitioned or global (stage 3).

TE and dataflow generation. Next TEs are created so that each TE only accesses a single SE, i.e. a new TE is created from a block of code when access to a different SE or a different instance of the current SE is detected (stage 4). The dispatching semantics of the dataflows between created TEs (i.e. *partitioned*, *all-to-one*, *one-to-all* or *one-to-any*) is chosen based on the type of state access.

More specifically, a new TE is created with the following strategy that tries to optimise the placement of SE with TEs to foster parallelisation opportunities in the SDG:

1. Java2SDG iterates over the methods of the class. All static analyses are intraprocedural. For each method, Java2SDG proceeds with the following steps;

2. when a TE uses partitioned access to a new SE (or to a previously-accessed SE with a new access key), the access key is extracted using reaching expression analysis, and the dataflow edge between the two TEs is annotated with the access key;

3. when a TE uses global access to a new partial SE, the dataflow edge between the two TEs is annotated with *one-to-all* dispatching semantics;

4. when a TE uses local access to a new partial SE, the dataflow edge is annotated with *one-to-any* dispatching semantics. In case of local (or partitioned) access after global access, all TE instances are synchronised using a distributed barrier before control is transferred to the new TE, and the dataflow edge has *all-to-one* dispatching semantics; and

5. for `@Collection` expressions. A synchronisation barrier collects values from multiple TE instances, and its dataflow edge has *all-to-one* semantics. After generating the TEs,

Java2SDG identifies the variables that must propagate across TEs boundaries (stage 5). For each dataflow, live variable analysis identifies the set of variables that are associated with that dataflow edge.

Bytecode generation. Next Java2SDG synthesises the bytecode for each TE that is executed by the system. It compiles the code assigned with each TE in stage 4 to bytecode and injects it into a TE template (stage 6) using Javassist [JBo], a tool that performs bytecode engineering. State accesses to fields and partial variables are translated to invocations of the runtime system, which manages the SE instances (stage 7).

Finally communication code across TEs for data dispatching is added (stage 8). Java2SDG produces code, (i) at the exit point of TEs, to serialise live variables and send them to the correct successor TE instance; and (ii) at the entry point of a TE, to add barriers for *all-to-one* dispatching and to gather partial results for merge TEs.

5.4 Limitations

Java programs also need to obey certain restrictions to be translated to SDGs.

State element implementation. We provide a standard collection of data structures that users must use as the SE implementations. This permits the system to partition objects of these classes into multiple instances (for partitioned state) or distribute them (for partial state), and recover them after failure. In particular, these classes include support for dirty state—required to achieve low-overhead checkpoints. When users require different custom SE, they can implement the following interfaces: (i) `Versionable`, which permits to create a dirty state structure; (ii) `Streamable`, which permits to stream the state through the network avoiding unnecesary data copying and; (iii) `Mergeable` that implements the necessary logic to merge together different data structures. When the state can be partitioned, users can also implement `Partitionable` to provide such functionality.

Location independence. Each object accessed in the program must support transparent serialisation/deserialisation: as SDGs are distributed, objects are propagated between nodes. This means that objects used by developers must be serialisable by the runtime. Developers can provide a serialiser/deserialiser to achieve this.

The program also cannot make assumptions about its execution environment, e.g. by relying on local network sockets or files. Instead, an API permits developers to specify input and output resources—data sources and sinks.

Side-effect-free parallelism. In Java, arguments that are passed to functions through references, such as objects, can be internally updated by the method. In order to support the parallel evaluation of multi-valued expressions under `@Global` state access, such expressions must not affect single-valued expressions: they must only read input parameters. As a concrete example, the statement `@Global coOcc.multiply(userRow)`, in line 20 in Algorithm 5 cannot update `userRow`.

Deterministic execution. The program must be deterministic, i.e. it should not depend on system time or random input. This enables the runtime system to re-execute computation when recovering after failure. When computation is not deterministic it is impossible to track which data has already been processed by downstream nodes, as this can change after a failure occurs. For this reason crucial system features that depend on non deterministic code, such as encryption, should be implemented at the system level and not at the user level.

5.5 Summary

In this chapter, we presented a technique to translate imperative code to a stateful dataflow graph, amenable to be executed by a stateful data-parallel processing system. The techniques presented require stateful dataflow graphs—the presence of explicit state in the SDG is a key feature required to capture the state of imperative programs.

We introduced Java2SDG, a tool that translates Java programs to applications that can run on top of a stateful data-parallel processing system. We discussed the annotations that are used by developers to disambiguate algorithm semantics and this provides the necessary information for Java2SDG to perform the translation.

As part of the translation process, we discussed some of the limitations of the approach. For example, users must provide their code in a Java template (function `configure()` in Algorithm 5). No side-effects are permitted other than modifications to the state, and code cannot depend on resources at a fixed location.

```java

public class CollaborativeFiltering implements SeepProgram {

 @Partitioned Matrix userItem = new Matrix();
 @Partial Matrix coOcc = new Matrix();

 void addRating(int user, int item, int rating) {
   userItem.setElement(user, item, rating);
   Vector userRow = userItem.getRow(user);
   for (int i = 0; i < userRow.size(); i++)
     if (userRow.get(i) > 0) {
       int count = coOcc.getElement(item, i);
       coOcc.setElement(item, i, count + 1);
       coOcc.setElement(i, item, count + 1);
     }
 }

 Vector getRecommendation(int user) {
   Vector userRow = userItem.getRow(user);
   @Partial Vector userRec = @Global coOcc.multiply(userRow);
   Vector rec = merge(@Global userRec);
   return rec;
 }

 Vector merge(@Collection Vector[] allUserRec) {
   Vector rec = new Vector(allUserRec[0].size());
   for (Vector cur : allUserRec)
     for (int i = 0; i < allUserRec[0].size(); i++)
       rec.set(i, cur.get(i) + rec.get(i));
   return rec;
 }

 @Override
 public SeepProgramConfiguration configure(){
   SeepProgramConfiguration spc = new SeepProgramConfiguration();
   Schema sch = SchemaBuilder.getInstance()
     .newField(Type.INT, "user")
     .newField(Type.INT, "item")
     .newField(Type.INT, "rating")
     .build();
   DataStore addRatingSrc = new DataStore(DataStoreType.NETWORK);
   spc.newWorkflow("addRating", addRatingSrc, sch);
   Schema sch2 = SchemaBuilder.getInstance().newField(Type.INT, "user").build();
   DataStore getRecSrc = new DataStore(DataStoreType.NETWORK);
   DataStore sink = new DataStore(DataStoreType.CONSOLE);
   spc.newWorkflow("getRecommendation", getRecSrc, sch2, sink, sch2);
   return spc;
 }
}
```

Algorithm 5: Annotated collaborative filtering in Java.

Chapter 6

Evaluation

In this chapter we evaluate SEEP, a prototype implementation, to validate stateful data-parallel processing as a new model to perform large scale data analysis with high performance, low latency and in a fault-tolerant manner. First, we provide an overview of the questions that we want to answer followed by a summary of the evaluation results.

Overview of evaluation. To evaluate the feasibility and performance of stateful data-parallel processing, we explore the new model as part of a prototype implementation in SEEP, a data-parallel processing system.

- **Can stateful data-parallel processing achieve high-performance competitive with other state-of-the-art systems?** We evaluate throughput and latency with well-known workloads in both *closed-loop* and *open-loop* scenarios. We compare the performance results against other systems such as Spark and Naiad.
- **Is it possible to achieve fault tolerance with SDGs?** We show the fault tolerance properties of SEEP in terms of both fast recovery and low runtime overhead, which is crucial for applications that require low latency results.

As part of the evaluation we also explore other factors closely related to the scalability and fault tolerance of SEEP. We want to understand the overheads associated with different checkpointing intervals, and how the state size impacts runtime overhead, as well as how efficiently the scale out strategy overcomes stragglers.

We evaluate stateful data-parallel processing in the context of a smart-grid analytics application, where the goal is to predict future energy consumption.

The rest of the chapter is structured along three main sections. The first addresses the performance related questions presented above (§6.1) with a thorough evaluation of performance in stateful data-parallel processing with micro-benchmarks and applications. We also present results related to fault tolerance (§6.2) and a comparison with other systems (§6.3). After answering those questions, we use SDGs in practice in the context of the smart grid analytics use case (§6.4).

6.1 Performance

We perform four different experiments to study the throughput and latency characteristics of SEEP:

1. **Closed loop workload.** We first explore the behaviour of SEEP when executing a workload in a closed loop fashion—all data must be processed to yield correct results. We implement the Linear Road Benchmark [ACG$^+$04], which presents an always increasing incoming stream rate, which requires SEEP to scale resources ***elastically***.
2. **Open loop workload.** We relax the requirements of processing all data to determine the behaviour of the system with a top-k query that uses Wikipedia data to report the most frequently visited pages over a time window. The aim of this experiment is to report throughput with ***non-trivial state***.
3. **Online collaborative filtering.** We use the collaborative filtering example to evaluate a demanding workload in which state spans multiple machines. Here ***both high throughput and low latency*** are important.
4. **Distributed mutable state.** We stress the system with the extreme case in distributed mutable state in a key-value store. Here we want to understand the throughput and latency properties of an application that requires ***fine-grained updates*** to state.

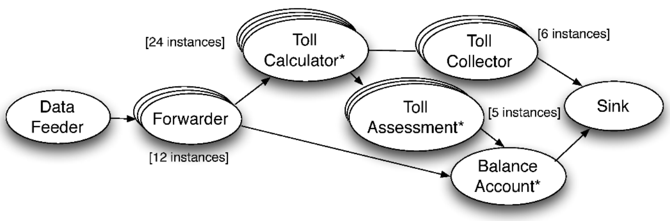

Figure 6.1: Query for the Linear Road Benchmark.

Closed loop workload. We first evaluate the effectiveness of our scale out approach—i.e. its ability to increase throughput while maintaining latency—when adapting to an increasing closed loop workload. The goal is to scale out automatically, as necessary, when the input rate increases, without tuple loss.

For this experiment, we use the Linear Road Benchmark (LRB), a streaming workload used by others to evaluate similar properties [ZR11, JAA$^+$06]. LRB models a road toll network, in which tolls depend on the time of day and the level of congestion. It specifies the following queries: (i) provide toll notifications to vehicles within 5 s; (ii) detect accidents within 5 s; and (iii) answer balance account queries about paid toll amounts. The goal is to support the highest number of express ways L—i.e. events per second—while satisfying the above latency constraint. Over the course of the benchmark, the input rate for a single express-way (L=1) begins at 15 tuples/s and increases to 1700 tuples/s. A system without dynamic scale out support would have to be provisioned to sustain the peak rate, which typically leads to a cost-inefficient overprovisioned infrastructure. To generate a sufficiently high input stream rate, we precompute the input stream for L=1 in memory and replicate it for multiple express-ways, which is an approach used in the literature [DRAT11].

Our LRB query implementation consists of 7 task elements, as shown in Fig. 6.1. A *data feeder* acts as the source and generates the input data stream. A *forwarder* TE routes tuples downstream according to their type. The stateful *toll calculator* maintains tolls and detects accidents. The stateful *toll assessment* TE computes account balances and responds to balance queries in tuples. The stateless *collector* TE gathers notifications. The stateful *balance account* TE receives the balance account notifications and aggregates the results. The *sink* TE collects all results.

We deploy the LRB query on Amazon EC2 [Amaa] and set up the system to scale out

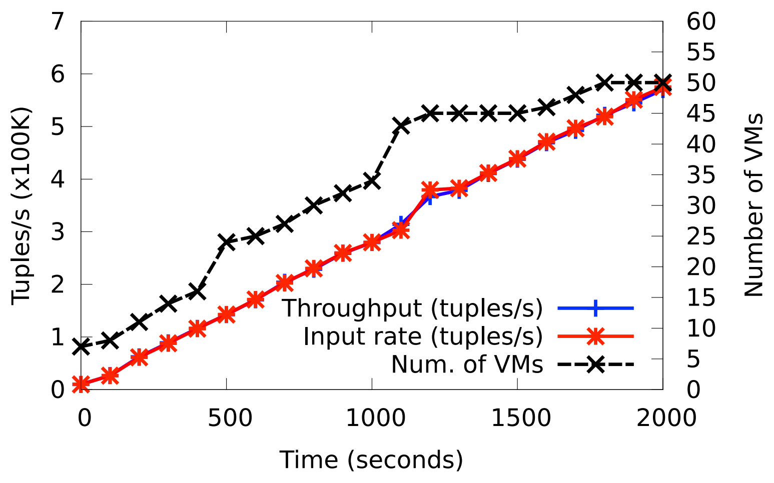

Figure 6.2: Dynamic scale out for the LRB workload with L=350 (closed loop workload).

automatically when the CPU utilisation of a machine rises above a threshold of 70% (such a strategy is effective because the workload is CPU-bound).

Fig. 6.2 shows the number of allocated VMs along with the input rate and the achieved result throughput over time. For L=350, the input rate is initially approx. 12,000 tuples/s and increases to 600,000 tuples/s. We observe that the system maintains the required result throughput for the input rate, requesting additional VMs as needed. At times t=475 and t=1016, multiple TEs are scaled out in close succession because bottlenecks appear in two TEs simultaneously.

Our deployment achieves a maximum L-rating of L=350 with 50 VMs. After that, the source and the sink become the bottleneck, handling a maximum of 600,000 tuples/s due to serialisation overheads. The main computational bottleneck in the query, the *toll calculator*, is scaled out many times by the system, followed by the *forwarder*.

This result is about 70% of L=512, which is currently the highest reported L-rating in the literature by Zeitler and Risch [ZR11]. Their result was obtained on a private cluster with 560 dedicated CPU cores with 2.27 Ghz—substantially more resources than we use. Since our approach only scales out bottleneck TEs at a fine granularity, it can be more resource efficient than the replication of the whole query graph used by Zeitler and Risch.

In Fig. 6.3, we show the processing latencies of output tuples, as a representative metric for the performance experienced by the query. The 99th and 95th percentiles of the latency are 1459 ms and 700 ms respectively and the median is 153 ms, which are all

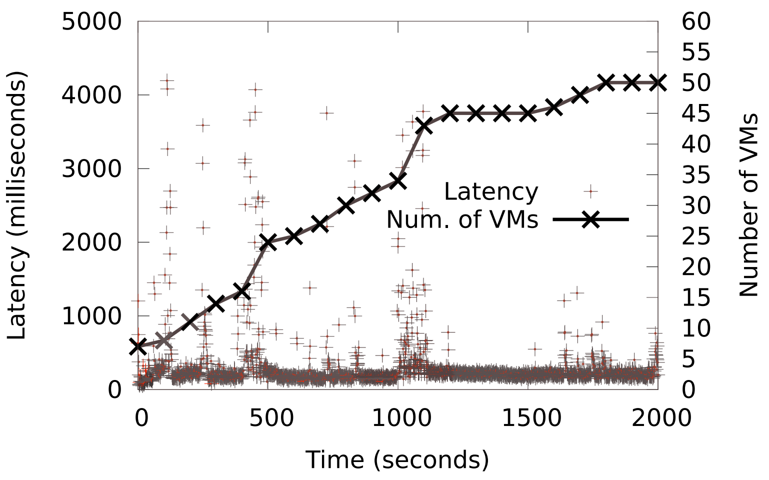

Figure 6.3: Processing latency for LRB workload.

below the LRB target of 5 s. This confirms that our maximum L-rating is indeed due to the limited source and sink capacities. Dynamic scale out, however, affects tuple latency—there are latency peaks of up to 4 s after scale out events due to stream buffering and replay. Towards the end of the experiment, the median latency begin increasing when the system becomes overloaded.

SEEP implements the SDG model efficiently, and it is capable of scaling out stateful TEs on demand as the workload increases. Elasticity gives it an advantage with respect to non-elastic systems, which typically require over-provisioned deployments to cope with workload peaks.

Open loop workload. We explore an open loop workload, in which we initially under-provision the system leading to tuple loss, and then let the system scale out to handle the workload.

MapReduce style top-k query. We implement a *top-k query* that outputs every 30 seconds the ranking of the most visited Wikipedia language versions based on Wikipedia data traces. Initially, we set the input stream rate to be above the performance capacity of the system, thus incurring tuple loss. The goal is to let the system scale out the query in order to sustain the incoming rate.

We use Amazon EC2 VMs for this experiment. We use 18 data sources to inject tuples to a stateless *map* TE, which removes unnecessary fields, and a stateful *reduce* TE, which maintains a top-k dictionary of the frequencies of visited Wikipedia language versions.

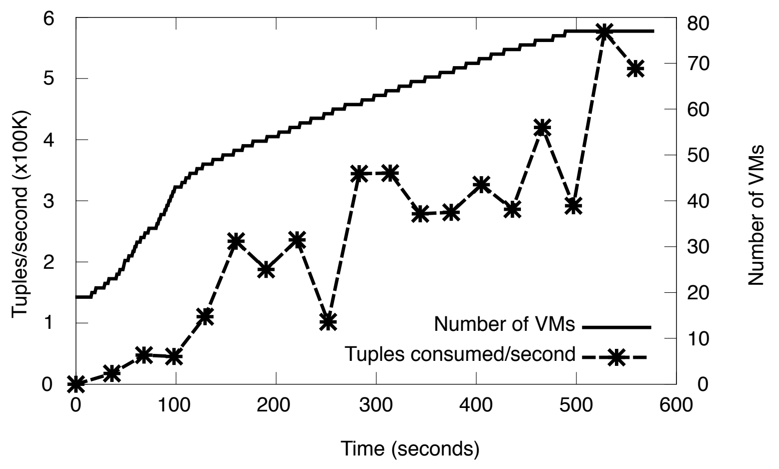

Figure 6.4: Dynamic scale out for a MapReduce style workload (open loop workload).

When the reducer scales out, we use the sink to aggregate the partial results and output the final answer.

We present the dynamic scale out behaviour in Fig. 6.4. As expected, SEEP scales out until it can sustain the incoming rate of 550,000 tuples/s. The scale out process leads to peaks in the tuple throughput. After scaling out a TE, the input buffers of the new partitioned TEs consume tuples faster than they can be processed. Only after the input queues have filled, performance stabilises again.

Another observation is that the rate of scale out is higher in the first part of the experiment (until t=100). The reason is that initially more map TEs are scaled out than reduce TEs: the stateless map TEs scale out faster than the stateful reduce TEs.

Unlike the experiment with the closed-loop workload, the open loop workload overloads the system initially. We show that even in this situation, and with the non-trivial state required by a top-k query over a 30-second window, the system can scale out resources to handle the demand.

Online collaborative filtering experiment. The *online collaborative filtering* application (see §1) stresses SEEP in two ways. It requires high-throughput in order to incorporate the latest ratings produced by users, and it needs to provide low-latency responses to recommendation requests to maintain a good user experience.

We deploy it on 36 Amazon EC2 VM instances ("c1.xlarge"; 8 vCPUs with 7 GB) using

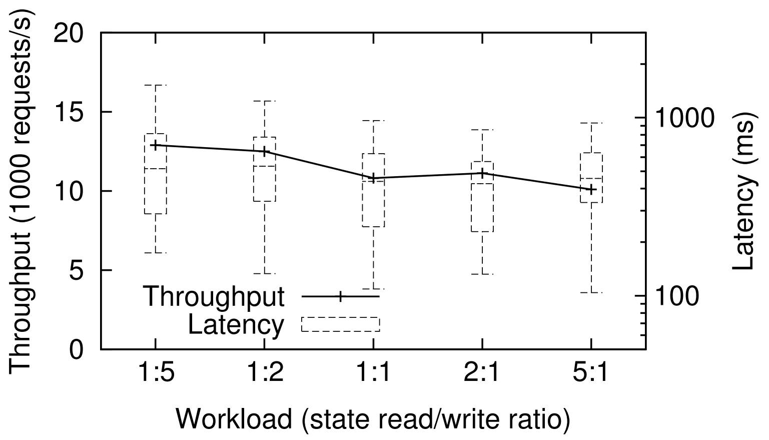

Figure 6.5: Throughput and latency with different read/write ratios (online collaborative filtering).

the Netflix dataset [Net], which contains 100 million movie ratings for the evaluation of recommender systems. We add new ratings continuously using the function `addRating`, while requesting fresh recommendations using the function `getRec`. The state size maintained by the system grows to 12 GB.

Fig. 6.5 shows the throughput of `getRec` and `addRating` requests and the latencies of `getRec` requests when the ratio between the two is changed. The achieved throughput is sufficient to serve 10,000–14,000 requests/s, with the 95th percentile of responses being at most 1.5 s stale.

As the workload ratio includes more state reads (`getRec`), the throughput decreases slightly. This is due to the cost of the synchronisation barrier that is necessary to aggregate the partial results of the multiple instances hosting a partial SE in the SDG.

With this experiment, we demonstrate that SDGs can combine the functionality of batch and online processing systems, while serving fresh results with low latency and high throughput over large mutable state.

Impact of state size. Next we evaluate the performance of SDGs as the state size increases. As a synthetic benchmark, we implement a distributed partitioned *key/value store* using SDGs because it exemplifies an application with pure mutable state. We compare to an equivalent implementation in Naiad [MMI+13] with global checkpointing, which is the only fault-tolerance mechanism available in the open-source version. The objective of this comparison is purely to serve as a reference. We use two differ-

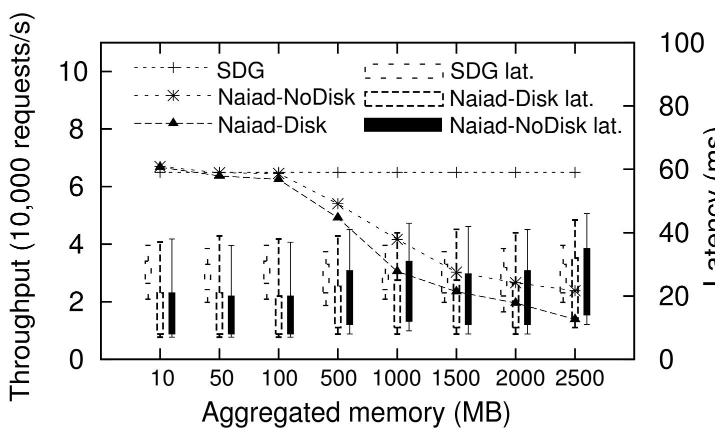

Figure 6.6: Throughput and latency with increasing state size on single node (key/value store).

ent configurations for Naiad: Naiad-Disk is a configuration that stores checkpoints on disk; Naiad-NoDisk, instead, relies on RAM disk storage of checkpoints. We deploy this application in one VM ("m1.xlarge") and measure the performance of serving update requests for keys.

Fig. 6.6 shows that, for a small state size of 100 MB, both SDGs and Naiad exhibit similar throughput of 65,000 requests/s with low latency. As the state size increases to 2.5 GB, the SDG throughput is largely unaffected but Naiad's throughput decreases due to the overhead of its disk-based checkpoints (Naiad-Disk), which is the mechanism used by Naiad to implement fault tolerance. Even with checkpoints stored on a RAM disk (Naiad-NoDisk), its throughput with 2.5 GB of state is 63% lower than that of SDGs, due to the stop-the-world-approach used by Naiad. Similarly, the 95th percentile latency in Naiad increases when it stops processing during checkpointing—SDGs do not suffer from this problem.

To investigate how SDGs can support large distributed state across multiple nodes, we scale the *key value store* store by increasing the number of VMs from 10 to 40, keeping the number of dictionary keys per node constant at 5 GB.

Fig. 6.7 shows the throughput and the latency for read requests with a given total state size. The aggregate throughput scales near linearly from 470,000 requests/s for 50 GB to 1.5 million requests/s for 200 GB. The median latency increases from 8 ms to 29 ms, while the 95th percentile latency varies between 800 ms and 1000 ms.

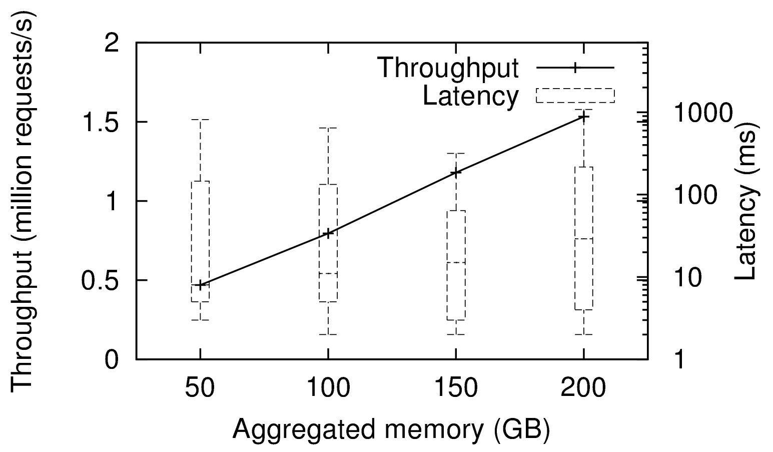

Figure 6.7: Throughput and latency with increasing state size on multiple nodes (key/-value store).

This result shows that SDGs can support stateful applications with large state, which is continuously updated in a fine-grained fashion, without compromising throughput or latency, and while maintaining fault tolerance.

Scaling Policy Evaluation

We want to understand the effect of varying the scale-out threshold §6.1, and the efficiency of a deployment in which SEEP decides to scale out TEs automatically. One important problem in the context of data-parallel processing is the appearance of stragglers. We therefore study also the mechanism based on scale-out employed by SEEP to address stragglers (§6.1).

What is the impact of varying the scale-out threshold?

We evaluate the impact of the scale out policy. We study how different scale out thresholds δ affect the number of allocated VMs and the tuple processing latency (see Fig. 6.8). The goal is to find the best trade-off between resource allocation efficiency and processing performance. To explore the efficiency of VM allocation, we compare our dynamic scale out approach to manual scale out by a human expert. We investigate different thresholds δ using LRB with L=64. We initially deploy the query with one VM per TE and observe the number of VMs at the end of the experiment and the processing latency.

Fig. 6.8 shows that, as δ increases from 10% to 90%, fewer VMs are allocated. The median latency curve is concave, increasing not only for high thresholds, when VMs

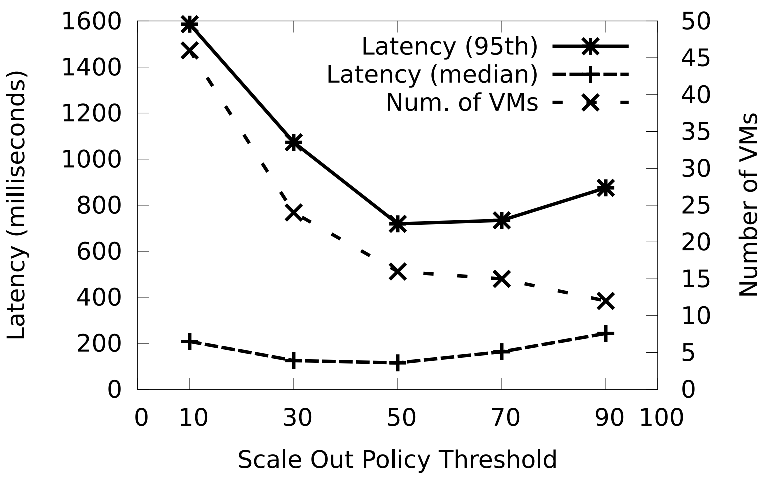

Figure 6.8: Impact of the scale out threshold δ on processing latency.

are close to overload, but also for low ones. This behaviour can be understood better by considering the 95th percentile of tuple latencies. When δ is small, the system performs many scale out operations, which impacts processing latency, especially at higher percentiles.

Based on these results, a threshold δ of 50%–70% provides the best trade-off. It follows the best practice in data centre management to maintain a headroom of unused resources in anticipation of workload bursts and transient peaks [GHMP08].

How efficient is the automatic scale-out policy?

We evaluate the efficiency of the dynamic scale out policy against a human expert who manually partitions TEs. In this experiment, we use the LRB query with L=115. The human expert is given a fixed number of VMs and uses them as effectively as possible to support this workload. The human expert, based on their understanding of the relative costs of TEs, tracks the bottleneck across multiple scaled out versions of the LRB query. The dynamic scale out policy allocates 25 VMs at the end of the experiment.

Fig. 6.9 shows the processing latency as a function of the number of VMs for the manual scale out decisions. In addition, the median (101 ms) and 95th percentile (714 ms) of latencies for the dynamic scale out policy are indicated in the figure. The results show that the most efficient manual allocation for this workload is 20 VMs—with fewer VMs, the 95th latency percentile starts to increase due to the high VM utilisation. In comparison, automatic scale out achieves low latency with only 25% more resources than the

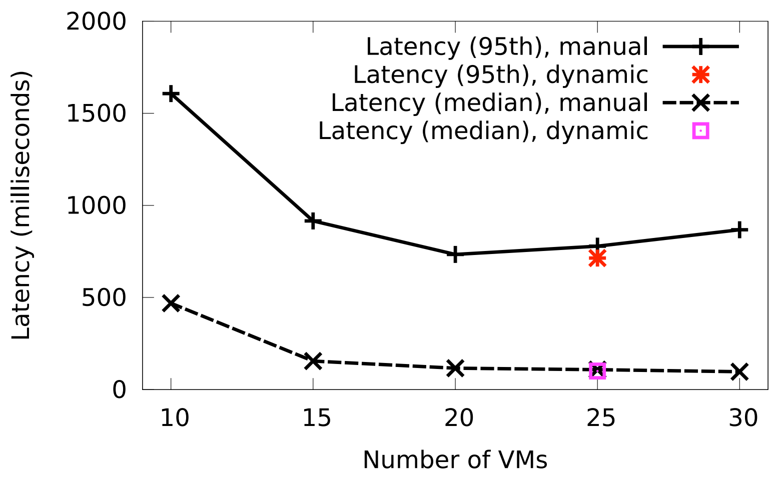

Figure 6.9: Comparison between dynamic and manual scale out.

baseline experiment.

Is it possible to mitigate stragglers?

We explore how SDGs handle straggling nodes by creating new TE and SE instances at runtime. For this, we deploy the collaborative filtering application on our cluster and include a less powerful machine (2.4 GHz with 4 GB).

Fig. 6.10 shows how the throughput and the number of nodes changes over time as bottleneck TEs are identified by the system. At the start, a single instance of the `getRecVec` TE is deployed. It is identified as a bottleneck, and a second instance is added at $t = 10$ s, which also causes a new instance of the partial state in the `coOcc` matrix to be created. This increases the throughput from 3600–6200 requests/s. The throughput spikes occur when the input queues of new TE instances fill up.

Since the new node is allocated on the less powerful machine, it becomes a straggler, limiting overall throughput. At $t = 30$ s, adding a new TE instance without relieving the straggler does not increase the throughput. At $t = 50$ s, the straggling node is detected by the system, and a new instance is created to share its work. This increases the throughput from 6200–11,000 requests/s.

This shows how straggling nodes are mitigated by allocating new TE instances on-demand, distributing new partial or partitioned SE instances as required. In more extreme cases, a straggling node could even be removed and the job resumed from a checkpoint with new nodes.

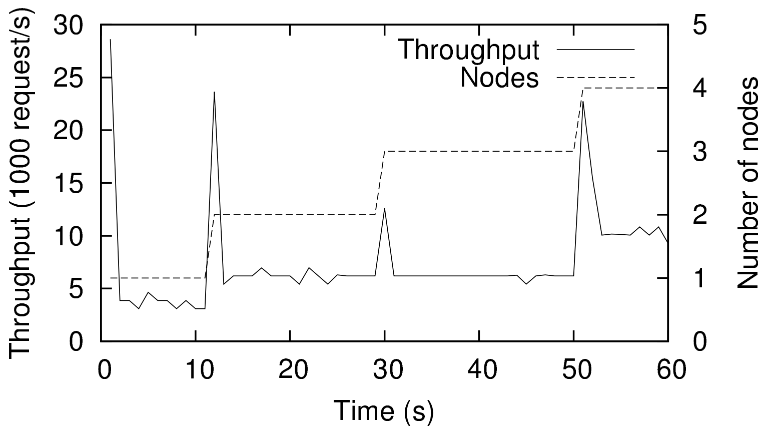

Figure 6.10: Runtime parallelism for handling stragglers (collaborative filtering).

6.2 Fault Tolerance

In this section we evaluate the efficiency of the two mechanisms for fault tolerance described in the thesis, i.e. *recovery using state management, (R+SM)* and the large state mechanisms. We separate the evaluation of both approaches into two sections. We start with the R+SM approach, followed by the large-state approach §6.2.

Recovery Using State Management (R+SM)

To evaluate the R+SM approach we first compare it with other recovery methods for stream processing systems such as *upstream backup (UB)* [BHS09] and *source replay (SR)* [TTS$^+$14b]. The comparison is relevant because the core of SEEP is a stream processing engine that is general enough to support batch-oriented workloads. We put special emphasis on the recovery of stateful TEs, and explore the benefits of performing recovery in parallel—i.e. by scaling out the state first.

For the evaluation, the queries are materialised—each node hosts a portion of the query and communicates with other nodes that contain upstream and downstream portions of the query. UB buffers tuples in each processing node and reprocesses them to recover SE. SR is a variant of UB, in which tuples are only buffered and replayed by the source.

SR and UB are only suitable for stateful TEs whose state can be restored after reprocessing few tuples. As a result, we must use a query for which the state that needs to be maintained is simple. We choose a windowed word count query, which counts word

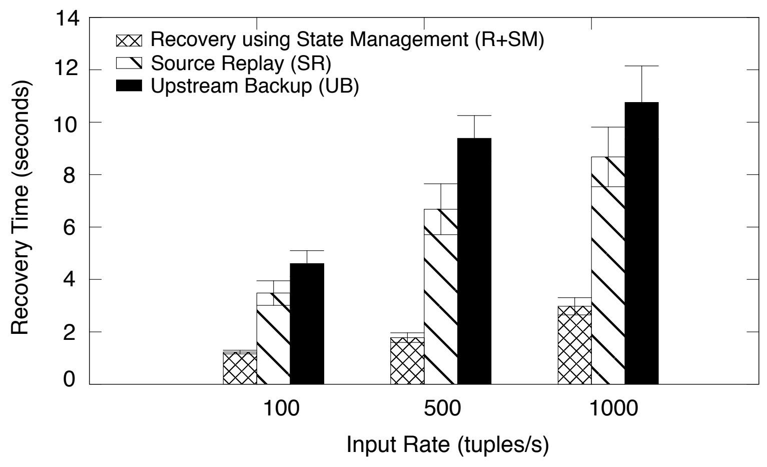

Figure 6.11: Recovery time for different fault tolerance mechanisms.

frequencies over a 30 s window. It executes over a stream of sentence fragments from a Wikipedia corpus, each 140 bytes in size. It has two TEs: a *word splitter* tokenises the input stream into words; and a *word counter* maintains frequency counters for each word. The state of the word counter is a dictionary of words and their counters.

We observe the recovery times for the three approaches. For R+SM we set the check-pointing interval c to 5 s. During the experiment, we fail the VM hosting the *word counter* TE and measure the time to recover (i.e. until the complete TE state was restored).

Fig. 6.11 shows results averaged over 10 runs for different input rates. SR achieves slightly faster recovery than UB because of the short length of the SDG pipeline and the fact that it stops the generation of new tuples during the recovery phase. *R+SM* achieves lower recovery times than both UB and SR. Due to the state checkpoints, it re-processes fewer tuples to recover the stateful TE. In the worst case, it must replay 5 s worth of tuples instead of the entire window of 30 s. Especially at higher input stream rates, the overhead of reprocessing tuples dominates recovery time.

In Fig. 6.12, we show the change in recovery time as a function of the checkpointing interval for different input rates. Tuple buffering is the main factor determining recovery time, which is why recovery time increases considerably with higher rates. While frequent checkpointing incurs overhead, it reduces recovery time, even for high rates.

To prevent the system from falling behind during recovery, *parallel recovery* combines scale out with recovery. In this experiment, we compare serial to parallel *R+SM* with

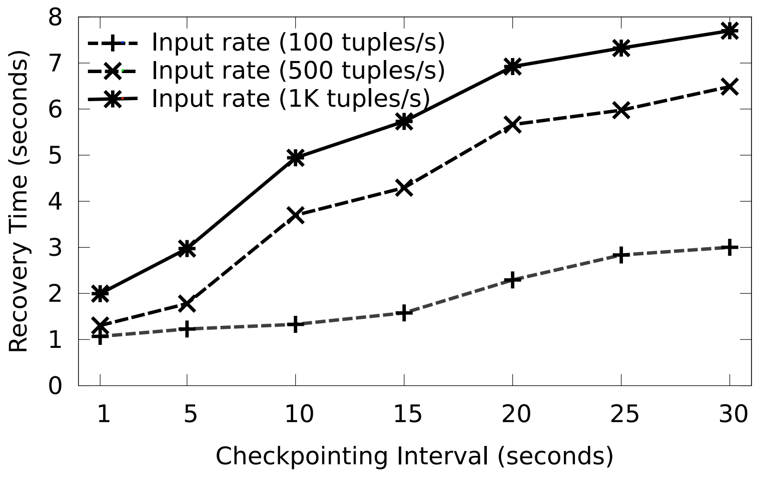

Figure 6.12: Recovery time for different R+SM checkpointing intervals.

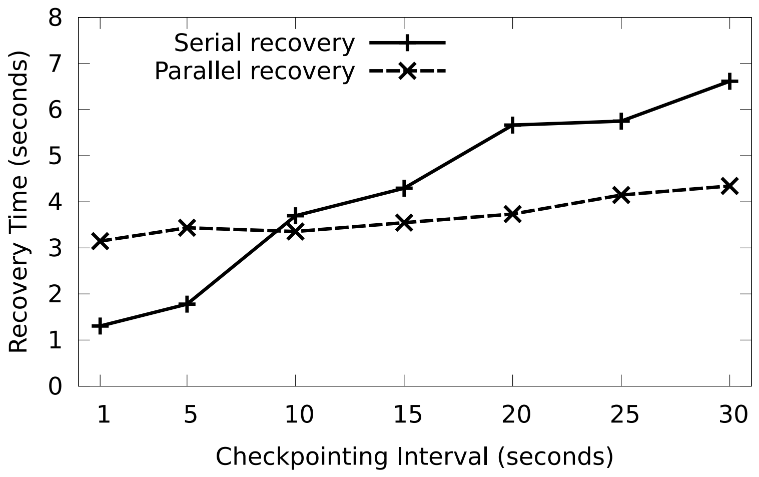

Figure 6.13: Recovery time for serial and parallel recovery using state management.

two partitioned TEs.

Fig. 6.13 shows recovery times for different checkpointing intervals with an input rate of 500 tuples/s. For short intervals, parallel recovery does not bring a benefit due to its higher overhead with two partitioned TEs. As the interval increases however, more tuples have to be replayed when restoring the SE, and parallel recovery can process at a higher rate with two partitioned TEs.

First, the major bottleneck of failure recovery is data replay. This introduces a tension between faster recovery time and thus more frequent checkpoints, with the runtime overhead. Second, parallel recovery can help overcome the previous trade-off, therefore justifying its additional implementation complexity.

Large State Fault Tolerance

Next, we describe the evaluation of the approach for large state fault tolerance. In particular, we: (i) explore the recovery time under different recovery strategies; (ii) assess the advantages of our asynchronous checkpointing mechanism; and (iii) investigate the overhead with different checkpointing frequencies and state sizes.

For these experiments, we deploy the *key value store* on one node of our cluster (8 CPU cores, 2.66 Ghz and 8 GB RAM).

Recovery time. We fail the node under different recovery strategies: an *m*-to-*n* recovery strategy uses *m* backup nodes to restore to *n* recovered nodes. For each, we measure the time to restore the lost SE, reprocess unprocessed data and resume processing.

Fig. 6.14 shows the recovery times for different SE sizes under different strategies: (i) the simplest strategy, 1-to-1, has the longest recovery time, especially with large state sizes, because the state is restored from a single node; (ii) the 2-to-1 strategy streams checkpoint chunks from two nodes in parallel, which improves disk I/O throughput but also increases the load on the recovering node when it reconstitutes the state; (iii) in the 1-to-2 strategy, checkpoint chunks are streamed to two recovering nodes, thus halving the load of state reconstruction; and (iv) the 2-to-2 strategy recovers fastest because

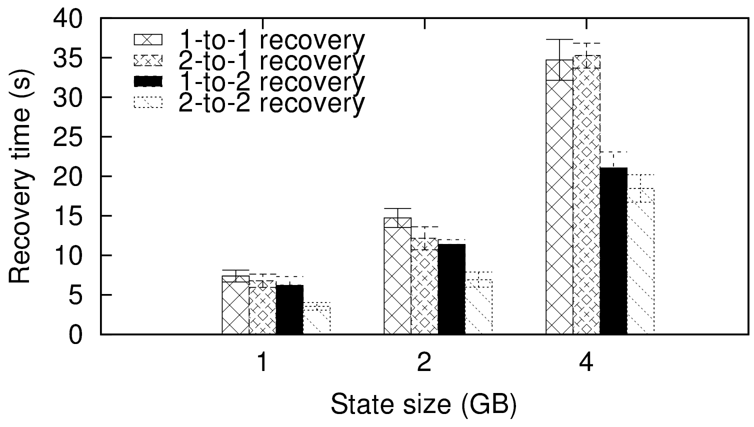

Figure 6.14: Recovery times with different m-to-n recovery strategies.

it combines the above two strategies—it parallelises both the disk reads and the state reconstruction.

As the state becomes large, state reconstruction dominates over disk I/O overhead: with 4 GB, streaming from two disks does not improve recovery time. Adopting a strategy that recovers a failed node with multiple nodes, however, has significant benefit, compared to cases with smaller state sizes.

Synchronous vs. asynchronous checkpointing. We investigate the benefit of our *asynchronous* large state mechanism in comparison with synchronous checkpointing that follows a *stop-the-world* strategy, as used by Naiad and SEEP with R+SM fault tolerance.

Fig. 6.15 compares the throughput and 99th percentile latency with increasing state sizes. As the checkpoint size grows from 1 GB to 4 GB, the average throughput under synchronous checkpointing reduces by 33%, and the latency increases from 2 s to 8 s because the system stops processing while checkpointing. With asynchronous checkpointing, there is only a small (5%) impact on throughput. Latency is an order of magnitude lower and only moderately affected (from 200 ms to 500 ms). This result shows that a synchronous checkpointing approach cannot achieve low-latency processing with large state sizes.

Checkpointing overhead. Next we evaluate the overhead of our checkpointing mechanism as a function of checkpointing frequency and state size.

Fig. 6.16 and Fig. 6.17 show the processing latency when varying the checkpointing fre-

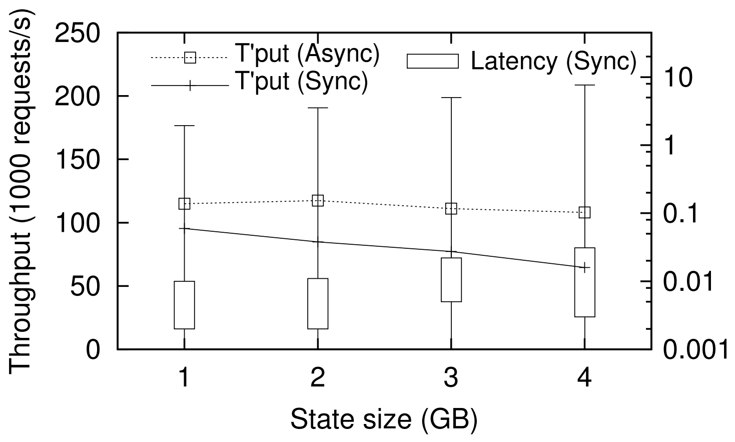

Figure 6.15: Comparison of sync. and async. checkpointing.

quency and size respectively. (No FT) represents the case where the checkpointing mechanism is disabled.

Checkpointing has a limited impact on latency: without fault tolerance, the 95th percentile latency is 68 ms, and it increases to 500 ms when checkpointing 1 GB every 10 s. This is due to the overhead of merging dirty state and saving checkpoints to disk. Increasing the checkpointing frequency or size (see Fig. 6.17) gradually also increases latency: the 95th percentile latency with 4 GB is 850 ms, while checkpointing 2 GB every 4 s results in 1 s.

Beyond that, the checkpointing overhead starts to impact higher percentiles more significantly. Checkpointing frequency and size behave almost proportionally: as the state size increases, the frequency can be reduced to maintain a low processing latency.

Overall this experiment demonstrates the strength of our checkpointing mechanism, which only locks state while merging dirty state. The locking overhead thus reduces proportionally to the state update rate.

We show that the large state mechanism is capable of achieving fast recovery times with a low runtime overhead. By using the N-to-M recovery mechanism, it is possible to alleviate network and IO bottlenecks, hence speeding up checkpointing and recovery times.

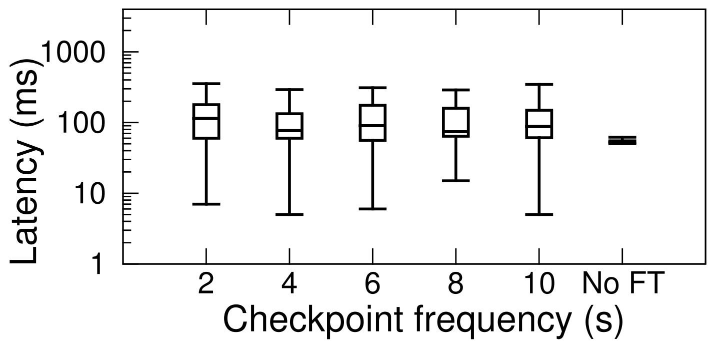

Figure 6.16: Impact of checkpointing frequency on latency.

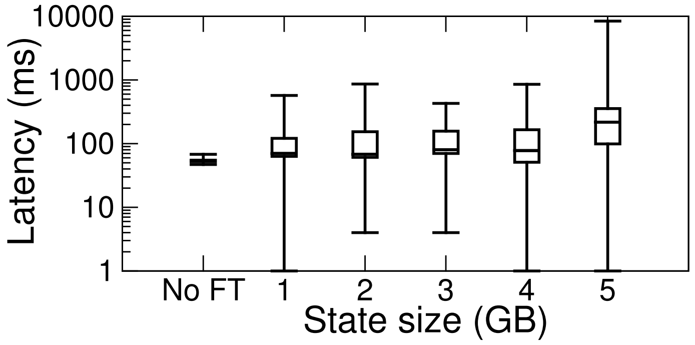

Figure 6.17: Impact of checkpoint size on latency.

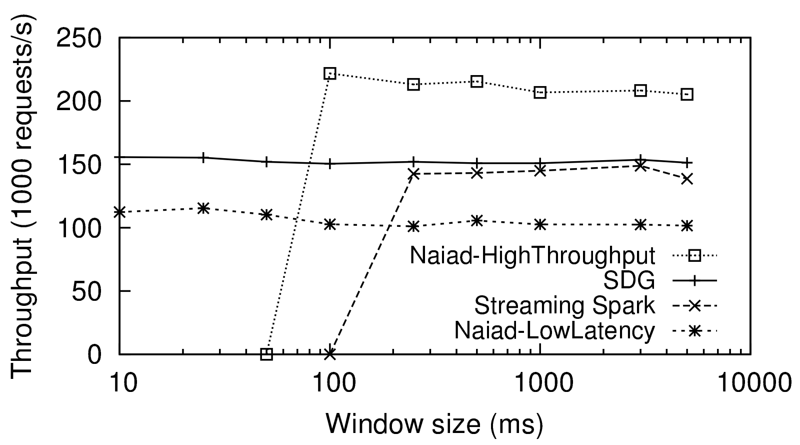

Figure 6.18: Latency with different window sizes (streaming wordcount).

6.3 Comparison with State-of-the-Art

We perform a comparison with other systems of the state-of-the-art along two dimensions. First, we want to understand the support of other systems for fine-grained updates comparing with Naiad and Streaming Spark. Second, we want to understand the differences when executing an iterative batch-oriented workload and compare its performance with Spark.

Update granularity. We show the performance of SDGs with frequent, fine-grained updates to state. For this, we deploy a streaming *word count* application on 4 nodes in our private cluster. *Word count* reports the word frequencies over a wall clock time window while processing the Wikipedia dataset. We compare to *word count* implementations in Streaming Spark [ZDL$^+$13] and Naiad.

We vary the size of the window, which controls the granularity at which input data updates the state: the smaller the window size, the less batching can be done when updating the state. Since Naiad permits the configuration of the batch size independently of the window size, we use a small batch size (1000 messages) for low-latency (Naiad-LowLatency) and a large one (20,000 messages) for high-throughput processing (Naiad-HighThroughput).

Fig. 6.18 shows that only SDG and Naiad-LowLatency can sustain processing for all window sizes, but SDG has a higher throughput due to Naiad's scheduling overhead. The other deployments suffer from the overhead of micro-batching: Streaming Spark has

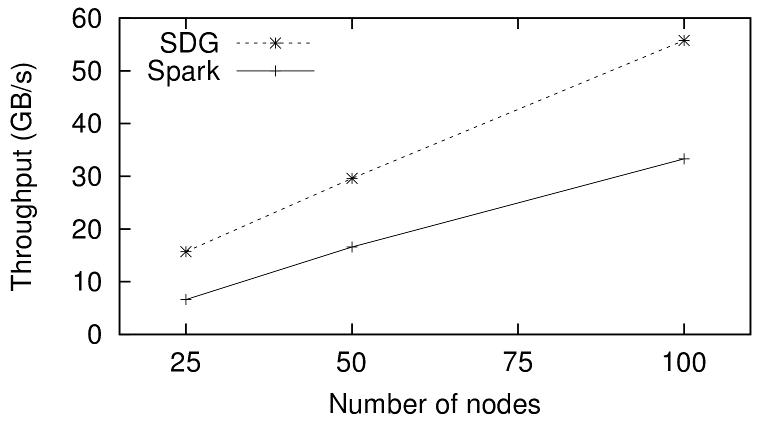

Figure 6.19: Scalability in terms of throughput (batch logistic regression).

a throughput similar to SDG, but its smallest sustainable window size is 250 ms, after which its throughput collapses; Naiad-HighThroughput achieves the highest throughput of all, but it also cannot support windows smaller than 100 ms. This shows that SDGs can perform fine-grained state updates without trading off throughput for latency.

Scalability with iterative batch workload. We explore if SDGs can scale to higher throughput with more nodes in a batch processing scenario. We deploy an implementation of logistic regression [MSSV09] on Amazon EC2 ("m1.xlarge"; 4 vCPUs with 15 GB). We compare to *logistic regression* from Spark [ZCD$^+$12], which is designed for iterative processing, using the 100 GB dataset provided in its release.

Fig. 6.19 shows the throughput of our SDG implementation and Spark for 25–100 nodes. Both systems exhibit linear scalability. The throughput of SDGs is higher than Spark, which is likely due to the pipelining in SDGs, which avoids the re-instantiation of tasks after each iteration. With higher throughput, iterations are shorter, which leads to a faster convergence time. We conclude that the management of partial state in the *logistic regression* application does not limit scalability compared to existing stateless dataflow systems.

6.4 Use Case: Smart Grid Analytics

This section describes a use case of a smart grid analytics application that was introduced as part of the 2014 ACM DEBS Grand Challenge [FWPG14b]. The goal is to conduct a comparative evaluation of systems by offering real-world data and requirements for queries. The 2014 edition of the challenge focuses on measurements of energy consumption at the level of individual electricity plugs in smart home installations. There are two queries: (i) short-term load forecasting and (ii) load statistics for real-time demand management.

We start with a description of the smart grid analytics problem (§6.4.1). After that, we explain the solution implemented as part of SEEP, describing each of the TEs in the SDGs for the queries (§6.4.2). Finally, the section concludes with an experimental evaluation (§6.4.3), which won the Audience Award at the conference.

6.4.1 Problem Description

The dataset available was obtained from smart plugs installed in private households. Each plug acts as a proxy between an appliance that is connected to the power grid and the wall power outlet. The plugs report every second measurements related to power consumption, i.e. work and load.

Houses are identified by a unique *id* in the dataset. Each house contains one or more households, which are identified with a *household-id*, on a per-house basis. Finally, each household contains smart plugs, which are identified with a *plug-id*. Smart plugs measure load in Watts and the total accumulated work since the start of the sensor in kWh units.

The dataset has the following characteristics:

High data volume. The data includes load and work events of individual plugs at a resolution of approximately one measurement per second. For the considered 40 houses with roughly 2000 plugs, this yields a volume of more than 4 billion events for one month. Given future predicted growth, applications of smart grid analytics can be expected to process data of hundreds to thousands of houses, increasing the volume by one or two orders of magnitude.

Unbounded and global state. The queries require sophisticated handling of processing state. Short-term load forecasting requires aggregates to be maintained for an un-

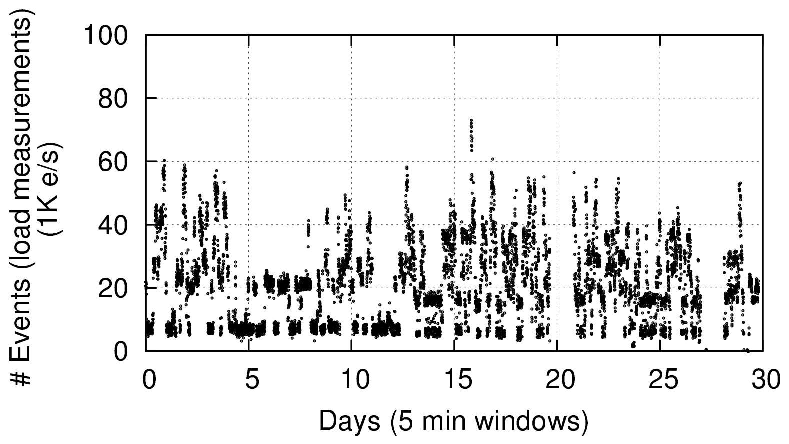

Figure 6.20: Load measurements (five minute window) indicating change of more than 5 watts.

bounded time window. This is challenging due to the ever-growing state to be stored. Performance of computation on this state is likely to degrade over time. Also, load statistics for real-time demand management require global aggregation over all houses and plugs, which limits the options to distribute processing.

Large measurement variability. For the load and work values in the dataset, we observe a large variability in the frequency with which the values change over time. Some plugs report close to constant load for long periods of time, and significant changes in load are correlated between plugs, following global energy consumption patterns. This effect is shown in Fig. 6.20, which depicts the number of load measurements per five-minute window when ignoring changes of less than five watts. Considering only the events that indicate larger changes in load, the amount of events that need to be processed varies by up to a factor of seven over time.

Queries

The analysis uses two different queries. **Query 1** aims to predict load based on current load measurements and a model that is continuously trained, i.e. it represents historical data. In particular, Query 1 must produce predictions on a per-house basis, as well as for each individual plug.

Query 2 aims to detect outliers that may affect the load prediction. The query must report the percentage of plugs per house with a median during the last hour that is greater than the median of all plugs during last hour. The query has two output streams, one is a sliding window of one hour and the other for 24 hours. Windows are defined by application time, i.e. they use timestamps contained and generated in each individual smart plug. The value of the aggregation must be recomputed every time an event enters or leaves a sliding window.

6.4.2 Solution Based on Stateful Data-Parallel Processing

This section describes how to implement the queries in a stateful data-parallel processing model. We first give an overview of the main ideas behind the queries and then we give details on the TE implementations.

1. *Optimised stateful TEs*. Given the complex state of the queries, our solution exploits the stateful data-parallel processing capabilities of SEEP as well as stream TEs with efficient state handling specific to a given query, e.g. through indexed in-memory data structures.

2. *Filtering and elasticity*. We exploit the long periods of relatively constant load measurements in the dataset by performing semantic load-shedding, thus reducing the total events to process downstream. To support resource-efficient deployments when the input event rate varies over time, our solution can dynamically provision processing resources on-demand.

3. *Fault tolerance*. Our solution supports fault-tolerant processing, which is crucial for any continuously running data analytics application on a cluster of nodes. Instead of reprocessing all events after failure, the SE is recovered from periodic state checkpoints with low overhead.

The structure of the logical dataflow graph for the queries is shown in Fig. 6.21, and the code is shown in Algorithm 6. A *filter* TE performs semantic load-shedding across all load and work measurements (denoted by <input>). This permits, for example, filtering of events that indicate only a minor change in load for a certain plug. The stateless filter TE can be scaled out trivially so that different instances realise data parallelism by partitioning the event stream <input> per house, household or even plug.

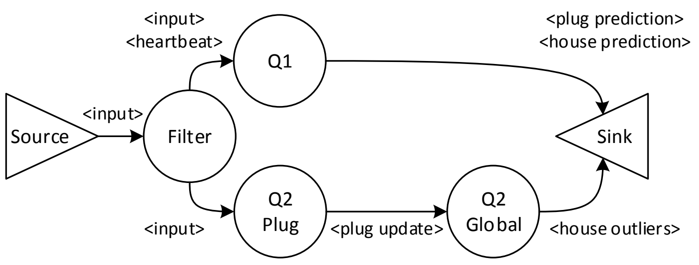

Figure 6.21: The stateful dataflow graph of our solution.

The actual queries are implemented by three TEs, namely *Q1*, *Q2 Plug*, and *Q2 Global*. Load forecasting and outlier detection are independent queries—their execution is done in parallel.

Query 1 for *load forecasting* is realised by TE *Q1*, and it is done at two levels of aggregation, i.e. plugs and houses. Hence, the TE can be scaled out by partitioning the respective event stream for the most coarse-grained aggregation, i.e. per house. Data-parallel processing is of particular importance for this TE because the query requires the maintenance of an unbounded time window, i.e. it must accumulate historical data in its state. At the same time, the query also requires frequent updates of the result stream, i.e. every 30 seconds as specified by the timestamps of the events. When events are streamed faster than real-time (which is of interest for testing the system) and distributed over a large number of TE instances, however, it becomes impossible to identify the intervals for updating the result stream at a particular instance. To solve this issue, we implement a heartbeat mechanism in the filter TE, which emits a signal to TE *Q1* whenever an update is due.

Heartbeat generation is implemented in TE filter because the preprocessing of data is less costly than the actual load prediction. Hence, the number of instances of TE filter can be expected to be much smaller than the number of instances of TE *Q1*. To cope with data quality issues such as missing values, e.g. as seen around days 20 and 28 in Fig. 6.20, TE *Q1* also features a correction mechanism that is based on the measurements of cumulative work per plug, which is detailed below.

Query 2 for *outlier detection* is split up into two TEs. Here, the idea is to separate the part of the query that can be parallelised from the part that requires global state. *Q2 Plug* thus takes the input stream and maintains the median of the load per plug for each of the time windows. The TE can be scaled out at the level of plugs.

Q2 Global, in turn, maintains the global median over all plugs by receiving all changes to medians propagated by upstream nodes. It also realises the outlier detection and emits the results. Due to its global state, the TE cannot be scaled out. To reduce the amount of computation done at the singleton instance of *Q2 Global*, it relies on the information about which measurements entered or left one of the investigated time windows (denoted by <plug update> in Fig. 6.21). As a consequence, a large part of the effort to maintain the time windows per plug is performed by *Q2 Plug*, which can be scaled out. In particular, we do not approximate the global median but rather implement an efficient propagation mechanism to keep it accurate.

Next we describe the TEs for the queries in more detail.

Filter

The filter TE: (i) eliminates duplicates; (ii) performs semantic load-shedding to reduce the incoming data load; and (iii) generates heartbeats to notify downstream TEs of time windows closing and opening.

Duplicate elimination. To filter duplicate measurements, the TE maintains the timestamps of the last load and work measurements for each plug. Only measurements with a timestamp larger than the last observed (per plug) are forwarded.

Variability-based filtering. To leverage the large variability in the frequency with which load values change over time, the filter TE can perform semantic load-shedding, ignoring measurements that denote a minor change in load with respect to the last non-filtered measurement. Note that measurements of work are only forwarded if the load measurement with the same timestamp has not been removed by the filter procedure. We evaluate later the trade-off between this type of filtering and the correctness of the query results.

Heartbeat generation. The aforementioned heartbeats are generated based on the timestamps of the processed events. Whenever an event with a timestamp larger than the time of the last heartbeat plus the heartbeat interval is received, a new heartbeat is emitted.

Query 1: Load Prediction

Q1 for load prediction is implemented as follows:

Prediction model. As a baseline, we rely on a prediction model defined as part of the challenge description, which combines current load measurements with a model over historical data. More specifically, the load prediction for the time window following the next one is based on the average load of the current window and the median of the average loads of windows covering the same time of all past days. The generation of prediction values is triggered by the heartbeats.

Work-based correction. To address the issues stemming from missing load measurements, we focus on the cumulative work per plug. Correction is triggered when the TE receives a work measurement, and the number of recorded load measurements for the preceding window is less than a threshold (chosen based on the expected rate for load measurements).

Since work is measured at a coarse resolution (1 kWh), the work values enable us to derive only an approximation of the actual average load. Therefore, the threshold on the number of load measurements allows for tuning how many load values are required to avoid computation of the window average based on work values.

If applied, the correction mechanism determines the maximum interval of adjacent windows with insufficient load measurements. The difference between the first and last work measurement for this period is used to conclude the average load for all the windows.

State handling. Load forecasting relies on the average load per window per plug over the complete history. To cope with the unbounded state of the query, our implementation strives for reducing the size of the state as much as possible.

First, we observe that although results have to be provided for five different window sizes, all of them can be expressed as multiples of the smallest window of one minute.

Therefore, our implementation only stores the state for the smallest windows.

Second, since prediction is based on the load average, the TE keeps only a sliding average for the current smallest window and the average load for all historic windows. Load averages are kept in a two-dimensional array (per plug, per window), and an index structure allows for quick access of a global identifier for a plug. The index is implemented as a three-dimensional array over the house, household, and plug identifiers.

For the work-based correction mechanism, additional state needs to be maintained. For each plug and window, the number of load measurements and the first recorded work value is maintained in further two-dimensional arrays.

Query 2: Outlier Detection

Outlier detection is realised by TEs *Q2 Plug* and *Q2 Global*. The former focuses on the calculation of windows and the median load per plug; *Q2 Global* maintains the global median and performs the actual outlier detection.

Plug windows and median. To maintain the time windows and calculate the median load per plug, the TE *Q2 Plug* proceeds as follows. On the arrival of a load measurement, the value and timestamp is added to either window (1 hour and 24 hours) for the respective plug. The timestamp of the received event is used to remove old events from both windows. Then, the median of the load values for the plug is calculated. If neither the median not the multi-set of values of both windows changed, no event is forwarded to *Q2 Global*. If there was a change, the new median for the plug as well as the load values added or removed to either window are sent to *Q2 Global* (<plug update>).

Outlier detection. To detect outliers, *Q2 Global* compares the median values per plug as computed by *Q2 Plug* with the global median. To compute the latter, the TE maintains two time windows over all plugs. However, these windows are updated only based on the values provided by the events of the <plug update> stream generated by *Q2 Plug*.

Receiving an event of the *plug update* stream leads to recalculation of the global median for the respective window. If that has not changed, only the house related to the plug for which the update has been received is considered in the outlier detection. If the global median changed, the plugs of all houses are checked. If the percentage of plugs with a median load higher than the global median changes, the result stream is updated.

State handling. To implement the time windows, *Q2 Plug* maintains two double-ended queues for each plug, one containing the timestamps and one containing the load values. Implemented as linked lists, these queues allow the insertion of new measurements in constant time. Accessing and removing events from the queue is done in constant time. The queue containing the timestamps is used to determine whether elements of the queue containing the load values should be removed.

To compute the median over the load values per plug, *Q2 Plug* maintains an indexable skip-list. Such a skip-list holds an ordered sequence of elements and also maintains a linked hierarchy of sub-sequences that skip certain elements of the original list. We use the probabilistic and indexable version of this data structure. The skip paths are randomly chosen and, for each skip path, we also store the length in terms of the number of skipped elements. This is an example of the benefits of using arbitrary SE implementations in SDGs.

The indexable skip-list allows for inserting, deleting and searching load values as well as accessing the load value at a particular list index in logarithmic time. Median calculation is traced back to a list look-up. Since the query requires the look-up only for the median element, and not for an arbitrary index, we also keep a pointer to the current median element of the list, which is updated with every insertion or deletion. Hence, the median is obtained in constant time.

Although bounded, handling the SE of *Q2 Global* is challenging due to the sheer number of measurements that need to be kept (up to 100 million events) and the update frequency. For both windows, our implementation relies on an indexable skip-list and uses a pointer to the median elements of these lists.

6.4.3 Experimental Evaluation

We evaluate the performance of our system by investigating: (i) *if it scales*, i.e. whether it supports more houses; (ii) *if it can cope with the current load with headroom*, i.e. whether the system can process faster than real time; and (iii) *how fast it can incorporate predictions*, i.e. whether the system can achieve low latency, even when it is distributed.

We deploy the SEEP prototype implementation in a private cluster composed of 10 Intel Xeon E3-1220 V2 4-core nodes (3.1 Ghz) with 8 GB of RAM, running a Linux kernel 3.2.0 with Java 7. We execute SEEP with the fault tolerant mechanism enabled.

Scalability. To measure scalability, we report relative throughput, where we normalise the throughput of the system for the baseline case, and show how it increases as we add more cluster nodes. We explain the bottlenecks observed when conducting the experiments.

Throughput. After analysing the available datasets, we find that we need a system capable of processing 377 events/s, 696 events/s and 1565 events/s, on average, for the 10-, 20- and 40-house datasets, respectively, to process the incoming input rate over a month. SEEP processes three orders of magnitude faster than this. For this reason, we report *speedup over RT (real time)* as the number of times that the system process faster than required to run the queries. As an example, a speedup over RT of 200×, which would allow for processing one month worth of data in 15 days.

Latency. We measure the end-to-end latency of those events that close windows in both queries. To measure latency accurately, we place the source and sink of our system on the same node so that both TEs use the same clock.

For the given queries, the processing cost per event is close to constant regardless of the dataset size. Dataset sizes, however, have an impact on the total memory required to run the queries. We exploit the stateful capabilities of SEEP to provide an implementation that expresses the state efficiently. Note that under this scenario, larger datasets do not impact the throughput of our system, but only the speedup, as there are more events to process.

Query 1 Results

The implementation of query 1 consists of two TEs, a *filter* and *Q1* (see Fig. 6.21). For the baseline system, each of the TEs is deployed on a single node of the cluster. For our

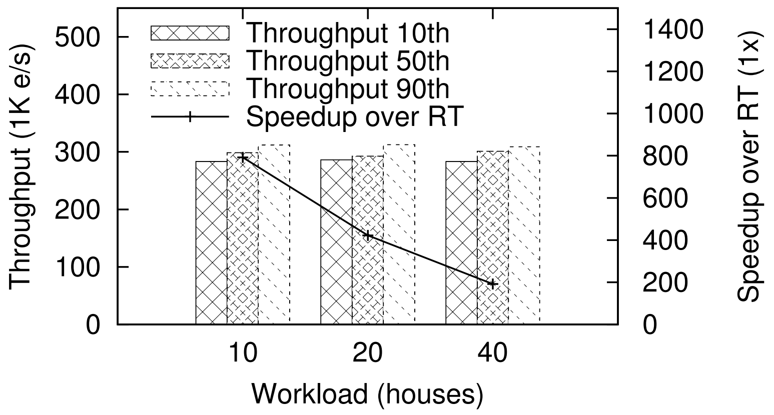

Figure 6.22: Throughput and speedup as a function of the number of houses. With a constant throughput, growing the size of the dataset implies a lower speedup.

distributed deployment, we scale out from 2 to 6 nodes.

Baseline deployment. Fig. 6.22 shows the 10^{th} , 50^{th} , and 90^{th} percentile of throughput. As expected, this is almost constant across the different workload sizes but the speedup over RT decreases as there are more events to process. With a speedup over RT of around 900×, the system can process one month worth of data from 10 houses in about one hour, while it will take around four hours to do the same for 40 houses.

Distributed deployment. Ideally, we want the system to scale to support the data from more houses. This is equivalent to keeping the speedup over RT constant. We exploit data parallelism to aggregate throughput, thus keeping constant or even increasing the speedup over RT.

Fig. 6.23 shows on the x-axis the number of cluster nodes. The relative throughput increases linearly from 2 to 3 nodes, sub-linearly until 5, and then we find a spike when using 6 nodes. The reason for the sub-linear behaviour is due to the sink TE: it aggregates the results from the distributed nodes, becoming an I/O bottleneck. To confirm this, we scale out the sink and run the system with 6 nodes, which shows how the throughput increases again. The speedup over RT in this experiment always increases, which confirms that SEEP can scale to bigger datasets while sustaining the throughput. We stop at 6 nodes when the source becomes a bottleneck. In a real scenario with distributed sources, this would not be an issue.

Table 6.24 shows the latencies for both the baseline and the distributed deployment.

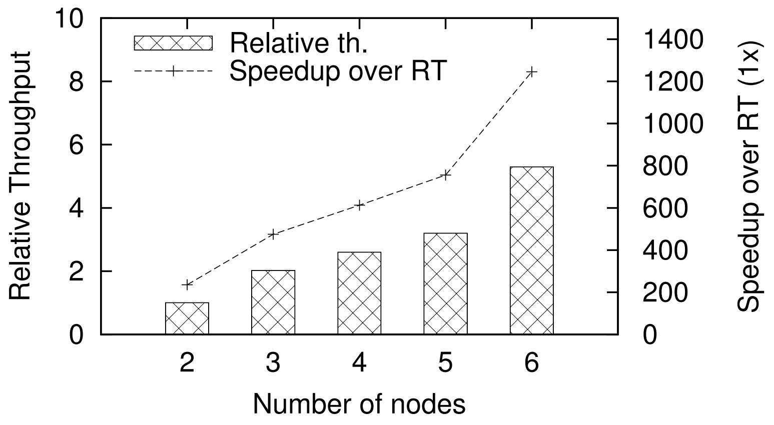

Figure 6.23: Throughput and speedup over RT as a function of the number of machines. We increase the speedup by scaling out the system to aggregate throughput.

	Q1	Distributed Q1	Q2	Distributed Q2
10th	4 ms	3 ms	118 ms	40 ms
50th	17 ms	12 ms	136 ms	150 ms
90th	31 ms	21 ms	160 ms	186 ms

Figure 6.24: Latencies for both queries with baseline and distributed deployment.

The major sources of latency spikes in SEEP are the buffering mechanism used for fault tolerance, and the interaction of this with the garbage collector under high memory utilisation scenarios. Neither of these happen for *Query 1*. Our latencies are slightly lower than in the non-scaled case. The reason for this is that the source cannot insert data at higher rates. Events thus traverse the same number of queues and processing elements as in the non-scaled case but with more headroom.

Query 2 Results

Our implementation of *Query 2* consists of three TEs, *filter*, *Q2 Plug* and *Q2 Global* (see Fig. 6.21). Hence, the baseline deployment comprises 3 nodes in the cluster. For the distributed deployment, we scale out from 3 to 7 nodes.

Baseline deployment. Fig. 6.25 shows the expected behaviour: speedup decreases as the dataset grows in event size. This query is computationally more expensive than *Query 1*. In our solution, this translates to a total time of 3.2 hours to process one month of data for 10 houses to 13 hours to do the same for 40 houses.

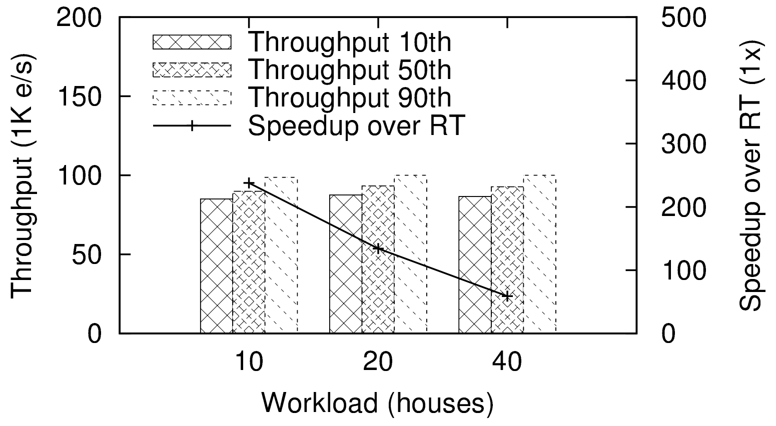

Figure 6.25: Throughput and speedup as a function of the number of houses for query 2.

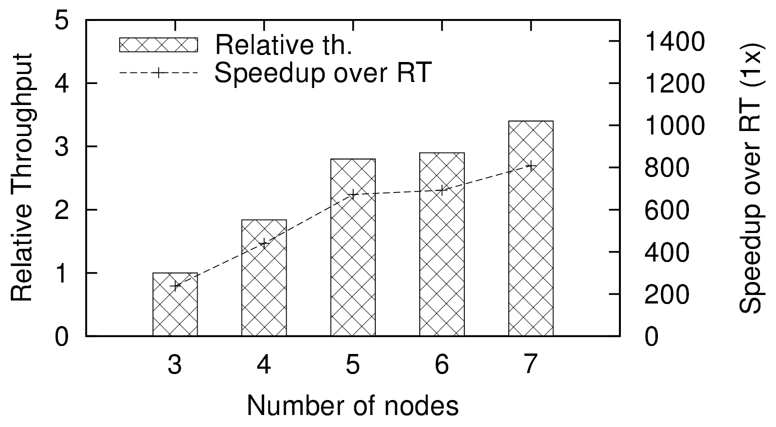

Figure 6.26: Distributed deployment of query 2. Scaling out the query aggregates throughput and increases the speedup.

Distributed deployment. We follow the same strategy of scaling out the system to increase the speedup over the minimum throughput required by the system, reported in Fig. 6.26. When adding more cluster nodes, the throughput increases, except between 5 and 6 nodes. The reason for this behaviour is that there were two simultaneous bottlenecks: a CPU bottleneck, which disappears after scaling from 5 to 6 nodes, gives rise to an I/O bottleneck. After scaling out the I/O bottleneck, the speedup can keep increasing. We stop our query when the source becomes a bottleneck.

The latencies for *Query 2* are reported in Table 6.24. They are higher than for *Query 1* because, while the bottleneck in query 1 is I/O (serialisation and deserialisation), this query is CPU-bound.

Impact of semantic load-shedding. To investigate the inherent trade-off of result ac-

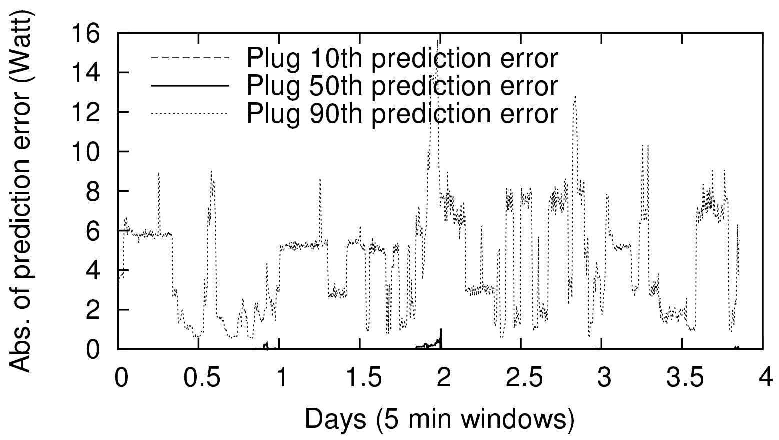

Figure 6.27: Time series over 4 days showing a low prediction error for plugs over 5-minute windows. Note that the 10th and 50th percentile lines overlap with the x-axis.

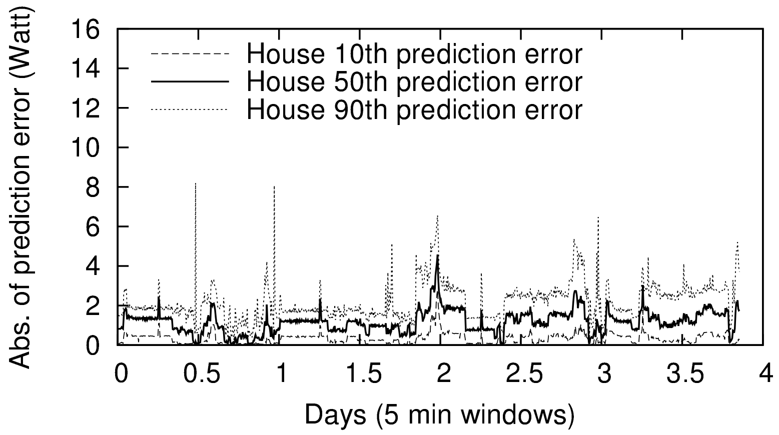

Figure 6.28: Time series over 4 days showing a low prediction error for houses over 5-minute windows.

curacy and computation efficiency implied by semantic load-shedding, we compare the load predictions derived by *Query 1* for a sample of 4 days. We focus on the predictions derived for the smallest time window (one minute). This window represents the most challenging case because, for larger windows, the relative importance of filtered events is smaller and thus accuracy is less affected.

In Fig. 6.27 and 6.28, we show the absolute prediction error for plugs and houses, respectively, aggregated for windows of 5 minutes. For individual plugs, although the 90th percentile shows spikes up to 15 watt, the median error is zero in virtually all cases. For load prediction of houses, in turn, the median error is largely between 1 and 3 watts and there is little variability in the results. Based on these results, we conclude that the error is small enough to justify the use of the mechanism.

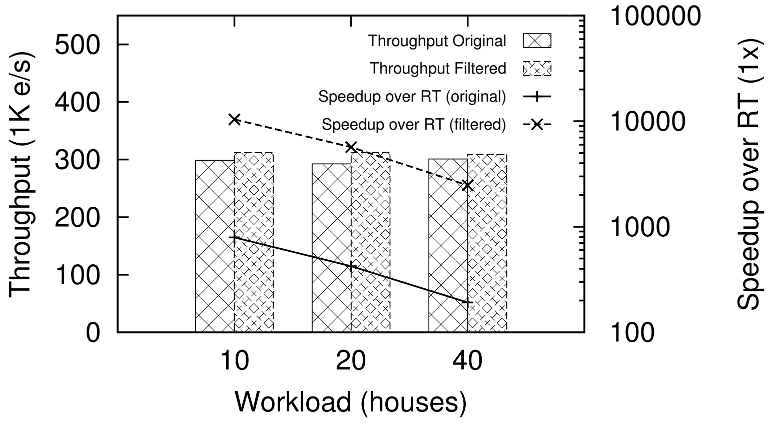

Figure 6.29: When applying semantic load-shedding, the speedup grows by two orders of magnitude (note the logarithmic scale). One month worth of data for 40 houses is processed in 17 minutes.

Regarding the benefits of load-shedding for processing performance, Fig. 6.29 shows the difference in throughput and speedup over RT when enabling the mechanism. As discussed before, the throughput per node is mostly unaffected because processing cost per event is close to constant. However, we observe an improvement of speedup of two orders of magnitude, meaning that one month worth of data for 40 houses is processed in 17 minutes. This drastic speedup together with the low accuracy loss justifies the usage of semantic load-shedding in this scenario—it results in more headroom to scale out the system to accommodate the load from more houses.

6.4.4 Summary

We presented a solution to Smart Grid analytics problem based on stateful data-parallel processing. This solution demonstrates benefits of Stateful Dataflow Graphs: by exploiting the explicit state representation, we could implement efficient data structures that lead to a high throughput, low latency and a fault tolerant solution.

6.5 Summary

In this chapter, we evaluated the idea of stateful data-parallel processing. We implemented challenging applications that demand both high-throughput and low-latency results. We used iterative machine learning applications to show how SDG can also process batch-oriented applications with good performance.

To contextualise the results, we compared with several state of the art systems such as Naiad and Spark. We showed how SDG can maintain high-throughput even with low latencies. This is particularly the case when applications access state with fine-granularity, a situation in which the performance of other systems suffer.

We also described some of the applications of stateful data-parallel processing. First in the context of a smart grid analytics application, and second as part of a deployment at a data-intensive Internet company, which relies on a stack formed by Apache Kafka and Apache Samza for stateful processing.

```
public class SmartGridAnalytics implements SeepProgram {
 @Partitioned // Counts number of load measur. rx for the current slice .
 Vector<ArrayList<Integer>> valueCountForCurrentSlicePerPlug = new Vector<>();
 @Partitioned // Holds avg load per plug per slice . Partitioned by house_id
 Vector<ArrayList<Integer>> loadAveragePerPlugPerSlice = new Vector<>();
 @Partitioned // First work value for each plug and slice . Partitioned by house_id
 Vector<ArrayList<Float>> workPerPlugPerSlice = new Vector<>();
 Map<Integer,Deque<Float>> valuesOneHourPerPlug = new HashMap<>();
 Map<Integer,Deque<Integer>> timestampsOneHourPerPlug = new HashMap<>();
 Map<Integer,List<Float>> valuesSortedByValueOneHourPerPlug = new HashMap<>();
 Map<Integer, Float> currentMedianOneHourPerPlug = new HashMap<>();
 SkipList valuesSortedByValueOneHour = new SkipList(); //Custom impl
 Map<Integer, List<Integer>> housePlugs = new HashMap<>();
 Map<Integer, Integer> plugHouses = new HashMap<>();
 Map<Integer, Float> shareOfOutlierPlugsOneHourPerHouse = new HashMap<>();

 public boolean filter(long id, int timestamp, float value, int property,
                       int plug_id, int household_id, int house_id) {
  boolean filtered = filterNotNewMeasurements(property, id, timestamp);
  filtered = filterMeasurementsThatDoNotChangeAvg(id, timestamp, value,
                                 plug_id, household_id, house_id, filtered);
  return checkAndSendHeartbeat(timestamp);
 }
 public Result q1(long id, int timestamp, float value, int property,
                  int plug_id, int household_id, int house_id) {
  boolean heartbeat = filter(house_id, id, timestamp,
                             property, household_id, plug_id, value);
  if (heartbeat) {
   if (checkSendUpdates(timestamp)) {
    return new Result(id, timestamp, value, property,
                      plug_id, household_id, house_id);
   }
  }
  useWorkForCorrections(timestamp, household_id, plug_id, value, workPerPlugPerSlice);
  updateLoadAvg(household_id, plug_id, value, loadAveragePerPlugPerSlice,
                valueCountForCurrentSlicePerPlug);
 }
 public Result q2(long id, int timestamp, float value, int property,
                  int plug_id, int household_id, int house_id) {
  filter(house_id, id, timestamp, property, household_id, plug_id, value);
  handleOneHourWindow(valuesOneHourPerPlug, timestampsOneHourPerPlug,
                      valuesSortedByValueOneHour, timestamp, value);
  insertValueIntoCurrentWindow(value, valuesSortedByValueOneHour);
  float median = getMedian(valuesSortedByValueOneHour);
  float outliers = checkOutliers(house_id, median, plug_id,
                                 shareOfOutliersPlugsOneHourPerHouse);
  return new Result(id, timestamp, outliers, property,
                    plug_id, household_id, house_id);
 }
 @Override
 public SeepProgramConfiguration configure(){
   SeepProgramConfiguration spc = new SeepProgramConfiguration();
   Schema sch = SchemaBuilder.getInstance()
     .newField(Type.LONG, "id")
     .newField(Type.INT, "timestamp")
     .newField(Type.FLOAT, "value")
     .newField(Type.INT, "property")
     .newField(Type.INT, "plug_id")
     .newField(Type.INT, "household_id")
     .newField(Type.INT, "house_id")
     .build();
   DataStore src = new DataStore(DataStoreType.NETWORK);
   DataStore sink = new DataStore(DataStoreType.CONSOLE);
   spc.newWorkflow("q1", srcData, sch, sink, sch);
   spc.newWorkflow("q2", srcData, sch, sink, sch);
   return spc;
 }
}
```

Algorithm 6: Smart Grid queries example

Chapter 7

Conclusion

Data-parallel processing systems were developed by big Internet companies facing a big data analysis problem as a means to address their processing needs. Today, big data technologies are present across many more sectors of society, from healthcare to entertainment, and this trend is only expected to grow in the future: from the *Internet of Things* [DB], which could multiply the current volume of data, to *The Fourth Paradigm* [THT], which pursues to improve how science is done today, we will witness how better big data technology affects our daily lives and the economy at large.

Research on big data encompasses many different disciplines, from systems and database research to machine learning and statistics. The availability of data-parallel processing systems and cheap access to public clouds has enabled domain scientists to benefit from these new technologies. This increased participation of people with different expertise has a twofold advantage: first, systems and database researchers have extended the limits of these systems; second, other domains have benefited from the new scale at which they can perform their experiments cheaply. For example, machine learning research can now be evaluated on large volumes of data, unlike in the past, where computationally expensive algorithms could only be run on small datasets.

There are still many opportunities to improve further the performance and features of data-parallel processing systems. In this thesis, we argued that by facilitating the access to these technologies, advances will speed up. This is the argument of the so-called *democratisation of data*. With more users getting access to large volumes of data, and the promise of the positive impact that this may bring to our society, data processing must not become the limiting factor.

The stateless processing model on which data-parallel processing systems were based restricts an important class of applications that are increasingly demanded by new domains: stateful applications. To facilitate researchers to benefit from data-parallel processing, it is necessary to: (i) permit them to write stateful algorithms concisely; and (ii) offer the same performance and features as other systems for this class of algorithms. Current stateless processing systems do not permit high-performance execution of stateful algorithms that update state at a fine-granularity.

This thesis proposed a new processing model, stateful dataflow graphs (SDGs), which permits the execution of stateful algorithms with high performance. In addition, it proposed a method to translate imperative programs to SDGs for execution. This permits developers to use state explicitly, therefore facilitating the representation of stateful algorithms.

We implemented SEEP, a stateful data-parallel processing system to evaluate the ideas in this thesis. With SDGs it becomes easier to support both batch-processing—optimised for high-performance—and stream-processing—optimised for low latency. Stateful data-parallel processing is then one way of unifying both processing models, while maintaining for both the scalability and fault tolerance properties.

7.1 Thesis Summary

This thesis began outlining the events of the last decade that gave rise to a wide range of data-parallel processing systems. The success of Internet companies that store user-generated data to provide better services to users, plus the rise of utility computing with cloud systems, triggered a wave of innovation that produced a rich ecosystem of data-parallel processing systems for different workloads.

All these systems are designed to be scalable—adding more computational resources should provide a linear throughput improvement. They target shared-nothing architectures, and for that reason they need to be fault-tolerant: failures are common in large scale shared-nothing clusters. In order to facilitate achieving these properties, all these systems provide a stateless processing model. This permits to capture parallelisation opportunities directly from higher-order functions in the code and to abstract away fault tolerance, so that developers do not need to handle it explicitly.

In the background section, we showed the evolution of the processing models of data-parallel processing systems. In the early days of the MapReduce model, the focus was on increasing the expressiveness of dataflow graphs, mainly by means of adding new higher-order functions. Dryad, Spark and Flink are examples of such enhancements. Systems have also improved their efficiency, mostly by focusing on in-memory computation whenever possible. Despite this evolution, all these systems do not permit high-performance execution of stateful algorithms that require fine-grained updates to state. Another consequence of their stateless processing models is that it is hard to write stateful algorithms that require explicit access to state.

To overcome these shortcomings, we introduced stateful data-parallel processing. In particular, we described Stateful Dataflow Graphs (SDG), which are a new dataflow abstraction that explicitly represents the state in the dataflow graph of computation. In order to provide scalable processing, it avoids remote state accesses by introducing two abstractions for distributed mutable state, partitioned and partial.

The new abstraction of stateful dataflow graphs must be scalable and fault tolerant. Additionally, it has to provide low-latency results and be elastic, i.e. so that it can dynamically adapt to workload spikes. Explicit state in the processing model makes achieving such properties challenging. To address it, we make state a first-class citizen. This means that the system knows about the state handled by applications and can manipulate it. For example, it can partition the state when necessary, as well as move it across machines to enable checkpointing for fault tolerance. We present a set of primitives to handle state, and compose the primitives in an integrated approach for scale out and fault tolerance. The integrated approach relies on the notion of a node failing being equivalent to a node scaling out.

Fault tolerance is a challenging problem when state is large, i.e. when it requires more memory than is available in a single machine. We introduce a technique that takes independent checkpoints of the state without blocking the critical data processing path.

This is achieved by capturing all intermediate state updates—those occurring while the checkpoint is being taken—in a data structure and reconciling this with the state once the checkpoint is taken. For low-overhead backup and state recovery, we introduce an n-to-m model that permits to backup state of any machine to n machines and to recover to m machines after a failure, thus avoiding network and disk bottlenecks.

After showing how we can implement a stateful data-parallel processing system that implements SDGs, we focus on the programming model, where the goal is to offer an imperative interface to developers. We presented Java2SDG, which is a tool that translates Java programs to SDGs, for high-performance and low-latency computation. The tool performs static analysis on the Java program provided by users to separate computation from data and state. It then synthesises a SDG that represents the algorithm. When synthesising state, however, the tool requires developer support: whether state is partitioned or partial depends on application semantics. In order to capture this information, we introduce annotations that developers must use in their Java programs to indicate the type of distributed mutable state. In the case of partial state, it is also necessary to indicate what type of access is intended.

We evaluated these ideas as part of SEEP, a stateful data-parallel processing prototype. We implemented different applications on top of the system and compare them with other systems. We show how stateful data-parallel processing achieves competitive throughput and how it can maintain lower latency. Additionally, we evaluate our fault tolerance mechanisms and measure overhead. Finally we present an application of the ideas presented in this thesis in the context of a smart grid analytics applications. Before discussing some future work, we explain next a successful case of stateful data processing in the context of LinkedIn, a data-intensive web company.

7.2 SDG in Practice: Big Data Integration

In this section we describe a real-world application of stateful data-parallel processing to the problem of Big Data Integration. The solution is used in production at LinkedIn [FPK+15], a data-intensive web company, and applies to many other companies with similar data integration problems. Stateful data-parallel processing is implemented in Apache Samza [Apac], a system focused on streaming workloads that runs on top of Apache Kafka [Apab]—a messaging systems that provides reliability and high-performance message communication.

With more sophisticated data-parallel processing systems, the new bottleneck in data-intensive companies shifts from the back-end data systems to the data integration stack, which is responsible for the preprocessing of data for back-end applications. The use of back-end data systems with different access latencies and data integration requirements poses new challenges that current data integration stacks based on distributed file systems—proposed a decade ago for batch-oriented processing—cannot address.

The solution described is a novel alternative to satisfy the data integration needs of modern web companies. It consists of a data integration stack that provides low latency data access to support near real-time in addition to batch applications. It supports incremental processing, and is cost-efficient and highly available. We refer to the stateful data-parallel processing architecture used to solve this problem as Liquid. Liquid has two layers: a **processing layer based on a stateful stream processing model**, implemented in Samza, and a messaging layer with a highly-available publish/subscribe system, implemented by Kafka. SDGs are implemented as part of Samza and Kafka. Samza is responsible for implementing both the TEs and SEs; Kafka provides communication capabilities, (i.e. the dataflows that connect TEs) and works as a repository to store checkpoints produced by Samza. We report the experience of a Liquid deployment with backend data systems at LinkedIn, a company with over 300 million users.

The section is structured as follows. We first start with an overview of the problem of big data integration (§7.2.1). We then describe the SDG implementation on top of Liquid (§7.2.2), and conclude with examples of applications that use SDGs and exemplify the benefits of a stateful processing stack for big data integration (§7.2.3).

7.2.1 The Problem of Data Integration

Web companies such as Google, Facebook and LinkedIn generate value for their users by analysing ever-increasing amounts of data. Higher user-perceived value means better user engagement, which, in turn, generates even more data. While this high volume of append-only data is invaluable for organisations, it becomes expensive to integrate using proprietary, often hard-to-scale data warehouses. Instead, organisations create their own data integration stacks for storing data and serving it to back-end data processing systems. Today's data integration stacks are frequently based on a MapReduce (MR) model [DG04]—they run custom ETL-like MR jobs on commodity shared-nothing clusters with scalable distributed file systems (DFS) such as GFS [GGL03] or HDFS [HDF] in order to produce data for back-end systems [LR13].

With the inflow of data-parallel processing systems facilitating the analysis of large volumes of data, a new bottleneck appears in the data integration stack. Many organisations today use a MR/DFS stack for data integration: the storage layer uses a DFS to store data in a cost-effective way, sharding it over nodes in a cluster; the processing layer executes batch-oriented MR jobs, which clean, normalise and pre-process the data. Such a design has several limitations.

First, intermediate results of MR jobs are written to the DFS, resulting in higher latencies as job pipelines grow. Second, to avoid reprocessing all data after updates, back-end systems must support incremental processing, which requires fine-grained access to the data, not supported by the MR/DFS stacks. To address the above challenges, a common approach is for backend systems to execute on their own duplicate copies of the "source-of-truth" data [TSA$^+$10, OHB$^+$11]. This approach breaks the single "source-of-truth" abstraction and requires handling divergent replicas.

Finally, when applications require low-latency results, current data integration stacks are not a viable option. Stream processing systems are then deployed to feed directly from data sources, overall leading to a more complex and brittle infrastructure.

Current Approaches for Data Integration

As organisations have discovered the limitations of existing data integration approaches, this has led to new architectural patterns:

Lambda architecture [wJW15]. In this pattern, input data is sent to both an offline and an online processing system. Both systems execute the same processing logic and

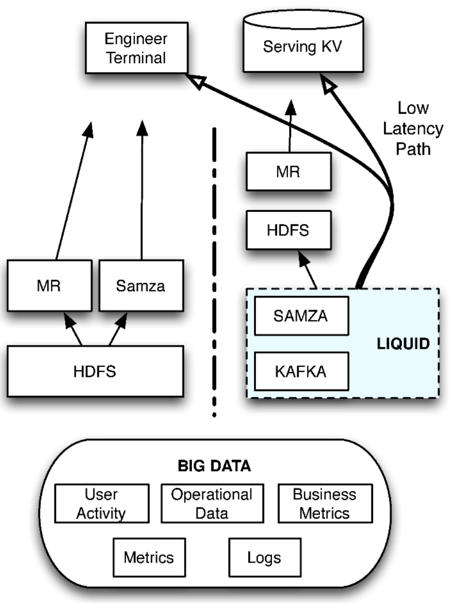

Figure 7.1: Data integration with Liquid.

output results to a service layer. Queries from back-end systems are executed based on the data in the service layer, reconciling the results produced by the offline and online processing systems.

This pattern allows organisations to adapt their current infrastructures to support nearline applications [Amab]. This comes at a cost, though: developers must write, debug, and maintain the same processing code for both the batch and stream layers, and the Lambda architecture increases the hardware footprint.

Kappa architecture [Kre]. In this pattern, a single nearline system, e.g. a stream processing platform, processes the input data. To reprocess data, a new job starts in parallel to an existing one. It reprocesses the data from scratch and outputs the results to a service layer. After the job has finished, back-end systems read the data loaded by the new job from the service layer. This approach only requires a single processing path, but it has a higher storage footprint, and applications access stale data while the system is reprocessing data.

Implementing the above architectural patterns in current MR/DFS-based data integration stacks introduces a range of problems, including an increased hardware footprint, data and processing duplication—that must be handled for consistency—and more complex management. Next we summarise the requirements of a modern big data integra-

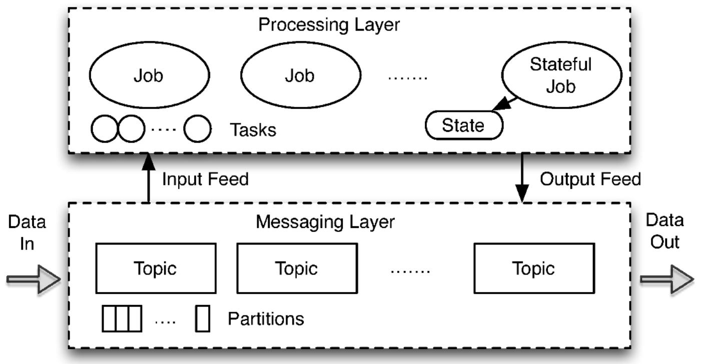

Figure 7.2: Architecture of the Liquid data integration stack.

tion stack.

7.2.2 Big Data Integration Based on Stateful Data-Parallel Processing

The stateful data-parallel processing properties of low-latency processing, fault tolerance and support for arbitrary state make it suitable to address the challenges of big data integration. Instead of relying on the MR/DFS model for data integration shown on the left of Fig. 7.1, Liquid presents a new architecture based on stateful data-parallel processing. SDGs are implemented in Liquid on top of two independent yet cooperating layers. The stateful processing layer (i) executes ETL-like jobs, potentially with state, for different back-end systems; (ii) provides low-latency results; and (iii) enables incremental processing. This layer implements the TEs and SEs of the SDG model. A messaging layer complements the stateful processing layer by (i) storing high-volume data that is kept with high availability; and (ii) providing a high-performance communication service, used to implement the dataflows of the SDG model.

The two layers communicate by writing and reading data to and from two types of feeds, stored in the messaging layer (see Fig. 7.2): *source-of-truth feeds* represent primary data, i.e. data that is not generated within the system; and *derived data feeds* contain results from processed source-of-truth feeds or other derived feeds. Derived feeds contain lineage information, i.e. annotations about how the data was computed, which are stored by the messaging layer. The processing layer must be able to access data according to different annotations, e.g. by timestamp. It also produces such annotations when writing data to the messaging layer.

Back-end systems read data from the input feeds, after Liquid has preprocessed them to meet application-specific requirements. These jobs are executed by the processing layer, which reads data from input feeds and outputs processed data to new output feeds.

The implementation of SDGs in two layers is an important design decision. By keeping both layers separated, producers and consumers can be decoupled completely, i.e. a job at the processing layer can consume from a feed more slowly than the rate at which another job published the data without affecting each other's performance. In addition, the separation improves the operational characteristics of the data integration stack in a large organisation, particularly when it is developed and operated by independent teams: separation of concerns allows for management flexibility, and each layer can evolve without affecting the other.

Messaging layer. The messaging layer is based on a topic-based publish/subscribe communication model. This model is appropriate because it abstracts data delivery, which makes it easier to offer it as a service—with advantages from an operational point of view. With this model, connecting TEs in the SDG only requires to agree on the topics to which each pair of TEs subscribes and publishes. Data is published to and consumed from *brokers* in the messaging layer that handle data delivery. For high throughput reads and writes, each dataflow in the SDG model is organised as multiple *partitions* of a topic in the messaging layer.

Clients of the messaging layer, i.e. TEs, use offsets included in the tuples to keep track of the latest consumed data per partition. TEs pull data from brokers by providing a set of offsets. After a pull request, brokers return the latest data after the specified offsets. This approach makes it efficient to maintain the latest consumed data, i.e. it only requires to store a single integer per partition.

Metadata-based access. The messaging layer uses a highly-available, logically-centralised offset manager to maintain annotations on the data, which can be queried by clients. For example, TEs can checkpoint their last consumed offsets to save their progress; after failure, they can ask for the last data that they processed. To reprocess data, TEs can include metadata, such as timestamps, with the offsets and retrieve data according to these previously-stored timestamps.

Processing layer. As shown in Fig. 7.2, a TE embodies computation over streams, provided by the messaging layer. Input data coming from feeds in the messaging layer is processed by the TEs, that can optionally update some SEs and output results back to the messaging layer.

For parallel processing, a TE can be partitioned and each instance consumes data from a different partition of a topic. Stateless TEs get all their input data from the input stream, while a stateful TE has explicit access to a SE that evolves as part of the computation. For fault tolerance, checkpoints are stored in the form of a replayable changelog that is maintained with high availability by the messaging layer.

Incremental processing. The stateful processing layer can process data incrementally by exploiting explicit state and the functionality of the offset manager. A TE can periodically checkpoint the offsets that it has consumed and maintain a summary of the input data as part of its SE. When new input data becomes available, the TE can thus ignore already processed data. For scenarios where data changes frequently, this incremental processing capability is highly valuable.

7.2.3 Evaluation

The SDG-based solution for data integration presented in this chapter is pervasive across the back-end systems at LinkedIn. The messaging layer, based on Apache Kafka, runs in 5 co-location centres, spanning different geographical areas. It ingests over 50 TB of input data and produces over 250 TB of output data daily (including replication). For this, it uses around 30 different clusters, comprised of 300 machines in total that host over 25,000 topics and 200,000 partitions. The stateful processing layer, based on Apache Samza, spans across 8 clusters with over 60 machines. Overall, Liquid is deployed on more than 400 machines that perform data integration and adaptation for back-end and front-end systems.

Data cleaning and normalisation. A crucial task in many organisations is to clean and normalise user-generated content. This is typically done by specialised algorithms that, e.g. disambiguate entities or detect synonyms in text data. To achieve best results, algorithms must operate on the latest content, which is challenging because (i) users continuously generate new content; and (ii) engineers continuously optimise their processing algorithms.

These two challenges require different system properties: when users generate new content, the cleaning pipeline must have low-latency, so that new information is incorporated quickly, e.g. appearing when users search the website; when the source code of the cleaning pipeline changes, it is necessary to reprocess data with the new algorithm so that all data was cleaned with the same algorithm.

Before the deployment of Liquid, there were two different sub-systems for data cleaning, one for the nearline case (i.e. new content from users) and another for the batch case (i.e. changes in the pipeline code). This meant that each time that new cleaning code was written, it had to be tested against both cases, which was time-consuming and error-prone. Even worse, these sub-systems were shared by different teams, making resource isolation impossible: bugs in one sub-system affected the other.

The use of a stack based on stateful processing brought several benefits: it achieved (i) more efficient re-processing, i.e. it is now easier to integrate the latest user-generated data with current results by using SEs, or to clean past data with new algorithms; and (ii) lower data access latency, which allows back-end systems to serve freshly cleaned data.

Site speed monitoring. To improve the user experience, web companies monitor the page loading times by tracking client-generated events, often referred to as real user monitoring (RUM). Events are stored first and analysed later to detect anomalies and performance problems in the loading times. A fundamental issue with this approach is that problems are not detected promptly, which prevents corrective actions to be issued in real-time. For example, if the root cause of a page loading problem is quickly isolated to a particular CDN, traffic can be re-routed to different servers.

With the new stack, when a client visits a webpage, an event is created that contains a timestamp, the page or resource loaded, the time that it took to load, the IP address location of the requesting client and the content delivery network (CDN) used to serve the resource. These events are consumed by an SDG, which groups them by location, CDN, or other dimensions.

Based on this data, Liquid can feed back-end applications that detect anomalies: e.g. CDNs that are performing particularly slowly, or increased loading times from specific client locations. Back-end applications can consume already preprocessed data that divides user events per session. As a result, back-end applications can detect anomalies within minutes as opposed to hours, permitting a rapid response to incidents. All of this is possible because TEs in the SDG of the application can keep arbitrary state in their SEs.

Call graph assembly. At LinkedIn, dynamic web pages are built from thousands of REST calls, which are executed by distributed machines. Each call can subsequently trigger other calls, and the responses of all these calls constitute the generated web page. This makes it important to detect slow calls, which indicate problems with a particular service.

Before the deployment of the new data integration stack, the usual procedure was to analyse all logs after they were stored in the DFS, i.e. a batch job constructed a call graph hours after an incident was logged. SDGs—with their implementation on top of Liquid—enabled to move such processing earlier in the pipeline reducing latency and identifying potential problems within seconds rather than hours. Other organisations use purpose-built systems for this task, such as Dapper [SBB+10] at Google, or Zipkin [Zip] at Twitter.

The call graph assembly is an SDG running on top of Liquid. Liquid records each event produced by the REST calls and stores them in the messaging layer with a unique identifier per user call, as assigned by the front-end system, i.e. all REST calls for a given request share the same identifier. The processing layer processes these events to assemble the call graph. The call graph is used in production to monitor the site in real-time, and to inform capacity planning decisions.

Operational analysis. Analysing operational data, such as metrics, alerts and logs, is crucial to react to potential problems quickly. Not only malfunctioning software or physical machines but also fraud attempts require prompt action. The volume of data grows with the number of monitoring metrics and logs, and increases due to new features and hardware resources. Previously all this data was stored in the DFS, which meant that it was retrieved and analysed only after a problem was detected.

At LinkedIn, an internal service running several different SDGs presents a range of business, operational and user metrics as visualisations that help different teams understand the current infrastructure status. With the new stack, integrating new data, such as crash reports from mobile phones, is straightforward: all data is transported by the messaging layer, which only needs to produce a new metric. The stateful processing layer helps prepare data for visualisations and provides aggregate values, to facilitate analysis.

7.3 Future Work

During the writing of this thesis, we identified a number of topics for future work.

Convergence of data serving and analytics. A use case that stateful data-parallel processing enables is to merge computation that requires low-latency and computation that requires high-performance. One such example is the online collaborative filtering use case presented in this thesis. Typically, data-parallel processing systems would be used to perform the analytics side of the process, and some other logic would make those results available for serving. Stateful data-parallel processing can provide an alternative to unify both.

Materialise vs schedule tasks. We take a particular stance in this thesis regarding our task placement model, and always materialise tasks. The reason for this decision is to support streaming workloads, for which materialised tasks are better suited. However, we aim to revisit our decision of only supporting materialising tasks: a scheduled system may be more efficient in pure batch-oriented scenarios and provide some additional flexibility when deploying the system with resource managers such as Mesos [HKZ+11] or YARN [VMD+13].

Elasticity within resource managers. We proposed the first elastic and fault tolerant stateful data-parallel processing system in a cluster environment. We focused on public clouds, where elasticity brings important cost benefits. With the adoption of resource managers to support multi-tenancy in clusters, we aim to revisit our ideas in this new context. Resource managers control the amount of resources granted to applications and can dynamically change such decisions. However, only applications know precisely when they require more resources, i.e. because they detect a load spike, for example. The interactions between both systems are intertwined, and we plan to investigate how we can coordinate them for correct and efficient operation.

Imperative programming model. The Java-based programming model that we presented in this thesis is a first step towards an interface that would facilitate writing stateful programs. A limitation of our approach is that we require users to explicitly provide a *merge* function when they perform global reads on partial state. This function is orthogonal to the algorithm's logic. We plan to investigate other data structures, such as CRDTs that can merge results implicitly, as well as incorporate static analysis to understand to what extent this problem can be alleviated.

Bibliography

[AAA$^+$] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H.V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Re, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The Beckman Report on Database Research. `http://beckman.cs.wisc.edu`. Online; accessed 23 February 2015.

[AAB$^+$05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *In CIDR*, pages 277–289, 2005.

[ABB$^+$03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[ABB$^+$13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[ABE$^+$14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[ACc$^+$03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, August 2003.

[ACG$^+$04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[Amaa] Amazon. Amazon web services (aws). `http://aws.amazon.com`. Online; accessed 23 February 2015.

[Amab] Amazon. AWS Lambda. `http://aws.amazon.com/lambda/`. Online; accessed 8 July 2015.

[Apaa] Apache. Apache Flink. `https://flink.apache.org`. Online; accessed 8 July 2015.

[Apab] Apache. Apache Kafka. `http://kafka.apache.org/`. Online; accessed 30 September 2015.

[Apac] Apache. Apache Samza. `http://samza.apache.org`. Online; accessed 23 February 2015.

[App] Apple. Siri. `https://www.apple.com/ios/siri/`. Online; accessed 2 August 2015.

[AXL$^+$15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[BBC$^+$02] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov 2002.

[BEH$^+$10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[BEKS14] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *CoRR*, abs/1411.1607, 2014.

[BGAH07] Roger Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, January 2007.

[BGK$^+$11] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, New York, NY, USA, 2011. ACM.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[BHS09] Magdalena Balazinska, Jeong-Hyon Hwang, and Mehul A. Shah. Fault Tolerance and High Availability in Data Stream Management Systems. In *Encyclopaedia of Database Systems*, 2009.

[Bir85] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[BPASP11] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient Processing of Data Warehousing Queries in a Split Execution Environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1165–1176, New York, NY, USA, 2011. ACM.

[CCD$^+$03a] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[CCD$^+$03b] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[CEGNY03] Bill Carlson, Tarek El-Ghazawi, B Numrich, and Kathy Yelick. Programming in the partitioned global address space model. *Tutorial at Supercomputing*, 2003.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[CGS$^+$05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[CHKS14] Merce Crosas, James Honaker, Gary King, and Latanya Sweeney. Automating Open Science for Big Data. *ANNALS of the American Academy of Political and Social Science*, 2014.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.

[CK09] Martin Campbell-Kelly. Historical Reflections: The Rise, Fall, and Resurrection of Software As a Service. *Commun. ACM*, 52(5):28–30, May 2009.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flume-Java: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[CWM+14] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, and Milind Bhandarkar. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1223–1234, New York, NY, USA, 2014. ACM.

[DB] Wired Magazine Daniel Burrus. The internet of things is far bigger than anyone realizes. `http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/`. Online; accessed 15 August 2015.

[DG92] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, June 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DGG+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[DLS$^+$09] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[DRAT11] Michael Duller, Jan S. Rellermeyer, Gustavo Alonso, and Nesime Tatbul. Virtualizing stream processing. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'11, pages 269–288, Berlin, Heidelberg, 2011. Springer-Verlag.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[Eco] The Economist. Data, data everywhere. `http://www.economist.com/node/15557443`. Online; accessed 2 August 2015.

[ELZ$^+$10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[EST$^+$13] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, and Volker Markl. Iterative Parallel Data Processing with Stratosphere: An Inside Look. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1053–1056, New York, NY, USA, 2013. ACM.

[FMKP14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit for Imperative Big Data Processing.

In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 49–60, Berkeley, CA, USA, 2014. USENIX Association.

[FP] Raul Castro Fernandez and Peter Pietzuch. Towards Low-Latency and In-Memory Large-Scale Data Processing. http://www.orgs.ttu.edu/debs2013/doc_submissions/debs2013_submission_157.pdf. Online; accessed 23 February 2015.

[FPK$^+$15] Raul Castro Fernandez, Peter Pietzuch, Joel Koshy, Jay Kreps, Dong Lin, Neha Narkhede, Jun Rao, Chris Riccomini, and Guozhang Wang. Liquid: Unifying Nearline and Offline Big Data Integration. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, CIDR'15, Asilomar, CA, USA, 2015.

[FWPG14a] Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch, and Avigdor Gal. Scalable Stateful Stream Processing for Smart Grids. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 276–281, New York, NY, USA, 2014. ACM.

[FWPG14b] Raul Castro Fernandez, Matthias Weidlich, Peter Pietzuch, and Avigdor Gal. Scalable Stateful Stream Processing for Smart Grids. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 276–281, New York, NY, USA, 2014. ACM.

[GAT$^+$15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1917–1923, New York, NY, USA, 2015. ACM.

[GAW$^+$08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[Gei94] Al Geist. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[Gel89] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, PARLE '89, pages 20–27, London, UK, UK, 1989. Springer-Verlag.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GHMP08] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[GJN11] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[Goo] Google. Google now. `https://www.google.com/landing/now/`. Online; accessed 2 August 2015.

[Gri08] Robert Griesemer. Parallelism by Design: Data Analysis with Sawzall. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 3–3, New York, NY, USA, 2008. ACM.

[GS97] Rachid Guerraoui and Andre; Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[HB09] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[HCG$^+$14] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

[HDF] HDFS. Hadoop Distributed File System. `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Online; accessed 8 July 2015.

[HDF11] Kai Hwang, Jack Dongarra, and Geoffrey C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[hkb] hkblok. Historial cost of computer memory and storage. `http://hblok.net/blog/storage/`. Online; accessed August 15, 2015.

[HKZ$^+$11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HTT09] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[HXCZ07] Jeong-Hyon Hwang, Ying Xing, U. Cetintemel, and S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing.

In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 176–185, April 2007.

[IBM] IBM. Infosphere streams 4.0. `http://www-03.ibm.com/software/products/en/infosphere-streams`. Online; accessed 23 August 2015.

[IBY^{+}07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[JAA^{+}06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, NY, USA, 2006. ACM.

[JBo] JBoss. Javaassist. `http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/`. Online; accessed 15 August 2015.

[JGL^{+}14] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big Data and Its Technical Challenges. *Commun. ACM*, 57(7):86–94, July 2014.

[KBB^{+}15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, CIDR'15, Asilomar, CA, USA, 2015.

[KC03] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan 2003.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[KEGM10] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order Finite-element Seismic Wave Propagation Modeling with MPI on a Large GPU Cluster. *J. Comput. Phys.*, 229(20):7692–7714, October 2010.

[Kre] Jay Kreps. Kappa Architecture. `http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html`. Online; accessed 8 July 2015.

[KTGN10] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 94–103, May 2010.

[LR13] Jimmy Lin and Dmitriy Ryaboy. Scaling Big Data Mining Infrastructure: The Twitter Experience. *SIGKDD Explor. Newsl.*, 14(2):6–19, April 2013.

[LYYZ10] Wen-Syan Li, Jianfeng Yan, Ying Yan, and Jin Zhang. Xbase: Cloud-enabled information appliance for healthcare. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 675–680, New York, NY, USA, 2010. ACM.

[LZR06] Bin Liu, Yali Zhu, and Elke Rundensteiner. Run-time Operator State Spilling for Memory Intensive Long-running Queries. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 347–358, New York, NY, USA, 2006. ACM.

[MAAC14] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic Parallelization for Multi-core and the Cloud. In *Proceedings of the 11th USENIX Conference on Operating Systems Design* *and Implementation*, OSDI'14, pages 645–659, Berkeley, CA, USA, 2014. USENIX Association.

[MBF14] André Martin, Andrey Brito, and Christof Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 198–205, New York, NY, USA, 2014. ACM.

[Mic] Microsoft. Cortana. `http://www.microsoft.com/en-us/mobile/campaign-cortana/`. Online; accessed 2 August 2015.

[MLSL04] Alan L. Montgomery, Shibo Li, Kannan Srinivasan, and John C. Liechty. Modeling online browsing and path analysis using clickstream data. *Marketing Science*, 23(4):579–595, 2004.

[MLV+04] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson. Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 277–286, Washington, DC, USA, 2004. IEEE Computer Society.

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[MPL12] Christopher Mitchell, Russell Power, and Jinyang Li. Oolong: Asynchronous Distributed Applications Made Easy. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 11:1–11:6, New York, NY, USA, 2012. ACM.

[MSS+11] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[MSSV09] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying Suspicious URLs: An Application of Large-scale Online Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 681–688, New York, NY, USA, 2009. ACM.

[Nat] Nature. Biology: The big challenges of big data. `http://www.nature.com/nature/journal/v498/n7453/full/498255a.html`. Online; accessed 17 August 2015.

[Net] Netflix. `https://en.wikipedia.org/wiki/Netflix_Prize`. Online; accessed 23 August 2015.

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[NRNK10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, Dec 2010.

[OHB$^+$11] Fatma Özcan, David Hoa, Kevin S. Beyer, Andrey Balmin, Chuan Jie Liu, and Yu Li. Emerging Trends in the Enterprise Data Analytics: Connecting Hadoop and DB2 Warehouse. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1161–1164, New York, NY, USA, 2011. ACM.

[ORS$^+$08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[OWZS13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[PL10] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[PLS+06] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[PWB07] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[Rac] Rackspace. Rackspace. `http://www.rackspace.com`. Online; accessed 23 February 2015).

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[Sav14] Neil Savage. Bioinformatics: Big Data Versus the Big C. *Nature*, 2014.

[SBB+10] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[Sch97] Robert R. Schaller. Moore's Law: Past, Present, and Future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[Sch10] Bertil Schmidt. *Bioinformatics: High Performance Parallel Computer Architectures*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[SKAEMW13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.

[soo] The Soot framework for Java program analysis: a retrospective. Online: accessed 23 February 2015).

[ss] JavaSpaces service specification. `https://river.apache.org/doc/specs/html/js-spec.html`. Online; accessed 15 August 2015.

[SSOG93] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[Sto86] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[Str] Microsoft StreamInsight. `https://technet.microsoft.com/en-us/library/ee362541(v=sql.111).aspx`. Online; accessed 23 August 2015.

[THT] Stewart Tansley Tony Hey and Kristin Tolle. The fourth paradigm. `http://research.microsoft.com/en-us/collaboration/fourthparadigm/`. Online; accessed 15 August 2015.

[Tib] Tibco. `http://www.tibco.com/products/automation/in-memory-computing/transactional-application-server/activespaces-transactions`. Online; accessed 15 August 2015.

[TSA+10] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the*

2010 ACM SIGMOD International Conference on Management of Data, SIGMOD ’10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[TSF90] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A taxonomy-based comparison of several distributed shared memory systems. *SIGOPS Oper. Syst. Rev.*, 24(3):40–67, July 1990.

[TTS$^+$14a] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’14, pages 147–156, New York, NY, USA, 2014. ACM.

[TTS$^+$14b] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’14, pages 147–156, New York, NY, USA, 2014. ACM.

[VMD$^+$13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC ’13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[VRDB10] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’10, pages 389–400, New York, NY, USA, 2010. ACM.

[WESM$^+$10] Fei Wang, Vuk Ercegovac, Tanveer Syeda-Mahmood, Akintayo Holder, Eugene Shekita, David Beymer, and Lin Hao Xu. Large-scale Multimodal Mining for Healthcare with Mapreduce. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI '10, pages 479–483, New York, NY, USA, 2010. ACM.

[wJW15] Nathan Marz with James Warren. *Big Data Principles and best practices of scalable realtime data systems*. Manning Publications Co, 2015.

[WT15] Yingjun Wu and Kian-Lee Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 723–734, April 2015.

[XRZ$^+$13] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[YIF$^+$08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[ZBSS$^+$10] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[ZCD$^+$12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference* *on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZDL^{+}13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[Zip] Zipkin. Twitter Zipkin. `https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin`. Online; accessed 8 July 2015.

[ZKJ^{+}08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[ZR11] Erik Zeitler and Tore Risch. Massive Scale-out of Expensive Continuous Queries. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 37*, VLDB. VLDB Endowment, 2011.

[ZSK04] Gengbin Zheng, Lixia Shi, and L. V. Kale. FTC-Charm++: An In-memory Checkpoint-based Fault Tolerant Runtime for Charm++ and MPI. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 93–103, Washington, DC, USA, 2004. IEEE Computer Society.

[ZTKL14] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 465–477, Broomfield, CO, October 2014. USENIX Association.