University of Zurich[UZH]

Master Thesis August 19, 2014

Investigating the Lambda Architecture

Nicolas Bär
of Zürich ZH, Switzerland

Student-ID: 08-857-195
nicolas.baer@gmail.com

Advisor: **Dr. Thomas Scharrenbach**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank Prof. Abraham Bernstein for giving me the opportunity to write this thesis at the Dynamic and Distributed Information Systems group and for providing the necessary resources to conduct the experiments. Special thanks go to Dr. Thomas Scharrenbach for the valuable feedback and great discussions throughout the course of this thesis. I would like to thank the S3IT group for providing me resources to conduct pre-studies. And special thanks go to Dr. Jakob Bär for proofreading this thesis.

Zusammenfassung

Die fortschreitende Integration von Informationssystemen stellt Systeme zur Echtzeitanalyse grosser Datenmengen zunehmend vor Herausforderungen: Einerseits sollen die Ergebnisse möglichst präzise sein, andererseits sollen die Daten rasch verarbeitet werden und zur Verfügung stehen. Einen neuen Lösungsansatz zur Bewältigung der entstehenden Probleme stellt die von Marz skizzierte Lambda-Architektur dar, zu der bisher allerdings noch keine Referenzimplementierung publiziert wurde.

Die vorliegende Arbeit stellt eine mögliche Umsetzung dieser Architektur auf der Basis von Open-Source-Software Komponenten vor. Die Grundlage des Batch-Layers bildet dabei ein skalierbarer inkrementeller Mechanismus, der eingehende Nachrichten repliziert ablegt und Operationen wiederholen kann, falls Fehler auftreten. Der verteilte Speed-Layer hingegen verwirft unverarbeitete Nachrichten, falls unerwartete Fehler auftreten, damit neue Nachrichten schneller verarbeiteten werden können. Die Architektur verspricht "eventual accuracy": Die allenfalls fehlerhaften Echtzeit-Resultate des Speed-Layers können durch die präzisen Ergebnisse des Batch-Layers ersetzt werden.

Die vorliegende Arbeit präsentiert auch die Ergebnisse der Evaluation des vorgeschlagenen Designs mit den Datensätzen des SRBench Benchmarks und der DEBS Grand Challenge 2014. Aufgezeigt wird das Verhalten der Architektur und deren Leistungsfähigkeit bei Instabilität der Infrastruktur und unter variierenden Datenfrequenzen.

Abstract

Information systems become increasingly integrated and cause new challenges to provide real-time analytics based on a high volume of data. The concept of the lambda architecture proposed by Marz provides a new solution to this problem, but the lack of a reference implementation limits its analysis.

This thesis presents a possible implementation of the lambda architecture based on open source software components. The design of the batch layer is based on a scalable incremental mechanism that stores incoming data in a distributed and highly available storage engine, which provides replay functionality in case of failures. The speed layer does not provide recovery mechanisms and in case of machine failures the speed layer drops messages and continues with the most recent data available. The architecture guarantees eventual accuracy, which provides the possibly inaccurate results of the speed layer in real-time and replaces these values with the accurate results of the batch layer.

The evaluation of the designed architecture measured its capabilities based on the SRBench Benchmark and DEBS Grand Challenge 2014 task and stressed its behavior with varying data frequency rates on an unreliable infrastructure.

Table of Contents

1

Introduction

There is a wide range of applications in which data is consumed via streams. In most cases this involves an external environment to generate data and push this data asynchronously to stream processing systems. These systems compute results based on the continuous data streams in a time-discrete manner. Stream processing requirements are found in the business and scientific domain. Examples include financial markets, surveillance, manufacturing, healthcare, infrastructure monitoring, radio astronomy, etc. [5].

The data frequency of streams depends on the problem an application solves and may range from a few to millions of data items per second. Use cases enforce different quality of service constraints regarding the response time of stream-based applications. For example a reactive use case with high-volume data streams may require an answer in a timely fashion. In such a scenario, complex computation processes have to be distributed and in case of data loss, for example through communication or hardware failures, the QoS constraints allow for very limited fault recovery due to response time restrictions. Other use cases involve QoS constraints regarding the precision of the results. In such a scenario data loss is not acceptable and response time is traded for the sake of obtaining precise and complete results.

The lambda architecture introduced by Marz [44] is an interesting proposal to the latency challenges in real-time stream processing. The architecture proposal decomposes the problem into three layers: (i) the batch layer focuses on fault tolerance and optimizes for precise results (ii) the speed layer is optimized for short response-times and only takes into account the most recent data and (iii) the serving layer provides low latency views to the results of the batch layer.

The reason to divide the architecture into three layers is the flexibility it offers to the potential applications. The fast but possibly inaccurate results of the speed layer are eventually replaced by the precise results of the batch layer.

1.1 Motivation

The purpose of this thesis is to investigate the lambda architecture in the context of purely stream-based applications and measure its effect regarding different QoS metrics. The batch and the speed layer consume from the same stream, but vary in their requirements towards response times and fault-tolerance.

Very limited related work to the lambda architecture is available and a reference implementation has not been published yet. This thesis includes a design based on open source software of the batch layer following an incremental approach and of the speed layer with reduced fault-recovery guarantees. It is then used to generate key performance indicators in order to measure the behavior of the system with different use cases such as the SRBench benchmark and DEBS Grand Challenge 2014 task.

A proof of concept includes the simulation of failures in the underlying system to check its fault-recovery behavior. Different frequencies and bursts in the data are simulated to evaluate the key performance indicators of the architecture.

1.2 Outline

Chapter 2 introduces the concepts of stream and batch processing and highlights the recent research in these areas with regard to the lambda architecture. Chapter 3 discusses the related work and further positions the research question of this thesis within its field. The designed architecture is presented in Chapter 4 that includes an introduction of frameworks that forms the basis of the design and presents the proposed solution for the speed and batch layer. The setup of the experiments conducted to qualify and quantify the performance of the architecture design is described in Chapter 5. The results of the experiments are then reported in Chapter 6 and discussed in Chapter 7. Chapter 8 highlights the limitation of the designed architecture and in Chapter 9 possible future work is listed. Finally, the conclusion of this work is presented in Chapter 10.

2

Preliminaries

The lambda architecture is a new concept and very limited related work is available. It includes techniques and methods from the stream and batch processing area that are described as follows. First, a brief introduction in stream processing systems is provided and the relevant work regarding the batch and speed layer is highlighted. Second, the recent work in the area of batch processing is introduced.

2.1 Stream Processing

Andrade et al. [5] describe the development of stream processing systems as the result of continuous evolution in the technology of managing high volumes of data. In particular advances in data warehousing and scalable data processing solutions lead to a higher degree of system interconnections and the emerging need for strongly integrated and fast data analysis. Distributed systems and their possibility to scale horizontally are the building blocks for stream processing and provide integration with other technologies and systems to combine data sources from a heterogeneous landscape of software components. The synergies of an interconnected software landscape and the progress in the area of data mining and business intelligence ultimately contributed to the needs to process information in a real-time, stream based manner. Cugola and Margara [22] further describe stream processing systems as an evolution of database oriented data processing methods. Traditional database management systems require data to be persistently stored and indexed before it can be processed. The processing of data is asynchronous to its arrival and is only processed on request. However, stream processing systems show similarities with database systems e.g. data streams are consumed in sequences and events are processed with common SQL operators such as aggregates, joins and filters.

The proposed architecture of this thesis primarily focuses on two classes of stream processing systems: (i) micro-batch based processing systems with strong fault-recovery mechanism and (ii) real-time processing systems providing high throughput or low latency capabilities. The first class is applied to the batch layer where incoming data is processed incrementally using micro batches and where the fault-recovery mechanism ensures precise results. The latter class empowers the speed layer to compute results in a timely fashion.

Comet [33] and Spark Streaming [61] are stream processing systems that collect data from continuous data streams into batches. They then periodically process these batches using batch computations similar to MapReduce [24]. For example a producer sends data to a queue in a sequential order. The queue then combines the data into multiple batches. Both systems accept either a time based batching mechanism e.g. one batch keeps the data from the last 1 second or a data size based batch e.g. one batch includes 10 MB of data. Das et al. [23] further enhance this model to adaptively adjust the batch size to decrease the end-to-end latency. However, the focus of the batch layer is not primarily to reduce latency, but to generate precise results and provide the necessary mechanisms to recover from node failure. Such goals are strongly bound to the batching mechanisms and the corresponding technique to checkpoint and recover in case of failures. Kwon et al. [38] present a stream processing model to checkpoint the state of the processing components periodically and in case of failure resume from the most recent checkpoint. Their model shows a lower impact of performance by checkpointing in an asynchronous fashion. The asynchronous aspect of the checkpointing mechanism introduces new challenges to the batch layer where global or local state may reflect the last checkpoint and therefore could cause inaccurate results.

The recent work towards batch based stream processing systems with fault recovery highlighted above, lacks the following properties that are necessary to provide a fault tolerant incremental algorithm for the batch layer: (i) only allowing for periodical checkpoints based on time or data size does not provide enough granularity to recover to a precise moment in time, which may cause inaccurate results for time window based computations and (ii) the checkpointing mechanism has to be bound to the computation process in which the recovery of data and the corresponding state is synchronized. A solution based on the open source stream processing system Samza is introduced in Section 4.3.

A second class of stream processing systems is based on a continuous operator model where the streaming process includes perpetual operators that form a graph and exchange messages with each other based on the specifically designed processing layout. The stream processing systems Aurora [2], Borealis [1], TimeStream [49], Naiad [46], TeleGraphCQ [18], and Storm [9] are recent examples of this class. Although these systems use batching models in input or output buffers to improve throughput, their main concept does not evolve around batch based communication and the batches have no influence on the client API. Lohrmann et al. [41] provide a broad analysis of this class of stream processing systems and recognize that the QoS requirements of potential applications have been disregarded. In order to fill this gap they introduced a stream processing system that optimizes the computation layout and communication channels regarding user specified latency and throughput constraints. The speed layer of the lambda architecture has to keep up with the ever-increasing volume of data that stresses the data flow of the system. At a certain point trading latency for throughput may not lead to the desired response time and dramatic measures have to be executed such as dropping messages. The implementation design proposed for the speed layer in Section 4.4 uses Storm [9] to achieve such requirements and the results of the experiments in Chapter 6 highlight the impact of these measures.

2.2 Batch Processing

The publication of the MapReduce [24] programming model and the Google File System [31] brought new attention to the research of batch processing systems. The Hadoop distributed file system [51] is an open source implementation of the concepts of MapReduce and Google File System. Many ideas evolved leveraging the possibilities of the newly introduced concept and due to broad adaption of Hadoop in the industry, different technologies were built on top of Hadoop: (i) high level languages to fill the gap between low-level procedural MapReduce and declarative SQL based programming methods and (ii) distributed storage systems to organize structured data and provide low latency queries.

Pig [47] is a high-level data flow language for MapReduce that supports ad-hoc analysis of very large data sets. Pig creates a logical query plan for a specified task and then compiles the high-level language into a series of MapReduce tasks. Pig does not include a storage system to optimize access to data and latency. Hive [55], Jaql [11] and Impala [57] offer integrated solutions to manage an indexed file structure in HDFS and provide query languages to analyze the data. Impala promises real-time data analytics, while Hive and Jaql make no promises in this regard.

Cattell [16] summarizes current SQL and NoSQL storage systems and highlights the following systems that support or that are based on Hadoop: HBase [30], Hypertable [Hypertable Inc.] and Cassandra [39]. HBase and Hypertable follow the design of BigTable [19], while Cassandra is a distributed storage engine that supports the Hadoop data file system.

Marz [44, chap. 2, 3 and 5] discusses possible solutions to store data on the batch layer and dismisses the above mentioned methods with the following reasons: (i) custom languages introduce a barrier between parts of codes (ii) modularization of code becomes increasingly difficult and (iii) the general purpose programming language and the data processing language are not tightly coupled and complicate the workflow. Marz [44] implemented a class of framework that integrates with Clojure. However, the coupling of these tool sets to a specific programming language discourages broad adaption.

3

Related Work

The lambda architecture proposed by Marz [44] starts from the problem to query petabytes of data. Such a query is unreasonably expensive and imposes high latency. This problem is divided into three layers. The batch layer precomputes the query function based on the full data set and updates the serving layer. This operation involves high latency and by the time the view of the precomputed query is finished it is already outdated. The speed layer only operates on the most recent data in order to provide views for the missing time span of the batch layer.

The batch layer holds the master data set that includes all data to precompute the necessary results. Marz [44] suggests storing all data as immutable and eternally true facts in order to iterate the preprocessing in replay. This also implies the data is stored only once in the data storage in order to produce precise results. The data storage system has to provide the following requirements: (i) efficient writes for new data (ii) scalability to cope with the increasing need to store more data and (iii) support for parallel processing and the ability to partition the data. Since the data set is continuously growing the latency to precompute the batch views on the whole data set becomes increasingly expensive and the time span to catch up with the speed layer will go up. The batch layer designed in this thesis therefore processes new incoming data using an incremental algorithm. Depending on the use case an incremental algorithm may be able to update the batch view faster, but could involve a higher complexity to store the necessary data required for incrementally update results.

The serving layer provides fast access to the precomputed results of the batch layer. These results are out-dated because of the high latency of the batch layer. Therefore the write speed to the serving layer is less important than the read performance. The serving layer has to provide low latency random reads in order to answer queries efficiently.

The speed layer is responsible for providing results to the most recent data and has to fulfill certain latency constraints based on the use case. This implies that it is not possible to compute results based on the full master data set. Instead incremental computation is applied to the most recent data in combination with persistent state if necessary. The speed layer is more complex and error prone than the batch layer, but any error is eventually compensated by the batch layer.

Fan and Bifet [28] suggest the lambda architecture as one solution to the future challenges of big data with regard to data mining. Robak et al. [50] explored the application of big data and linked data concepts in supply chain management and listed the lambda

architecture as one possible solution to deal with high volume of data in this field. They argue that such an architecture could provide lower latency to react in critical situations. However, the possibility that the real-time results may be inaccurate and therefore could bias the decision-making process is not discussed. Ye et al. [58] present a cloud based big data mining and analyzing service platform with integration of the R programming language consisting of four layers. They list the lambda architecture as related work, although no comparison is provided due to the lack of information about the lambda architecture and a missing reference implementation.

Google announced a new data processing service provided on top of their cloud platform called Google Cloud Dataflow.[1] It shows similarities to the lambda architecture by allowing clients to process data with stream and batch processing methods. The framework abstracts the burden of managing the underlying services and their configuration. Google Cloud Dataflow evolved from MapReduce [24] and successor technologies published by Google such as Flume [17] and MillWheel [4]. No technical report has been published yet, but the announcement of Google Cloud Dataflow indicates similarities to Spark [60]. Both frameworks rely on parallel collections of any size that are distributed across multiple machines in order to provide scalability. It is not possible to compare the Google Cloud Dataflow framework to the work of this thesis, due to very limited information.

Amazon maintains a set of loosely coupled, but well integrated tools that provide the means to apply the lambda architecture. Kinesis is a fully managed real-time stream processing service that uses a messaging concept similar to Kafka [37]. [2] Both systems enforce a partitioning scheme that defines the maximum parallelism of the computation and provide retention policies. While Kafka allows the user to define the retention policy Kinesis only supports storage for 24 hours. Kinesis integrates the storage engine of Amazon called S3 and its MapReduce service Elastic MapReduce. [3]. Amazon provides the necessary tools to build a batch and speed layer, but the integration of both layers and its challenges are not solved.

[1] http://googlecloudplatform.blogspot.ch/2014/06/sneak-peek-google-cloud-dataflow-a-cloud-native-data-processing-service.html

[2] http://docs.aws.amazon.com/kinesis/latest/dev/key-concepts.html

[3] http://aws.amazon.com/s3/, http://aws.amazon.com/elasticmapreduce/

4

Architecture

The architecture is divided into five components: (i) orchestration (ii) coordination (iii) batch layer (iv) speed layer and (v) service layer. Figure 4.1 illustrates the components of the lambda architecture and the data flow through the system. The orchestration component manages resource allocation, provides scheduling functionality to the batch, service and speed layer and allows the simulation of node or process failures within the system. The coordination component facilitates the entry point for data and offers high-level services to synchronize processes within different components. The batch layer facilitates a micro-batch processing system with high reliability and fault tolerance. The speed layer embeds a real-time stream processing system with a focus on fast response times. The service layer collects results and performance data from both the batch and the speed layer in order to measure the key performance indicators of this architecture.

4.1 Frameworks and Services

4.1.1 Coordination and Provisioning

Apache Hadoop YARN

YARN (Yet Another Resource Negotiator) [56] evolved from the urge to leverage Hadoop computational clusters to run heterogeneous tasks in a multi-tenant environment. Hadoop was designed to run MapReduce jobs on large datasets in a distributed fashion. It offers a simple API to its clients and abstracts the common challenges of provisioning distributed systems such as scheduling, replication, failover and input partitioning [51]. Hadoop has been widely researched and its limitations and benefits are well understood. YARN addresses the resource management and scheduling limitations of Hadoop. It separates the resource management functionality from the programming model and its API allows clients to specify fine-grained scheduling and resource allocation policies. This allows a broad range of heterogeneous tasks to run on a Hadoop infrastructure such as stream-based applications that enforce a new programming model different from MapReduce.

The architecture of YARN is based on one master node (resource manager) and multiple worker nodes (node manager). The resource manager acts as the central authority

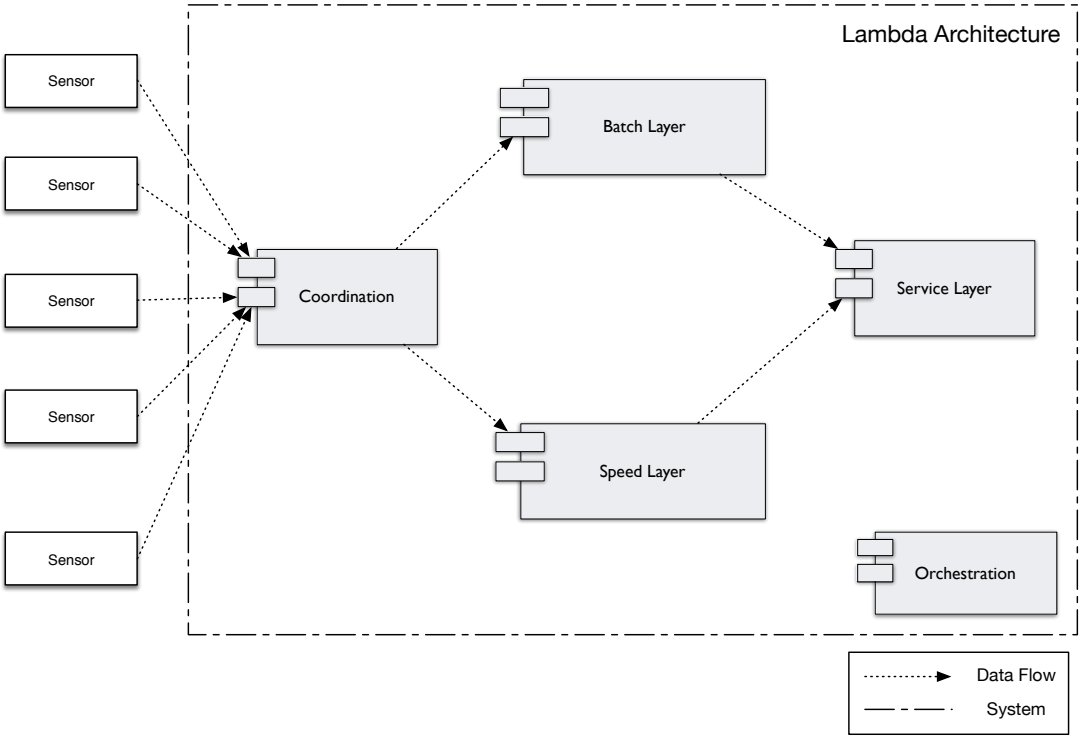


Figure 4.1: Components of the lambda architecture: External sensors send data continuously to the coordination component. The batch and speed layer provide the computational facilities to compute results and send those results to the service layer. The orchestration component manages the provisioning and orchestration and it embeds a node failure simulation mechanism.

responsible for resource scheduling and job submission. The node managers report information about resource availability, faults and job lifecycle management to the resource manager in a heartbeat fashion. From these heartbeats the resource manager acquires a global view of available resources and the status of jobs. Based on the available resources the resource manager can provision leaves (containers) on the node manager to run client applications.

The API enables clients to submit an application master to the resource manager, which encapsulates the logic to negotiate resources and start jobs in form of containers. The resource manager's responsibility does not include the proper management of node failures or status changes of client applications, but it offers a communication channel for the application master to handle such events. Figure 4.2 illustrates the architecture of YARN.

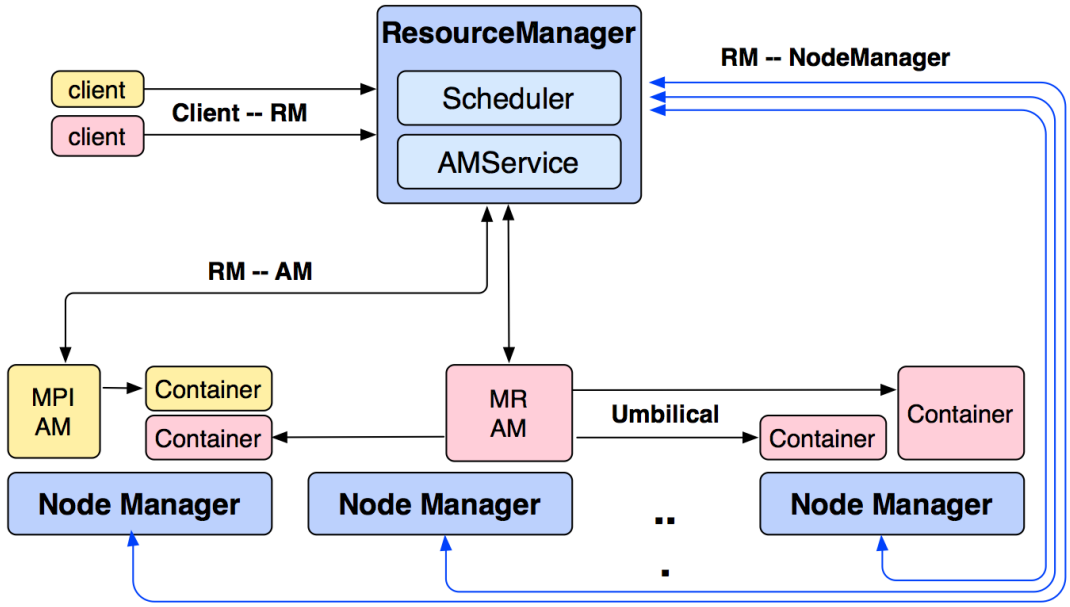


Figure 4.2: YARN architecture overview [56]: The resource manager orchestrates the YARN cluster and manages task provisioning. The node managers are worker nodes that provide resources for computation tasks. The client registers an application with the resource manager. Then the application master (AM) is started as a container on one of the node managers. It then negotiates resources with the resource manager to start containers.

Apache Zookeeper

Apache Zookeeper is a wait-free coordination service for distributed systems with the goal to provide a coordination-kernel with relaxed consistency guarantees [34]. Zookeeper does not implement specific primitives, but it offers high flexibility through its API and enables clients to build their own coordination logic such as rendezvous, group membership, configuration management, simple locks, simple locks without herd effect, read/write locks and double barrier.

The power of Zookeeper evolves around its simple design based on hierarchical namespaces with a strong similarity to file systems. Clients can write and read nodes within the namespace and watch for changes on specific nodes. A node stored on the ZooKeeper namespace is either defined as regular or ephemeral. The first allows clients to manipulate the node explicitly. The latter allows deleting nodes based on the heartbeat of a specific client. A node will be deleted when the client fails to send its heartbeat within a defined timeout.

ZooKeeper enforces two ordering guarantees: (i) linearizable writes guarantee that all updates to the state of ZooKeeper are serializable and respect precedence and (ii) all requests from one client are executed in FIFO order.

Apache Kafka

Apache Kafka [37] is a distributed and scalable messaging system. It combines the benefits of traditional log aggregators and provides an API to consume events in real time.

A topic defines a stream of messages of one particular type. Producers can publish messages to a topic and these messages are stored on nodes called brokers. Consumers subscribe to one or more topics and pull messages from the brokers. A consumer utilizes the iterator interface from one particular message stream in order to consume the messages being produced in a continual stream. The iterator is designed to block if no new messages are available and continues upon arrival of new messages. A topic is further divided into multiple partitions. Partitions are the smallest unit of parallelism and enable load balancing by dividing the partitions among the brokers. Consumer groups can divide and distribute the load according to the partitioning scheme of Kafka. This eases the common state management and locking challenges of traditional messaging systems.

A message stored in Kafka is identified by its logical offset. A consumer gets messages from one partition sequentially and acknowledges the offset of the read messages. A broker in Kafka is stateless and holds no information about the progress of the consumers. A consumer is responsible to manage the last consumed offset. This not only reduces the complexity and overhead of a broker but also allows consumers to rewind the stream and consume already read messages again. This is a powerful feature in case of node or application failures, since the application can replay messages from a stored offset after the application recovers.

Brokers and consumers coordinate among themselves in a decentralized fashion using the functionality of ZooKeeper. In particular, ZooKeeper is used to solve two major challenges main tasks: (i) detecting new or removed brokers and consumers, and rebalance each of them according to the current topic and partitioning scheme, and (ii) consumer groups can store and manage the offset of partitions in a highly available fashion.

Kafka provides two distinct interfaces to consume messages from partitions: (i) a consumer group implementation that leverages the functionality of ZooKeeper to synchronize different consumers and their respective offset and (ii) a simple consumer that enables developers to implement their logic to consume messages in a distributed fashion. The first choice abstracts many low level functionality and therefore lowers the risk of faulty code. It makes two assumptions about the consuming applications: (i) at-least-once message delivery guarantee and its possibility to send duplicate messages is not in focus and (ii) the order of messages consumed from different partitions is irrelevant.

Kafka producers can either send messages in a synchronous or asynchronous fashion. Producing messages synchronously means sending one message at the time. The thread is stopped in order to deliver each message to Kafka and returns after the broker acknowledges the message. An asynchronous producer holds a batch size of n messages in memory and whenever the batch is full it starts a new thread to deliver all messages of the current batch to Kafka. Figure 4.3 illustrates the performance implications of the batch size on the producer. While using the synchronous mode shows the lowest performance, an asynchronous producer implies further problems with regard to fault

tolerance. In case an asynchronous producer fails and is restarted, determining the precise restore point is only possible by inspecting the corresponding topic.

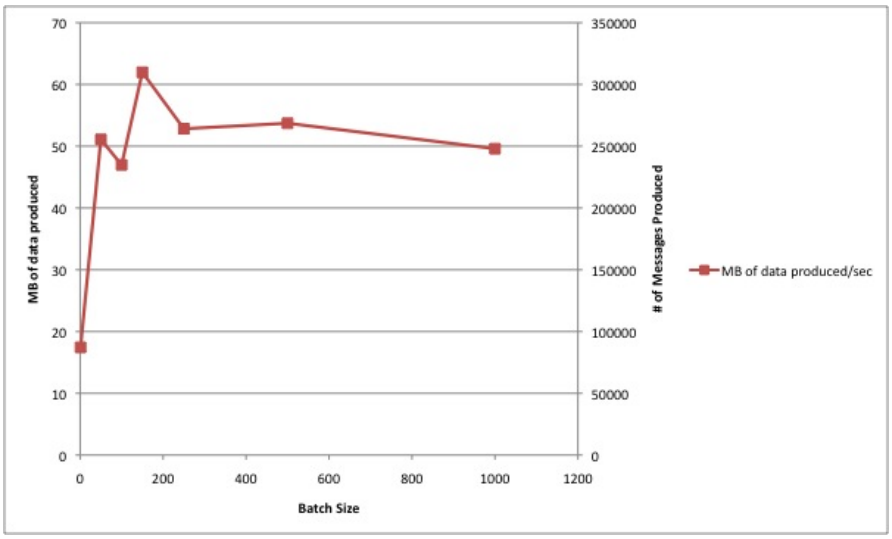


Figure 4.3: Kafka producer batch-size performance comparison [8]: The x-axis shows the batch size used by the producer to send messages to Kafka. The left y-axis shows the amount of data (MB) produced and the right y-axis shows the corresponding number of messages.

4.1.2 Stream Processing

Apache Samza

Apache Samza is a distributed stream-processing framework that leverages the capabilities of YARN to provide fault tolerance, resource management services and process isolation. Its design of task distribution and messaging strongly relies on Kafka. Samza provides a simple API to process messages and abstracts low-level operations, such as state management, processor isolation and fault recovery. A client implements the processing method to process one or multiple continuous streams and Samza manages the incoming messages and provides interfaces for outgoing messages. Figure 4.4 illustrates an example data flow within a Samza topology. The processing layout of Samza allows clients to define computational graphs and enables jobs to consume from and write to one or many streams. Samza makes no assumptions about the graph and therefore cyclic and acyclic topologies are possible. A Samza job can either write to a Kafka stream or provide its implementation to write to an external data store.

Task Distribution
A Samza job is a logical unit that processes a set of input streams and outputs messages to a set of output streams. In order to distribute the load of a set of input streams Samza

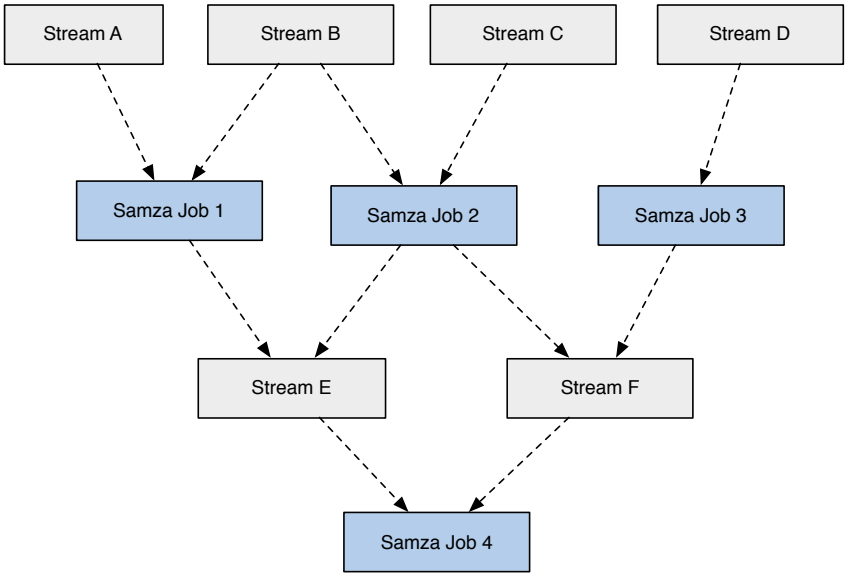


Figure 4.4: Samza stream processing topology: Every Samza job can consume messages from one or many data streams and emit messages to many streams. The processing layout may be designed as a directed cyclic or acyclic graph.

splits one job into multiple tasks based on the Kafka partitioning scheme. Therefore a task that consumes one Kafka partition is the smallest unit of parallelism. Messages are consumed sequentially in order of their offset. If a task joins two streams the messages are processed in round robin. Alternatively the client may specify a custom method through the defined interfaces.

Tasks are distributed according to the available resources on YARN. Figure 4.5 shows the distribution of stream tasks with respect to the Kafka topic partitioning and the available resources on YARN. In this example two available containers on YARN are available and Samza runs two tasks on each container in order to process the four partitions of each topic. The distribution of partitions to tasks is based on the partitioning scheme of Kafka. This mapping is negotiated the first time Samza starts and will not rebalance in case partitions are added or removed.

Two factors influence the load balancing mechanism of Samza: (i) the number of partitions and (ii) the number of containers to start on YARN. The number of containers Samza will start is configurable per job. In case the number of containers is lower than the number of partitions Samza starts multiple threads on each container as illustrated in figure 4.5. The number of total threads a Samza job starts is always bound to the number of partitions. If a job consumes from multiple topics with an unequal number of partitions, some tasks will only consume messages from one topic. For example topic A holds two partitions and topic B holds four partitions. Samza would start four tasks in order to consume from both topics. Tasks one and two would consume from both topics, but tasks three and four would only consume from topic B, since topic A only manages

two partitions. Therefore the number of tasks is defined as the maximum number of partitions in all topics a job consumes from.

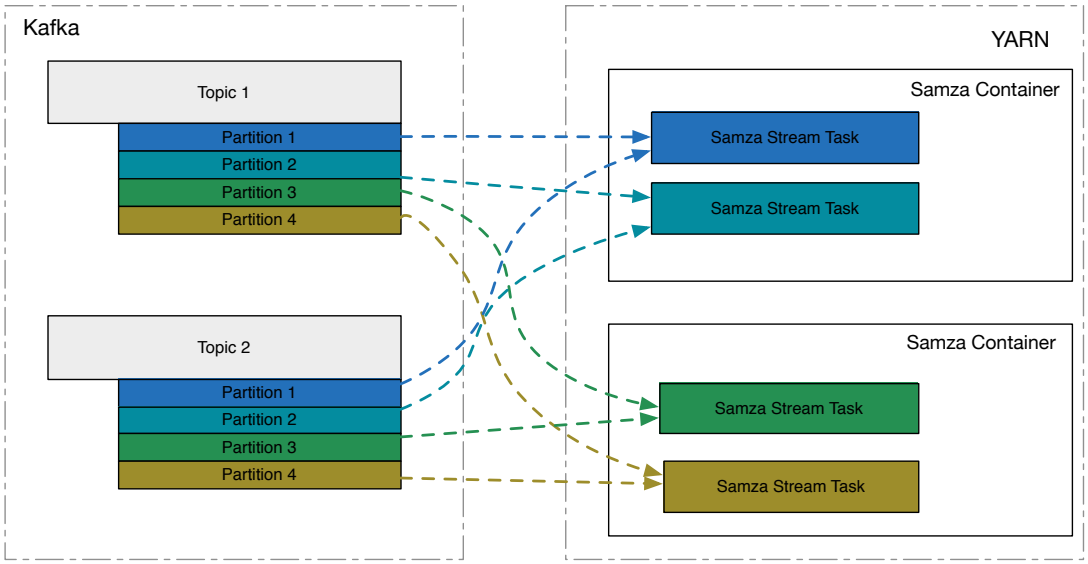


Figure 4.5: Samza task distribution: The task distribution of Samza is constrained by the maximum amount of partitions in a Kafka topic and the number of available YARN containers. A task can consume multiple Kafka topics and it will always receive messages from the same partition number in both topics.

Checkpointing and Fault Recovery

Samza promises fault recovery in case of node failure or YARN containers are deprovisioned. For each task Samza creates a checkpointing topic on Kafka. On a configurable time interval Samza writes the most recent processed offset to the checkpointing topic. In case a task fails Samza will restart a container on YARN and the task will restore to the last written offset in the checkpointing topic. The checkpointing interval is specified in milliseconds and is bound to the system time. A client can enforce the checkpointing behavior within its processing implementation. Currently an enforced checkpoint will commit not only the offset of the current task, but also all other tasks running within the same container.

State Management

Samza provides a configurable key-value store based on LevelDB [25] to manage persistent state. The key-value store is isolated on a task level. Therefore each task holds a reference to its key-value store and queries that join state from different tasks are not possible. But local storage offers the benefits of fast performance that is only limited by local resources and not bandwidth or latency.

Samza creates a separate checkpointing topic for every persistent store of one task in order to recover the persistent state after task failure. The checkpointing interval is bound to the checkpointing interval of the task. Therefore Samza guarantees synchronization of the persistent store with the task checkpoint.

Apache Storm

Apache Storm [9] is an open source distributed real-time stream processing system. The main concept of Storm is a model to represent the entire streaming application as a graph of computation. This graph is built using the Storm API and then deployed to the Storm cluster. The Storm cluster takes care of the distribution of tasks within the given infrastructure and the API is designed to handle message passing, task discovery and fault-tolerance.

Storm enables clients to run topologies consisting of streams, spouts and bolts. Figure 4.6 highlights a processing graph consisting of two spouts and five bolts. A stream is an unbounded sequence of tuples that is processed in parallel and distributed among multiple worker nodes. The source of a stream is called a spout and the processing units are called bolts. Streams are defined as tuples with a corresponding schema that names its the fields.

Spouts read data from an external input source and emit the data in form of tuples to the topology. A spout is either reliable or unreliable. A reliable spout handles communication failures and replays tuples in case it failed to be transferred or processed by the topology, while an unreliable spout does not keep any buffer of the emitted tuples and in case of failure the tuple is lost. A spout might emit tuples to multiple streams and therefore allows partitioning of data and enables scalable processing of multiple streams. The Storm framework defines basic abstractions for spouts in order to match arbitrary use cases.

Bolts are the processing units of a Storm topology and may fulfill a wide variety of use cases such as filtering, aggregations, joins, talking to external persistence layers, etc. In order to build a computational graph with multiple processing vertices the topology needs to define at least one input stream for each bolt. Bolts may emit tuples to one or multiple streams after processing the input stream.

Storm allows for parallelization of spouts and bolts using different tuple grouping strategies to pass message streams. The grouping strategy defines the partitioning of a stream and controls the number of parallelism of the next computational unit. Storm offers seven built-in grouping strategies:

Shuffle grouping
: Distributes tuples randomly across the bolts tasks and guarantees that each bolt receives an equal number of tuples.

Fields grouping
: The stream is partitioned by one or multiple values of the emitted tuple. Tuples with the same grouping field value will always be handled by the same bolt, but different values may go to different bolts.

Global grouping
The entire stream is emitted to the bolt task with the lowest id.

None grouping
None grouping indicates no preference regarding the grouping of the stream and currently this will default to the shuffle grouping.

Direct grouping
Direct grouping allows the producer to directly specify which consumer will receive the data for each tuple independently.

Local or shuffle grouping
In case the consuming bolt has multiple tasks in the same worker process the emitted tuples will be shuffled to the in-process tasks only. Otherwise shuffle grouping is applied.

All grouping
The stream is replicated to all receiving bolts.

An interface specifies the necessary behavior of a grouping strategy and it is possible for clients to create a grouping strategy to fit the parallelization needs of their use case.

Reliability
The Storm reliability features guarantee that every tuple emitted by a spout will be fully processed by the corresponding topology. Storm tracks the tree of tuples emitted by every spout in order to enable at least once processing of tuples. In case a spout tuple is not completed within a certain message timeout the tuple is failed and may be replayed later. Acknowledgment messages are sent on a separate stream to indicate a tuple has been completely processed by a bolt. The reliability features of storm are customizable and enable clients to adapt these features to the requirements of the use case.

Streams

Streams [12] is a framework to define topologies and their respective data flow in XML notation. Its functionality was briefly assessed as a possible module in the speed layer, but due to the lack of flexibility regarding the stream grouping mechanisms of Storm, streams could not be applied in this architecture. Currently Streams only allows shuffled grouping to pass messages between tasks.

Spark Streaming

Spark Streaming [61] was briefly evaluated as a possible module of the batch layer, since it combines the event-driven aspects of stream-based applications and the batch processing programming model of MapReduce. The basic concept of Spark Streaming

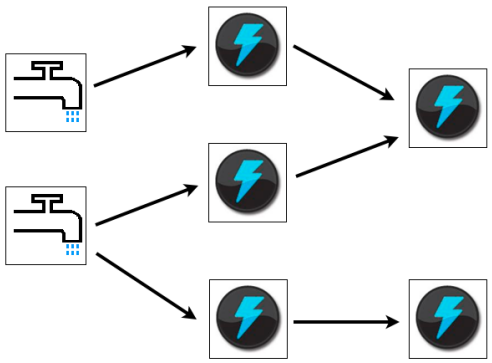


Figure 4.6: Storm topology [44]: A Storm topology consists of bolts and spouts. Data enters the topology through spouts and is then emitted to an acyclic graph consisting of bolts.

is to split the input stream into a series of deterministic batch computations. Each split is stored reliably across the cluster as a resilient distributed dataset (RDD) [60] and is then further processed using stateless and stateful operators, such as map or aggregates.

An interface allows reading messages from Kafka on multiple nodes and then further process these messages using RDDs. A test scenario proofed Spark Streaming to be inefficient and not able to recover from node failures regarding its Kafka interface. The scenario included two Spark Streaming nodes and two Kafka broker nodes. The application written for this test periodically generated six random numbers form zero to ten and published each number to one partition of a Kafka topic. In order to leverage the distributed aspects of spark streaming, the parallelization factor of spark was chosen to match the number of partitions. Although six Spark Streaming tasks were started on the cluster only one task was consuming messages from Kafka. The task consuming from Kafka started six threads in order to consume from the six partitions and added up the data from each thread without locking or synchronization. The missing synchronization of these tasks caused data loss. However, Spark Streaming allows repartitioning the tasks on the nodes during runtime. Using this functionality Spark Streaming was able to consume from one topic on different nodes, but the implementation uses the high-level consumer API of Kafka on each node, leading to random behavior regarding offset management. In some cases messages were skipped or were consumed multiple times. Furthermore in case a node running the Spark Streaming Kafka consumer fails the task is marked as finished and does not recover.

4.1.3 Event Processing

Event processing systems or frameworks perform operations on incoming events. The most common subset of operations includes creating, reading, transforming and deleting events [27]. An event processing framework abstracts the underlying complexity to perform these operations and allows to further process events using a specific domain

language.

Esper

Esper [26] is a complex event processing framework with capabilities to analyze large volumes of incoming messages or events. It targets event driven architectures with either real-time or historical data. Esper offers a domain specific language (DSL) for event processing. The processing language follows a declarative syntax for dealing with large volumes of high frequency time based messages. Esper enables clients to build the business logic on top of its DSL and supports various ways to filter and analyze incoming data streams and respond to certain conditions of interest.

Esper is a lightweight kernel written in Java and provides a highly scalable way to compute metrics in-memory with minimal latency [10]. It is capable of handling historical data or medium to high-velocity data and high variety data. Esper considers every source of input an event streams and clients build logic on top of these streams. It's architecture focuses on low-latency queries where events are processed in-memory with additional possibilities to access persistent storage. The domain specific language allows expressing rich event conditions such as correlations, aggregations with the possibility to span time windows.

4.1.4 Persistent Storage

The architecture has to provide persistent or temporary storage facilities to store results or store global state across different streams. Redis and MongoDB are introduced in the following sections.

Redis

Redis is a fast non-relational database that is widely used as a key-value cache or store [15]. It allows storing a mapping of keys to five different types of values. It supports in-memory or persistent disk storage, replication to scale read performance and sharding to scale write performance. Redis allows data to be stored persistent based on different conditions: (i) based on the number of writes in a given period of time or (ii) when the user manually calls the corresponding command. In addition redis can store every operation on disk as it happens.

MongoDB

MongoDB is a document oriented database that provides high-performance, high-availability and automatic scaling [20]. A record in MongoDB is a document that is composed of field and value pairs. A value may include documents, arrays or arrays of documents. MongoDB provides high performance data persistency and supports embedded data models to reduce I/O activity and indexes to reduce the latency of queries. MongoDB provides horizontal scalability in form of automatic sharding of data across a cluster of machines.

In addition a cluster may be configured to replicate data among multiple nodes in the cluster.

4.2 Coordination

The coordination layer is the entry point of data into the system. A typical scenario involves reading from an input source and delivering the messages to either the batch or speed layer. Figure 4.7 highlights the main components of the coordination layer and the data flow between the components and the interfaces to the batch and speed layer. An input reader fetches messages from an external data source (e.g. file, database or socket) and sends these messages to the time synchronizer. The time synchronizer has two main responsibilities: (i) incoming data has to be forwarded in order of the timestamp of the data and (ii) events might be delayed based on a data frequency rate. The delay is defined relative to the system time e.g. the data of one minute is processed in one second system time. The detailed design of the coordination layer is described as follows. First, the implementation design of the coordination layer is introduced. Second, persistent messaging and the interface to the batch layer are described. Third, in-memory messaging and the respective interface to the speed layer are addressed.

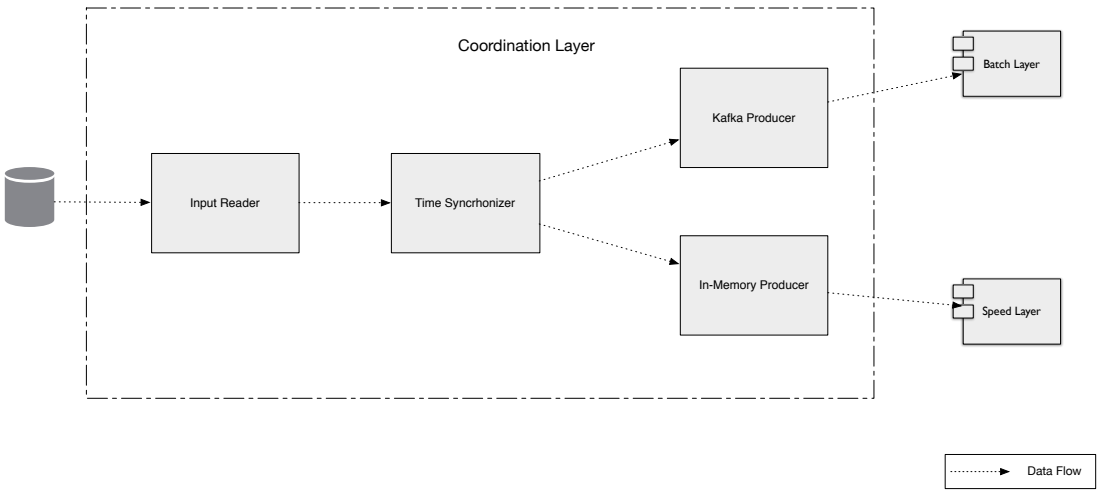


Figure 4.7: Coordination layer data flow: The coordination layer encapsulates three steps. First, the data is received from an external storage. Second, the time synchronizer enables throttling of events. Third, a producer (kafka producer or im-memory producer) enables the batch and speed layer to consume data.

4.2.1 Implementaton Design

Different setups are possible to run the coordinator in a distributed environment. The most basic setup is to run one instance of the coordinator on one machine reading from only one input source and produce messages either in-memory or in a persistent fashion to the batch and speed layer. The coordinator is designed to run in multiple scenarios e.g. on multiple machines and reading from multiple input sources. The coordinator operates in pipelines (see figure 4.8), where one pipeline corresponds to one input source. Each pipeline starts separate threads for the input reader, time synchronizer and message producer. Messaging between these threads is based on a ring buffer implementation [LMAX Trading].[1] The potential bottlenecks of the coordinator are the reader and the producer. Therefore each element of the pipeline keeps a buffer of messages pending to process e.g. the input reader will read up to 2,048 messages until it waits for the time synchronizer to free up space in the buffer.

It is assumed each input source corresponds to a separate partition of the data and therefore the order of messages is only guaranteed within a pipeline and not over all pipelines. Depending on the number and type of input sources the coordinator achieves better throughput distributed to multiple nodes e.g. in case of a file based input source the reader achieves better performance by only reading one file on one disk at the time in order to leverage the performance benefits of sequential reads. Each pipeline can be started on a separate machine with different levels of data throughput throttling in order to simulate bursts from selected input sources.

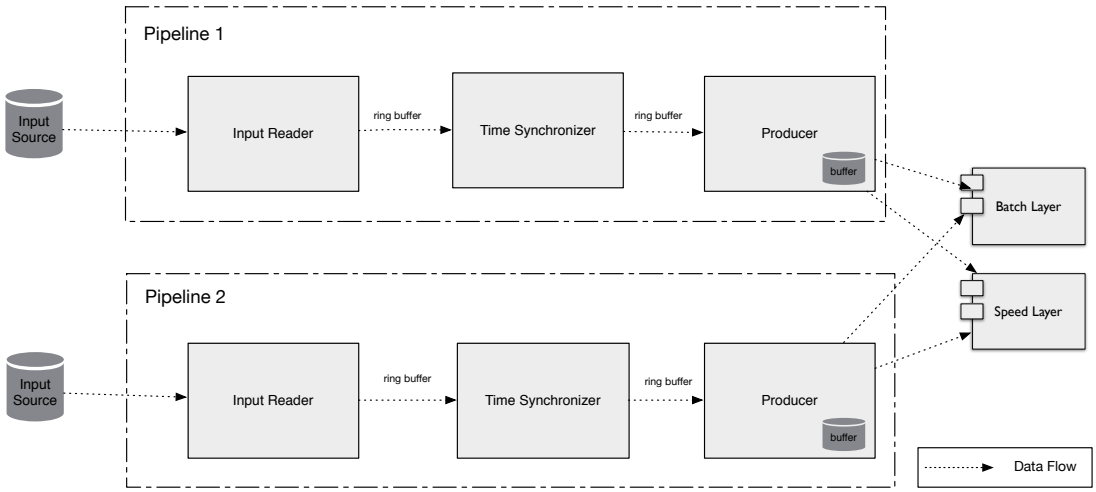


Figure 4.8: Coordination pipelines: The coordination component starts a pipeline consisting of an input reader, a time synchronizer and a producer for each input source. Pipelines are fully decoupled and may reside on different nodes or processes.

[1] Using the SRBench dataset the ring buffer achieves a throughput of 500,000 messages/s in contrast to a blocking array queue implementation with a throughput of 200,000 messages/s

4.2.2 Persistent Messaging

Apache Kafka (see 4.1.1) is used to deliver messages from the coordination layer to the batch layer. Kafka provides a producer interface to send messages to Kafka and stores the received messages first on the disk of the partition leader and second on the replication nodes. Different options are available to fine-tune the behavior of the producer and the Kafka brokers. The most important configuration options are first listed and then discussed with respect to the specific use case [6]:

request.required.acks
: This is the most important configuration option to control the reliability. It allows for three different modes: (i) the producer never waits for an acknowledgment by the broker (ii) the producer waits for an acknowledgment from the lead broker, but not from the replication broker and (iii) the producer waits for acknowledgment of the leader and from all possible replication brokers.

producer.type
: The Kafka producer can either operate in synchronous or asynchronous mode. While the asynchronous mode allows for higher throughput, the synchronous mode enables higher reliability in case of producer failure. Higher throughput is achieved by batching multiple requests together and sending the data as a bundle instead of sending each message in one request.

batch.size
: The number of messages a producer might cache before dispatching all messages to a Kafka broker.

queue.buffering.max.ms
: The maximum number messages are kept in the producer queue. After the specifc time interval is elapsed the data is produced to Kafka. Messages are either send when the batch size is reached or the specified time is elapsed.

queue.buffering.max.messages
: The maximum number of unsent messages that can be queued by the producer when using async mode before either the producer must be blocked or data must be dropped.

The coordinator will send messages to Kafka in an asynchronous mode waiting for all replication brokers to receive and persist the data. It is assumed that the coordination layer does not fail and therefore implications from using an asynchronous producer are not applicable. In case of a broker or network failure, the producer will wait for the leader election to select a new partition leader and resend an unacknowledged batch. Only at least once delivery semantics are supported. An open proposal [Kreps] suggests design changes in the implementation of the Kafka producer and the behavior of brokers in such a scenario in order to support only once message delivery. The proposed solution builds on the idea of a unique identifier for each message a producer sends and on a

catalog of received messages on the broker. To circumvent the inefficiency implicated by maintaining a database of O(number of messages) the design proposal suggests to leverage the sequential ordering of messages on the Kafka broker to keep the last state as a PID of the broker.

4.2.3 In-Memory Messaging

The communication between coordination and speed layer has to provide high throughput capabilities. To avoid the latency disadvantage of a persistent message queue, the coordinator provides an in-memory messaging service. The in-memory messaging service uses the buffer of the producer within the coordination pipeline to store messages for retrieval. A socket is opened to receive requests from clients. The management of open sockets is handled through the asynchronous event-driven network application framework Netty [Project Netty]. The asynchronous aspects of Netty are especially important to handle connection failures or read timeouts. In general client timeouts on the server side are detected using a timeout threshold and in case of failure the channel is re-opened. Netty allows for asynchronous connection handling. In case of failure the client establishes a new connection and the server will asynchronously span a new thread to deliver messages to the client, while the broken connection stays alive until a timeout is reached. Messages are stored in a thread-safe queue.

The in-memory message producer implements a simple pull based protocol on top of TCP as highlighted in fig. 4.9. After establishing a connection between the server and client, the client initiates a data transfer with the control command "fetch". After receiving the fetch command on the server side, the server will send a batch of messages to the client using the same connection. The data is lost in case the connection is closed during this transfer and the protocol is not designed to replay messages.

A push-based model is an alternative design to the pull based model described above, where the server starts sending messages to the client after a connection is successfully established and stops on connection loss. But a push based model would cause a significant amount of data loss due to the lack of fast and reliable detection of connection errors. Hence, the pull based protocol is better suited.

4.3 Batch Layer

The goal of the batch layer is to compute precise results that eventually replace the possible inaccurate results of the speed layer. In case of node failures the processing may be delayed in order to guarantee this promise. There are two possible strategies to implement the batch layer: (i) a recomputation algorithm that periodically computes the results over the full master set or (ii) an incremental algorithm that processes new data when it is introduced into the system. The design of the batch layer follows the latter approach. It is designed for stream based data that continuously flows into the system. In comparison with the recomputation approach the incremental method has to be designed more thoroughly, but may result in lower latency. The incoming stream

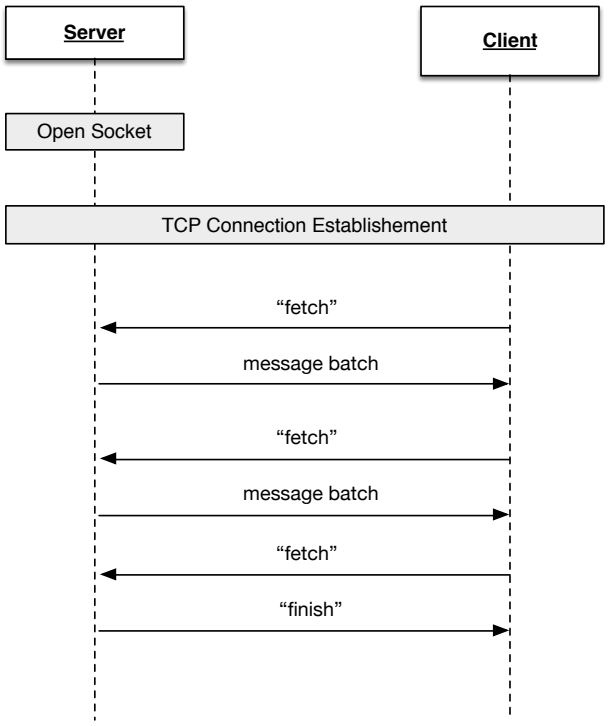


Figure 4.9: In-Memory messaging protocol: The in-memory message queue implements a pull-based model that clients use to pull data from the queue. After the connection is established between client and server the client sends a control command to the server to receive a batch of messages.

of data has to be stored persistently to provide the possibilities to recover from node failures.

The detailed description of the batch layer is structured as follows. First, the implementation design of the batch layer and its interfaces to the coordination and service layer are discussed. Second, the micro-batch processing mechanism is analyzed and compared to a recomputation approach. Third, the replay mechanism that is designed for recovery is described. Fourth, the recovery mechanism in case of node failures is further explained with regard to failures on different levels e.g. the messaging service is not available or the underlying provisioning system is unreliable.

4.3.1 Implementation Design

The batch layer has two main components: (i) a stream processing system to compute results based on micro batches and (ii) a persistent message queue to communicate between different nodes and processes, as well with the coordination layer. Figure 4.10 illustrates the components of the batch layer, its interfaces to other layers and the data

flow through the system.

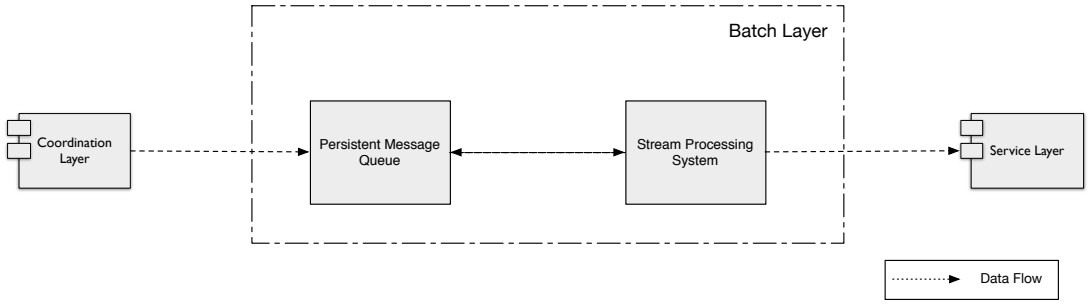


Figure 4.10: Batch layer components: The batch layer receives data from the coordination layer and stores these messages into a persistent message queue. The stream processing system of the batch layer interacts with the message queue and may consume or produce messages. The results of the computation are then send to the service layer.

The persistent message queue Kafka is the entry point of data into the batch layer. A persistent message queue offers the benefits of recovery in case of node failures compared to an in-memory queue which will loose data on node failures. Another important aspect of the message queue is its retention policy. A message queue may for example delete a message after a peer consumes it. Kafka not only stores messages persistent, but also allows for data replication on different Kafka broker nodes and the message retention policy is configurable in terms of time spent in the queue.

The components of the batch layer may be distributed on multiple compute nodes. Kafka's capability of splitting message streams into multiple partitions sets the maximum number of parallelism of the stream processing system. Therefore a stream with ten partitions may be processed by a maximum of 10 processes on a maximum of 10 nodes. Each Kafka partition may reside on a different Kafka broker node. Figure 4.11 illustrates a possible setup of the batch layer components on four nodes. In this example two Kafka brokers are managing the two partitions with a replication factor of two and two nodes run the components of the stream processing system. The loose coupling of the persistent message queue and the stream processing service ensures flexibility for different use cases. The batch layer is not designed to identify the physical distance between these components. As a result it is not guaranteed that a stream processing unit will consume messages from the Kafka broker that resides on the same node.

The stream processing system is the computing element of the batch layer. It consumes messages from the persistent message queue and computes the results based on the task of the compute job. The batch layer leverages the Samza stream processing framework to distribute the computation and communicate with Kafka. The stream processing system operates in micro-batches and not on the full batch of the data. The benefits of micro-batches are discussed in Section 4.3.2. Samza offers no high level API to compute analytics on the data. Therefore the batch layer uses Esper to provide an SQL like

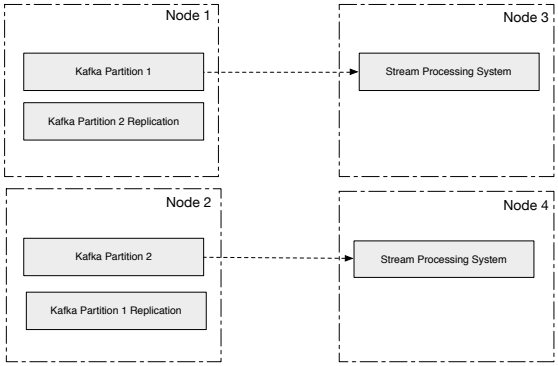


Figure 4.11: Batch layer node layout: The node layout with 4 different nodes shows an example distribution of message queue partitions and compute jobs. It shows a simple setup with 2 partitions and the corresponding 2 compute jobs. One node may be responsible for more than one partition, but the replication has to reside on a different node. The compute jobs on the other hand can all reside on the same node if necessary.

querying mechanism over a micro-batch of data. The integration of Esper enables a client to formulate computing problems in a high-level language. In addition Esper features a strong support for data analysis based on time windows. As a result the micro-batch of data is sent to Esper, which in turn provides a result after each micro-batch is finished. The result of a successfully computed micro-batch is then send to the service layer to store and further analyze the results. Figure 4.12 illustrates the batch layer components with Samza as the stream processing system. Samza uses the Apache YARN resource scheduling system to distribute processing containers on multiple nodes. The maximum number of processing containers is defined by the number of partitions in one Kafka topic.

4.3.2 Micro-Batch Processing

Micro-batch processing is a method to process data in small batches in a time discrete manner [44]. Figure 4.13 illustrates multiple batches in a continuous stream of data.

A time window or an indication within the data itself usually determines the batch size. A micro-batch based on time windows is either bound to the system time or to a timestamp of the data. The consuming of messages in batches enables for greater throughput than processing messages one by one. Although high throughput is not the primary goal of the batch layer it is necessary to optimize the access to persistent storage to improve the imposed latency. Kafka is capable of receiving and sending messages in batches defined by the number of messages. A consumer may consume messages in batches or one at a time and the same holds for the producer. It is highly important to understand the implications of consuming and writing in batches from the stream processing system with regard to reliability. This sensitivity can be illustrated as follows:

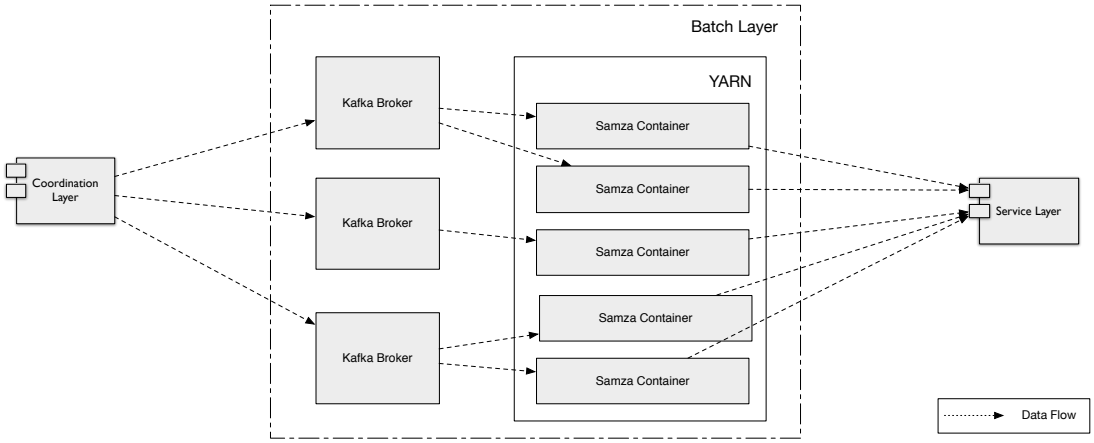


Figure 4.12: Batch layer partitioning: The data from the coordination layer is partitioned to 3 different Kafka brokers. In this example the Kafka brokers hold 5 data partitions and 5 Samza containers consume from the corresponding partition.

figure 4.14 illustrates a stream processing system with two jobs and two Kafka topics. Job1 consumes from topic1 (consumer1), computes analytics based on micro-batches and stores the results to topic2 for further processing (producer1). The second job consumes messages from topic2 and processes it (consumer2). Assuming consumer1 consumes messages in batches and commits an offset each time a time window is completed. Let's further assume producer1 produces messages in batches (asynchronous) and therefore keeps all messages in memory until a time or number based threshold is reached. In case job1 fails the messages in memory are lost, since the last committed offset reflects the last fully processed message from job1, but not the fully transferred messages of producer1. In such a scenario producer1 has to send messages in synchronous mode.

Micro-batch processing can be applied to two use cases [44]: (i) computation that only occurs within one batch and no global state is managed and (ii) computation tasks that involve state management over multiple batches. The latter use case requires users to carefully design the state management mechanism. Samza for example is designed to handle state management according to the offset management of the consumed streams and therefore has built-in support for data integrity based on the processing state. In comparison with typical batch processing systems (one at a time processing) such as Hadoop [51] the micro-batch processing method may have a shorter latency of one message completing the processing cycle [44].

4.3.3 Replay Mechanism

The checkpointing system of Samza is optimized for stream processing applications that process live data. Offline processing would imply the ability to commit an offset based on the timestamp included in the data rather then the system time. The possibility to

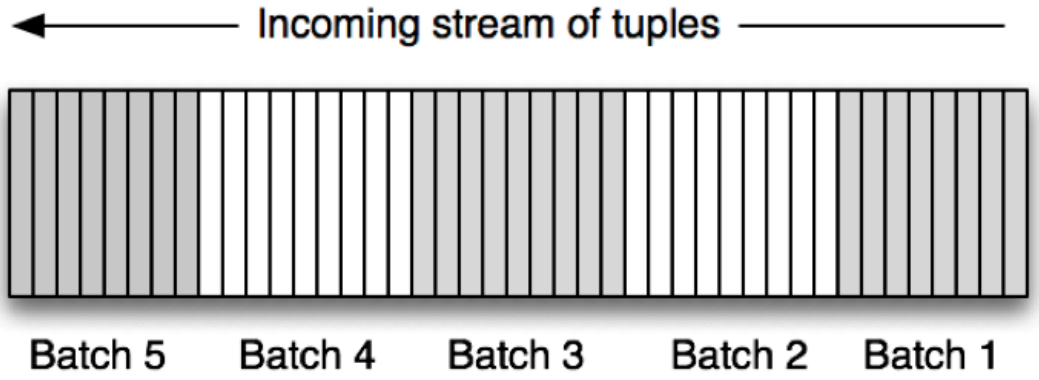


Figure 4.13: Micro-batches [44]: A continuous stream of data holds many incoming tuples. These tuples are organized in batches.

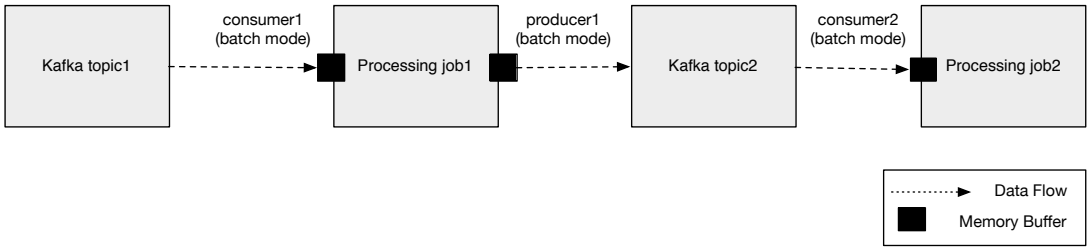


Figure 4.14: Memory buffers of batch based consumers and producers: Job 1 and job 2 consume messages in batches. Job 1 also produces messages in batches. The memory buffers store full batches and are not resilient to node failure. Therefore job 1 has to synchronize the outgoing messages with the checkpoint of the consumed batches.

enforce checkpoints within a task implementation is not suited, since it enforces checkpointing on all tasks and these tasks may not be synchronized regarding the incoming data.

The flowchart illustrated in figure 4.15 shows a hypothetical solution to this problem within the task logic itself. Each time a new time window is initialized based on the timestamp of the data, the offset is written to the persistent key-value store of Samza. In case of task recovery the offset is read from the persistent store and all messages between the current offset and the last time window offset are reread from Kafka.

This approach is inefficient, because it mirrors the built-in functionality of Kafka. Based on this reflection the Samza project has been modified to support offset commits from within the task logic that do not populate to other tasks. A task can therefore control the offset itself and only commit the checkpoint if a new time window is started. This drastically reduces the overhead to manage an additional recovery strategy.

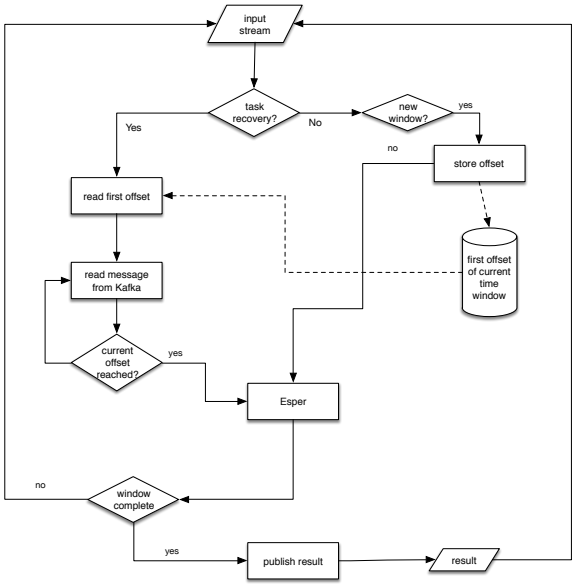


Figure 4.15: The flow chart shows the missing checkpointing mechanism of Samza to recover a full time-window of data and process it with Esper. Samza only provides checkpoints based on system time.

4.3.4 Precise Recovery

The batch layer has to be able to recover from node or processing failures on different levels in order to guarantee precise results. The recovery of a failed Samza job is handled by the offset commit management and its replay mechanism described in Section 4.3.3. But possible failures may also involve the YARN resource manager or Kafka brokers. Both systems are further analyzed with regard to node failures and their impact on the batch layer.

Two different scenarios are considered in case a YARN node fails during processing: (i) a Samza task is destroyed and (ii) the application manager is stopped. The first case results in a message to the application manager that in turn starts a new Samza task. Due to the offset committing and checkpointing mechanisms of Samza the task can resume from the last known state. In the latter case YARN starts a new application manager for Samza. The new application manager destroys all running tasks from the old application manager and starts corresponding new tasks.

Samza stores the offset commits and the checkpointing data into separate Kafka topics. Although Kafka allows topic replication to guarantee a certain degree of fault redundancy, the live data and the replication might be lost at a certain level of node failures. Potential risks are illustrated by the following three examples. Lets first consider a scenario where the offset topic is available, but the data of the state management was lost because the live broker and the replication broker failed. This will cause a corrupt state and therefore local state that needs to be kept over multiple micro-batches is lost.

Lets then consider an example where one or multiple partitions of the topic that a job is consuming from is not recoverable due to broker failures. In this case unconsumed messages are lost and Samza will continue processing the new incoming data from the new leader. A special scenario is the loss of all offset commits. If in the same time the corresponding Samza task fails, then Samza will restart the task and due to the lack of offset commits it will start again either from the beginning of the corresponding topic or from the top of the partitions. This option is configurable for each Samza job.

4.4 Speed Layer

The main goal of the speed layer is to deliver results in real-time and therefore optimize the latency and throughput to process incoming data. In comparison to the batch layer, the speed layer does not guarantee the same high availability and fault tolerance as the batch layer. In case of node or processing failures the speed layer drops messages to rapidly process the new incoming data instead of replaying all messages to the last processed state. This design leads to inaccurate results in case of node failure, but these results are eventually replaced by the results of the batch layer.

The detailed design of the speed layer is described as follows. First, the implementation design of the speed layer is discussed. Second, the impact of node failures and its implication of data loss is analyzed. Third, the scalability of the speed layer is addressed.

4.4.1 Implementation Design

The in-memory queue of the coordination layer is the data source of the speed layer. Messages are transferred using the protocol described in Section 4.2.3. The consumed messages are then sent into a Storm topology. Figure 4.16 illustrates the main components of the speed layer. A spout called ‘NettySpout‘ communicates with the coordination layer to fetch messages from the in-memory queue. These messages are then sent to the bolts for processing. The persistent distributed caching system is deployed in order to keep a global state which is not bound to the runtime of one bolt.

The NettySpout is the entry point of data into the system. It communicates with the coordinator using the Netty framework [Project Netty]. Storm requires a Spout to be non-blocking and in case no data is available nothing is emitted. Therefore the NettySpout starts a new thread to communicate with the coordinator. This thread implements the pull based protocol of the coordinator and keeps a buffer of 5,000 messages. This buffer is responsible for steadily providing the spout with new messages. If the spout pulled data from the coordinator only when Storm periodically calls for a new tuple the network latency would slow down processing. To detect failures in the socket connection to the in-memory message queue of the coordinator a heartbeat is implemented. The heartbeat checks periodically for connection failures and possibly reestablishes a connection. In case of worker failure the buffer of the NettySpout is lost and not recoverable, since the in-memory message queue of the coordination layer discards messages after delivery.

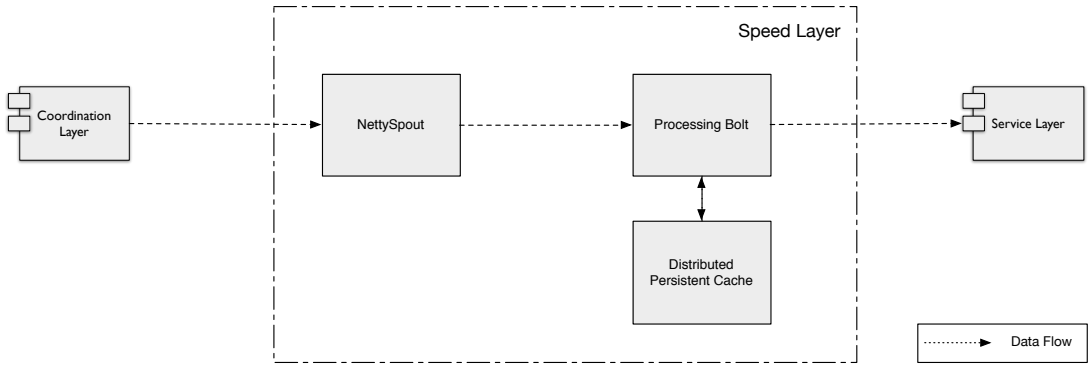


Figure 4.16: The speed layer consumes data from the coordination layer through the NettySpout. The NettySput then forwards the messages to the processing bolt. The processing bolt uses a distributed persistent cache to store temporary data and it stores the results of the computation task in the service layer. All components within the speed layer are highly scalable.

The grouping scheme that defines destination bolts is based on the partitioning of the data. The built-in functionality 'field grouping' (see Section 4.1.2) was first evaluated as a possible grouping scheme. But field grouping only guarantees messages with the same key to be transferred to the same bolt, but one bolt might receive messages with different keys. In order to guarantee that one bolt is responsible for one partition of the data a custom partitioning scheme was implemented. The custom partitioning scheme provides compatibility with the unified partitioning method of this architecture described in Section 5.3.5.

Storm can either operate a topology in a reliable fashion with acknowledgments or in an unreliable way. Acknowledgments are activated, but do not provide any reliability guarantees in this setup. The spout does not hold a list of pending messages to possibly reemit in case of failure. Also the spout is not capable of replaying message from the in-memory message queue. The acknowledgment mechanism provides the sole purpose to control the message throughput of data. In case acknowledgments are completely disabled in a Storm topology, the spout would emit messages until the memory is full, with the possible consequence of a huge risk of data loss in case of failures.

Redis is used as a persistent caching system to store global or local state and is further described in Section 4.1.4. Global state management might be used for two purposes: (i) to keep local state in a more reliable destination than memory and (ii) to share state across multiple processing units. The second purpose may need further consideration to guarantee ACID transactions and is not further analyzed in this thesis, because the evaluated use cases do not manage global state.

4.4.2 Node Failures

In order to fully understand the impact of node failures it is important to analyze the internal messaging service of Storm. The following configuration options control the buffering of messages within a Storm topology [7]. The values for this architecture are stated in brackets.

topology.max.spout.pending (30,000)
The maximum number of unprocessed tuples emitted by a spout. In case this value is not set Storm will fill all emitted messages from the spout into memory.

topology.message.timeout.secs
The maximum time a message emitted by the spout has to be fully processed and acknowledged before it is failed.

topology.transer.buffer.size (32)
The size of the transfer queue of one worker to buffer messages before sending to another worker.

topology.receiver.buffer.size (8)
The maximum number of messages to receive from network and store as a batch before sending to the executor.

topology.executor.receive.buffer.size (16,384)
The size of the executor queue to receive messages from the receiver of the worker.

topology.executor.send.buffer.size (16,384)
The size of the executor queue to send messages to the sender of the worker

Figure 4.17 highlights the messaging components of a worker. Each worker has a single receiver thread that stores incoming tuples in a list. The tuple is then send to the ring buffer of the bolt or spout. After processing the message the bolt or spout can emit a tuple to its corresponding sender thread. The sender thread stores the message in a ring buffer and the transfer thread consumes it and stores the data into a list. The buffer of the transfer thread is then flushed within a certain time span or when the buffer reaches a certain queue size. The speed layer of this architecture does not replay messages in case of failure. Therefore any message stored in these buffers is lost in case of failure. To reduce the impact of a node failure these buffers should be kept small, but these buffers also control the throughput and latency. A desirable property to increase the throughput of the system is to only send messages in batches over network, but the longer a message resides in a buffer the higher is its latency. On the other hand does a buffer size of 1 not reduce the latency, because the system will spend more time waiting for available slots in the buffers. Fine-tuning these properties is dependent on the use case of the application. The above described values are optimized for the data sets used to test this architecture.

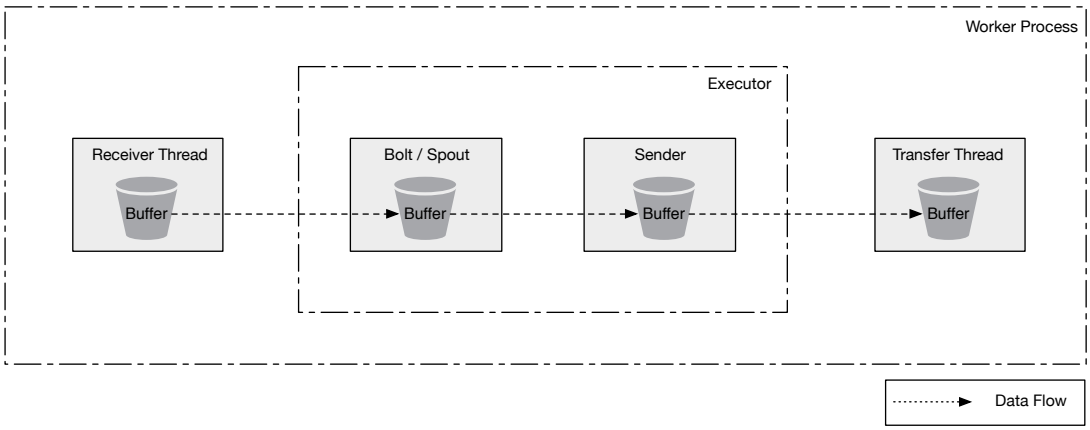


Figure 4.17: The storm worker process manages multiple queues at different levels. One worker has a single receiver thread that keeps an internal (list) buffer of received tuples. The spouts or bolts managed by this worker receive data through their ring buffer. The sender buffer of the executor keeps the outgoing data in a ring buffer and forwards it to the transfer thread. The transfer thread keeps a list of tuples and flushes it periodically to the network.

4.4.3 Scalability

Storm provides its own resource management system consisting of one master node called nimbus and one or more worker nodes called supervisors. Nimbus is responsible for controlling a topology and the worker nodes that execute spouts and bolts. In general a topology may start as many spouts and bolt as possible regarding the given resources of the cluster. Two factors of this architecture constrain the number of parallelism of spouts and bolts: (i) the number of pipelines in the coordination layer and their in-memory queues determine the maximum number of spouts and (ii) the maximum number of bolts are constrained by the number of partitions of the dataset. For example one pipeline would constrain the maximum number of spouts to one, but if the data consumed from the pipeline is divided into 10 partitions the processing bolts could scale up to 10 processes. Figure 4.18 shows a possible setup of pipelines, spouts and bolts. This example setup could cause a possible performance bottleneck in the coordination layer.

4.5 Orchestration

The lambda architecture embeds highly distributed services on each layer with the disadvantage of difficult synchronization and resource management challenges. In order to mitigate these challenges an orchestration layer is introduced. This layer leverages the Apache Hadoop YARN framework and the built-in Storm service for resource allocation and Apache ZooKeeper and Kafka for coordination.

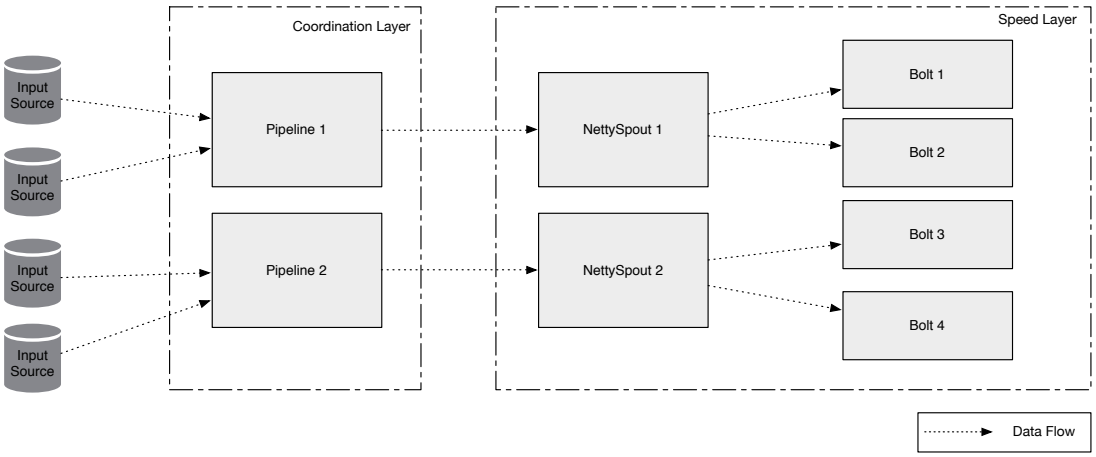


Figure 4.18: The scalability of the speed layer is partially defined by the coordination layer. One spout of the speed layer can consume data from one pipeline of the coordination layer. The spout can then further partition the data and emit it to multiple bolts.

First, the challenges of the orchestration and coordination with regard to resource management and cluster health are discussed. Second, the details of node failure simulations are explained.

4.5.1 Resource Management and Cluster Health

Each machine within a compute cluster serves a certain purpose which is defined by the services running on the node and the available hardware resources. YARN and Storm's nimbus service use two different strategies for the resource utilization. The first distributes processes according to the resource request of the client. Currently the amount of memory and cpu cores are supported. The latter distributes all tasks in an even or almost even way on the available resources. Both resource managers employ a heartbeat mechanism to detect unresponsive nodes. A node failure may not be detected unless the heartbeat fails. Therefore the detection of node failures is delayed by at least the minimum heartbeat frequency.

Apache Kafka is the persistent messaging system of the batch layer. Kafka follows a master-less cluster setup and supports for replication and fault tolerance. However, it does not provide a solution to the CAP theorem. The CAP theorem highlights the trade-off between being correct and being always available in distributed systems [14]. Gilbert and Lynch [32] proved the CAP theorem to be unsolvable. As a consequence it is impossible to implement a read-write storage in an asynchronous network that would always guarantee availability, consistency and partition tolerance. One of the design goals of Kafka is to allow n-1 node failures, where n is the total amount of nodes in a cluster setup [37]. Each partition is managed by one leader and may have multiple

replication brokers. If n-1 nodes fail, the remaining node becomes the partition leader. At this point Kafka trades availability against consistency and allows clients to read and write to the managed topic, even though replication hosts are not available. But in case the last remaining node fails before any replication node becomes available, the messages received during the failure are lost. When a replication node recovers it will resume operations from its last known state. This scenario not only depends on the number of brokers in the cluster, but also the replication factor defined for each topic. In case the replication factor r is not set to n-1, where n is the number of nodes in the cluster, the failure of r+1 nodes may already result in data loss.

4.5.2 Node Failure Simulation

The goal of the node failure simulation is to gather information on the behavior of services and the impact on the outcome of a computational task e.g. decrease in precision, recall and throughput. A node failure is defined as the stop of each service of this architecture on the selected machine. To simulate a machine failure the SIGKILL signal is sent to each running process and its sub processes. The SIGKILL signal does not allow the process to perform clean-up actions and therefore produces similar results to a complete machine failure with regard to the running process [42].

Node failure simulation not only involves the shutdown of processes and services but a node may also restart services. Two different scenarios are possible for the restart of a machine and its services: (i) a machine may experience unrecoverable hardware issues resulting in the complete loss of data or (ii) a node may fail on the operating system level and its state is recoverable after a hard boot.

The node failure simulation component embeds a coin flipping mechanism to determine which machine to fail at what time. The following arguments are considered:

Probability
: The probability of a node failure that occurs within the given interval. Each interval a pseudo-random number between 0 and 1 is generated using the Mersenne Twister [45] pseudo-random number generator. In case the pseudo-random number is below the given probability a node failure will be initiated.

Interval
: The interval of possible node failures in seconds. This represents a floating value, where each interval may be enlarged or reduced in a pseudo-random fashion in order decrease the regularity of a node failure possibly leading to more diverse insights.

Concurrent Number of Nodes
: The maximum number of concurrent node failures that will occur in the same interval.

Re-alive Timeout
: The time to wait until the services on the failed nodes are restarted.

4.6 Service Layer

The service layer provides a set of tools to support near real-time process monitoring, localized performance measurements and result collection. The service layer does not compare to the serving layer of the lambda architecture as defined by Marz [44]. Marz [44] defines the serving layer by its ability to access the results of the batch layer computations with low latency. The serving layer would index views of the batch layer and provide the necessary interfaces to access the precomputed data with low latency queries.

This thesis does not analyze the possible solutions for the serving layer and merge layer. It focuses on the effects of the batch and speed layer. The service layer provides a toolset to further analyze and understand these effects. The description of this layer is structured as follows: First, the corresponding infrastructure in terms of services and dependencies is discussed. Second, the process monitoring component and its visual output is explained.

4.6.1 Logging Infrastructure

Figure 4.19 illustrates the log collection and aggregation architecture and its dependencies to external processes. A process uses the Log4j2 framework to emit log messages regarding performance and operations.[2] Log4j2 will start an asynchronous process to emit them to the flume appender.[3] The flume appender opens a socket connection to the flume agent. The internal pipeline of flume receives messages from a flume source, sends them to a channel and finally the flume sink will redirect the messages to the configured output. At each step arbitrary log transformation are possible. After transformation the flume sink sends the data to the elasticsearch database.[4] Elasticsearch indexes the received log messages on every accessible field and provides a real-time view to the user in form of the Kibana web interface.[5]

4.6.2 Process Monitoring

The logging infrastructure described in Section 4.6.1 enables a user to gain insights into the running processes in near real-time in form of a web service. Due to the different steps of log collection, transformation and aggregation a delay of approximately 10s is anticipated. While the architecture allows for a very diverse set of analysis based on the data ingested into the log collection and aggregation framework, the following two points are focused to monitor the architecture: (i) show the progress of the computation in terms of messages processed on each level of the architecture (see figure 4.22) and (ii) show the throughput over time for each component of the architecture (see figure 4.20 and 4.21).

[2] http://logging.apache.org/log4j/2.x/
[3] http://flume.apache.org/
[4] http://www.elasticsearch.org/
[5] http://www.elasticsearch.org/overview/kibana/

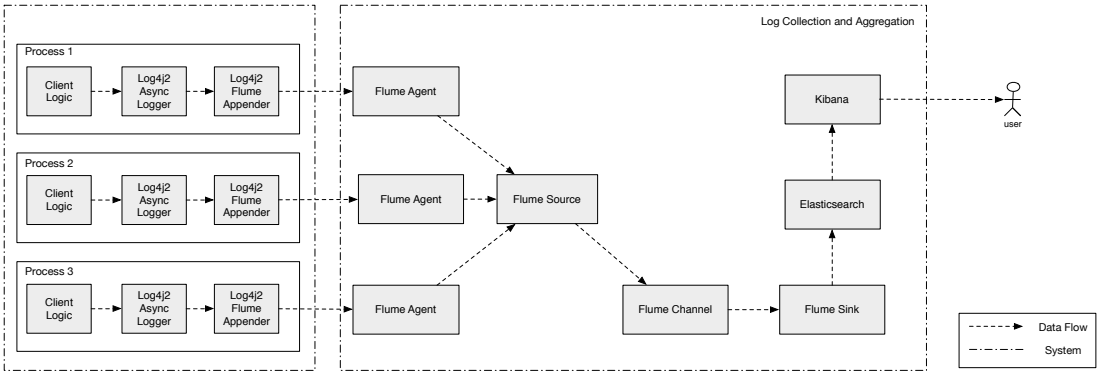


Figure 4.19: Log collection and aggregation: The client logic can send log messages through the Log4j2 interface to an asynchronous logger. This logger transfers the received messages to Flume. Flume opens multiple agents to receive log messages and after internally processing sends these messages to elasticsearch. Elasticsearch is fully indexed and can be accessed from the web interface Kibana.

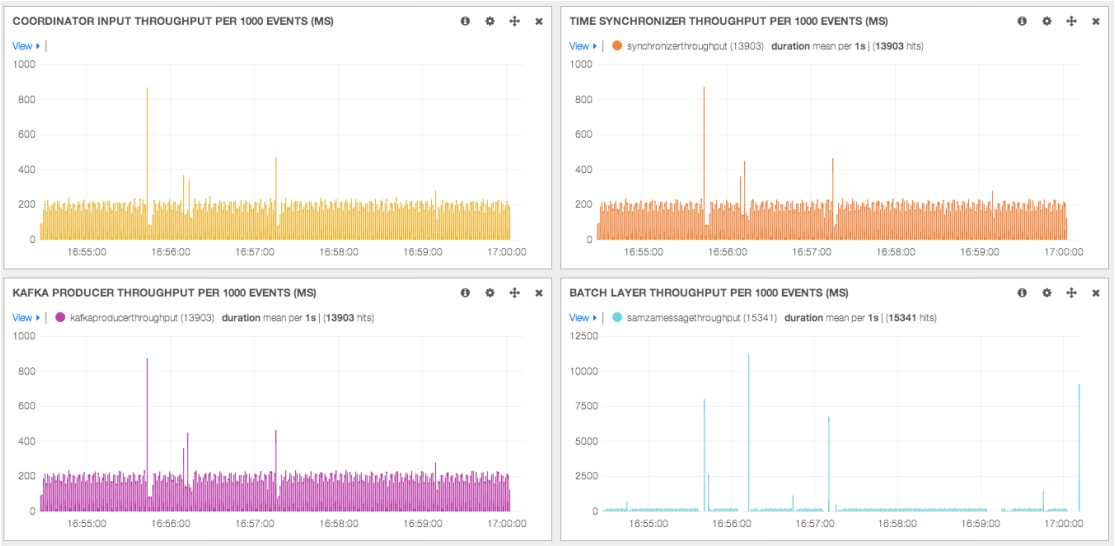


Figure 4.20: Batch layer monitoring: The graphs show histograms of the average time it takes to process messages at different components of the architecture. From top left to bottom right: (i) input reader (ii) time synchronizer (iii) Kafka producer and (iv) Samza job

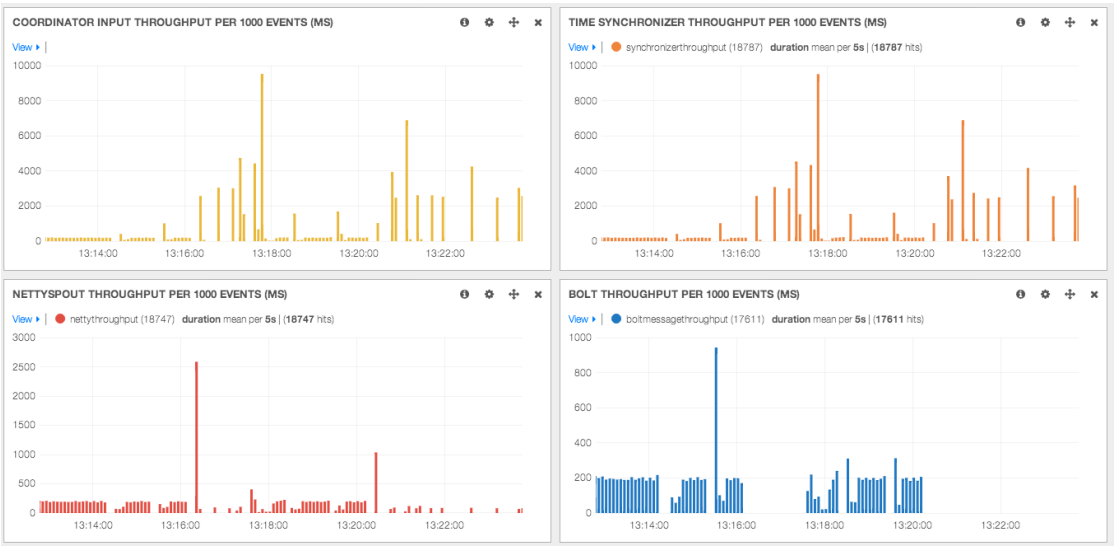


Figure 4.21: Speed layer monitoring: The graphs show histograms of the average time it takes to process messages at different components of the architecture. From top left to bottom right: (i) input reader (ii) time synchronizer (iii) NettySpout and (iv) Bolt.

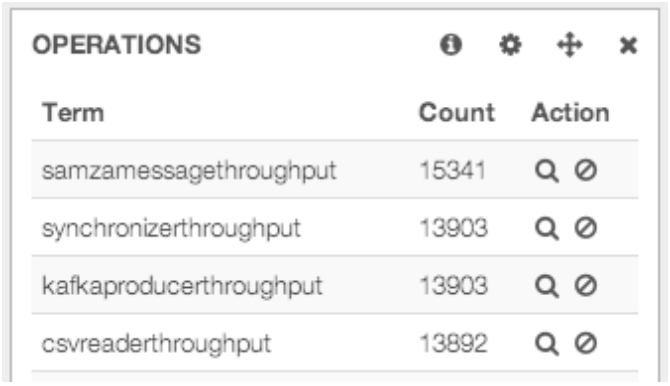


Figure 4.22: Progress monitor: Shows the total number of events processes by the components of this architecture.

5

Design of Experiments

The design of the experiments and the setup of the experiments is described as follows: First, the infrastructure where the experiments are conducted is introduced. Second, the deployment of the services and components is described and the automatic deployment scripts are briefly explained. Third, the dataset and corresponding queries to test the architecture are highlighted. Fourth, the node failure scenarios of the experiments is explained.

5.1 Infrastructure

The experiments were run on a 8 nodes cluster provided by the Dynamic and Distributed Information Systems Group at the Department of Informatics of the University of Zurich. Table 5.1 highlights the hardware information of the nodes in the cluster.

Memory	128 GB
CPU	40 CPUs, 2 threads per core, 10 cores per socket, 64-bit, 1200.000 Mhz
Ethernet	1 Gbit

Table 5.1: Shows the resource information for each node of the cluster the experiments were run on.

5.1.1 Automatic Deployment

The cluster used to run the experiments is controlled by the SLURM [59] resource management system. SLURM allows to specify the amount of resources needed for a task and handles the task provisioning of multiple tasks depending on the available resources. The automatic deployment is started as a SLURM job with parameters for the following options: dataset, speed or batch layer, number of nodes, query to analyze the data, speed of the coordination layer, node failure probability, node failure interval, node failure re-alive timeout and number of concurrent node failures.

The automatic deployment is responsible for starting all services of the architecture, starting the experiment and collecting the results. The cluster layout is defined in a

JSON based configuration file. The starting procedure includes the following four steps. First, the master daemons of the orchestration layer are started e.g. Storm nimbus, YARN resource manager, ZooKeeper. Second, the worker nodes are connected to the master daemons e.g. Storm supervisor, YARN node manager, Kafka broker. Third, the batch or speed layer is started and prepared to receive data. The distribution of tasks to worker nodes is managed through the corresponding mechanism in YARN or Storm. Fourth, the coordination layer is started on each node and the flow of the input data is delayed until the node failure simulation is started. After a successful experiment the results are stored for further analysis as a database dump.

5.2 Experimental Setup

Figure 5.1 highlights the cluster setup for the experiments. The master node provides all master daemons of the batch and speed layer. The worker nodes are responsible for processing and therefore hold all worker daemons and processing layers. The batch and speed layer are not processes by themselves, but represent components managed by the respective orchestration services. The coordination layer is a standalone process and started through the automatic deployment script as described in 5.1.1. Kafka is configured with a replication factor of three for each partition. A higher replication factor leads to more network traffic within the cluster, but would increase fault-tolerance.

5.3 Data Sets and Queries

In order to gain insights of the designed architecture and generate key performance indicators such as throughput, precision and recall two distinct data sets and their respective queries were implemented complementary to this architecture. The DEBS Grand Challenge 2014 data set [3] and the SRBench data set [62] were used to generate key performance indicators. For each data set two distinct queries were selected to measure the systems behavior. The results of the SRBench queries are either true or false without further granularity. The queries selected for the DEBS data set produce floating point values and enable further discussion of their difference to the baseline.

5.3.1 SRBench Data Set

The SRBench benchmark is a general-purpose data set primarily designed for streaming applications such as RDF/SPARQL engines. It is based on real-world data from the Linked Open Data Cloud [62]. The data set contains a subset of the US weather data collected since 2002 by MesoWest7. This includes sensor measure phenomena such as temperature, visibility, precipitation, pressure, wind speed and humidity during time periods that several major storms were active including Hurricane Katrina, Bill, Bertha, Ike, Wilma etc. Due to time and resource constraints only the Bertha data set from the SRBench benchmark is used for the experiments. The data set was chosen randomly.

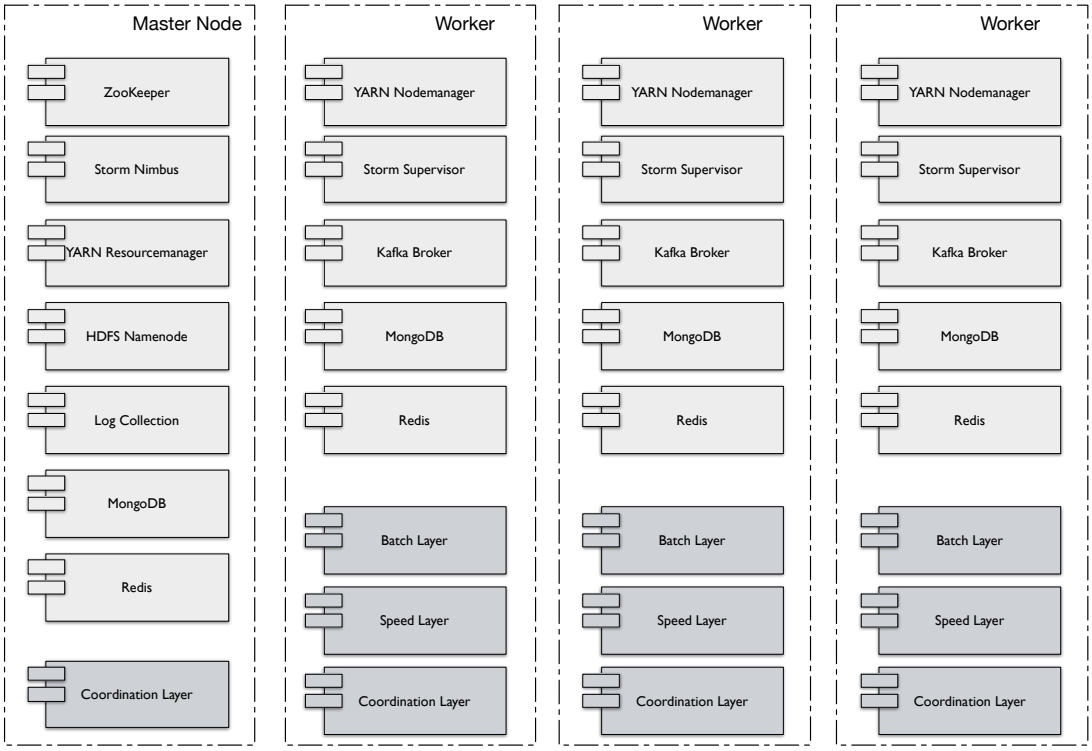


Figure 5.1: Shows the setup of this architecture on the master and worker nodes.

Furthermore, only queries 1 and 7 were run during the experiments. Both queries produce boolean results. The decision to choose this type of results is explained in Section 5.3 The corresponding key performance indicators produce results that are either true or false in nature.

5.3.2 DEBS Grand Challenge 2014 Data Set

The DEBS Grand Challenge 2014 data set originates from smart plug recordings of private households. A smart plug is equipped with a range of sensors in order to measure power consumption related values. Every smart plug deployed for this challenge collected data roughly every second for each sensor in each smart plug. The data set is collected in an uncontrolled environment and therefore possibly includes malformed or missing measurements. The resulting data set contains over 4 Billion measurements in 2,125 plugs distributed across 40 houses. The measurements cover a one-month period starting from the 1st of September 2013. The first query of the DEBS Grand Challenge includes the prediction of the load for each house within a given time window. This query was implemented according to the baseline prediction given in the query task [3]. A second query "average load" measures the average load in batch windows of 15 minutes. The first query uses global storage, while the second query only operates on the flowing data.

5.3.3 Data Set Statistics

Table 5.2 highlights the key statistics for both data sets. The SRBench data set is further categorized by the name of the observed storm.

Data set	Event Count	Avg. Size	Max.	Min.	Variance	Std. Deviation
Debs	4,055,508,721	110.473 B	120 B	88 B	23.176	4.814
SRBench Bertha	21,254,512	170.593 B	200 B	152 B	133.449	11.552

Table 5.2: Data set statistics for DEBS Grand Challenge 2014 and SRBench: number of events, avg. event size, max. event size, min. event size, variance event size, std. deviation event size.

5.3.4 Baseline

A baseline was calculated for each query on each data set. The baseline was computed by running the query on the batch layer five consecutive times with the same results. In addition the baseline was verified by comparing the results with the outcome of the speed layer.

The baseline includes the results of the query and the start and end time of the corresponding time window. A particular result is comparable to the exact time window it originates from.

5.3.5 Partitioning

The partitioning of a data set is depending on the computational task to solve. The queries implemented for the SRBench data set compute results based on events of the stations. Every SRBench sub data set includes many hundreds of stations. The queries implemented for the DEBS data set compute aggregates and predictions either based on the house or the plug. A hash bucket algorithm (consistent hashing) was applied in order to define one unified method to partition the data of both data sets. Two constraints apply to this method: (i) the partitions may differ in size and (ii) adding new partitions at runtime will cause reshuffling of certain buckets. The number of partitions for the experiments was set to 8. Section 8.3 explains the source of this decision and the resulting limitations regarding the evaluation of the experiments.

5.4 Node Failure Simulation

Node failures may occur in different flavors. Two different node failure scenarios were considered for the two data sets: (i) the scenario of an unstable networking, where node failures occur frequently and the network may become available again within seconds and (ii) an elastic setup where node failures may occur frequently and a monitoring component detects failed nodes and fires up a new node within minutes. The DEBS

Grand Challenge 2014 data set is bigger in size and number of events than the SRBench data set and therefore better suited for the latter use case. Due to the small size of the SRBench data set only frequent and short node failures are possible to simulate.

A node failure includes the shutdown of the following processes with the method described in Section 4.5.2: (i) YARN node manager (ii) Storm supervisor and (iii) Kafka broker.

6

Results

This chapter presents the results of the experiments and an in-depth discussion of the results follows in Chapter 7.

The experiments were conducted by running two distinct queries on each data set with and without node failure simulation. The node failure simulation is capable of simulating multiple concurrent node failures, but in order to adequately measure the impact of node failures only one node at the time was simulated. The failing node was chosen at random as described in Section 4.5.2. Multiple nodes were not available throughout the experiment, but no experiment enforced concurrent node failures higher then 1. However, concurrent node failures caused by system failures were possible and occurred throughout the experiments.

All results concerning throughput measures were stripped at the beginning and end by 10 seconds in order to remove the effects of the initial bootstrap and the shutdown mechanism.

First the relevant key performance indicators (KPIs) are introduced and the corresponding technique or process used to obtain the data is described. Second, the KPIs of the relatively small SRBench data set are highlighted. Third, the results of the DEBS Grand Challenge data set are described.

6.1 Key Performance Indicators

The following list compiles the key performance indicators used to elaborate on the outcome of the experiments:

Throughput
: The throughput shows the number of events per second processed by the system accumulated over all nodes. It is measured in the processing component of batch or speed layer in steps of 1,000. A node failure may cause incorrect results by a maximum of 1,000 events per failed node. The unit of the throughput is stated as events/s and MB/s. The latter is calculated using the average size of one event (see Section 5.3)

Precision
: Precision refers to the fraction of accurate results within the returned result set.

A result is only counted as precise if the result exactly matches the result from the baseline (see Section 5.3.4).

Variance
The variance shows the average variance of the difference to the baseline within a time frame of results measured in floating points.

Recall
Recall refers to the fraction of accurate results within the possible result set. The experiments contain two different types of results: (i) a boolean type that is either accurate or not or (ii) a floating point type whith an additional indication of the variance of the results. The latter is highlighted in two flavors. First, the recall of the accurate results is shown and second, the recall of the results below the variance is stated with a corresponding indication.

Health
Health is measured by the fraction of responsive nodes over time in order to indicate the health of the cluster within different time frames.

Time-to-precision
Time-to-precision measures the time span it takes to replace an inaccurate result of the speed layer with the precise result of the batch layer.

Each experiment stores the result of the query. Based on the result the precision, recall and variance are measured. The health of the cluster is calculated using log files for each node that provide the current state of the node at time x. The throughput is measured based on a performance log message send every 1,000 processing steps. Section 4.6.1 shows the process used to collect and aggregate these performance measurements.

6.2 SRBench Data Set

The SRBench Bertha data set was tested using two queries: (i) get the rainfall observed once an hour and (ii) detect broken stations. Multiple experiments were conducted for each query. Experiments with node failures were conducted using a throttling mechanism in the system time synchronizer of the coordination layer (see Section 4.2), because the node failure simulation implies interval limitations higher than the processing time of the full data set without throttling.

6.2.1 Rainfall Observed once an Hour

This query tests the engines ability to filter and aggregate events. First, a performance run of experiments without throttling and node failures is highlighted. Second, experiments with throttling are shown. Third, experiments with throttling and node failures are described.

Performance Run

This experiment measures the systems behavior with the maximum throughput possible and no node failures. Table 6.1 highlights the overall statistics of this experiment for the batch and speed layer. Both layers show the maximum precision and recall possible, while the speed layer finishes the experiment 5.313 times faster than the batch layer.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	170.103 s	avg: 117,000 (19.035 MB/s) max: 220,000 (35.792 MB/s) min: 17,000 (2.766 MB/s)	1.0	1.0	1.0
Speed	32.012 s	avg: 521,000 (84.762 MB/s) max: 684,000 (111.280 MB/s) min: 380,000 (61.822 MB/s)	1.0	1.0	1.0

Table 6.1: Overall statistics of the query "rainfall observed once an hour" run on the batch and speed layer without node failure simulation using the SRBench Bertha data set.

Figure 6.1 highlights the results of the batch layer (a, b) and the speed layer (c, d) without node failures and throttling. It shows that the throughput of the batch layer fluctuated dramatically between 10,000 and 210,000 events/s in the first 30 seconds of the experiment and stabilized around 120,000 events/s. In the last 20 seconds the throughput declined steadily until the end of the experiment. The throughput of the speed layer slightly fluttered in the first 23 seconds of the experiment and then fluctuated dramatically for the last 20 seconds.

Both layers were not subject to any node failures and the precision and recall remained stable at 1.0. No failures in the results were measured.

Throughput Throttling

This experiment shows the behavior of the batch and speed layer with a throttled data input speed. Table 6.2 shows the overall statistics of this experiment. Both layers process events at the same throughput rate of 41,000 events/s on average. No events were lost during the experiment and therefore the precision and recall of both the batch and speed layer are 1.0.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	498.470 s	avg: 41,000 (6.670 MB/s) max: 55,000 (8.947 MB/s) min: 24000 (3.905 MB/s)	1.0	1.0	1.0
Speed	498.392 s	avg: 41,000 (6.670 MB/s) max: 56,000 (9.110 MB/s) min: 33,000 (5.369 MB/s)	1.0	1.0	1.0

Table 6.2: Overall statistics of the query "rainfall observed once an hour" run on the batch and speed layer with input speed throttling applied to the SRBench Bertha data set.

Figure 6.2 highlights the results of the batch layer (a, b) and the speed layer (c, d) with a throttled input speed. The throughput of the speed and the batch layer gradually fluctuate between 35,000 and 50,000 events/s. Both layers show similar patterns in the fluctuation of the throughput. Therefore the speed of the input throttle defines the same latency for both layers.

Node Failure

This experiment highlights the results of the batch and speed layer with node failures and input speed throttling (see Section 4.2). The node failure simulation was configured with an interval of 45 seconds and a re-alive time of 15 seconds. The probability of a node failure was set to 1 in order to guarantee at least one node failure every interval.

Table 6.3 shows the overall statistic of this experiment. In total both layers show the same cluster health throughout the experiment. The batch layer finished 1.6 times faster than the speed layer. The throughput of the batch layer is 1,000 events/s slower than without node failures. The precision of the batch layer is 1.0 with a recall of 1.284. Therefore all retrieved results are precise, but the result set includes more results generated than in the baseline e.g. duplicates. The speed layer shows a slow processing speed of 17,000 events/s that is 2.4 times slower than without node failures. In addition the precision dropped to 0.987 with an even smaller recall of 0.798.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	658.817 s	avg: 40,000 (6.508 MB/s) max: 304,000 (49.457 MB/s) min: 0	1.0	1.284	0.963
Speed	1,040.746 s	avg: 17,000 (2.766 MB/s) max: 549,000 (89.316 MB/s) min: 0	0.987	0.798	0.963

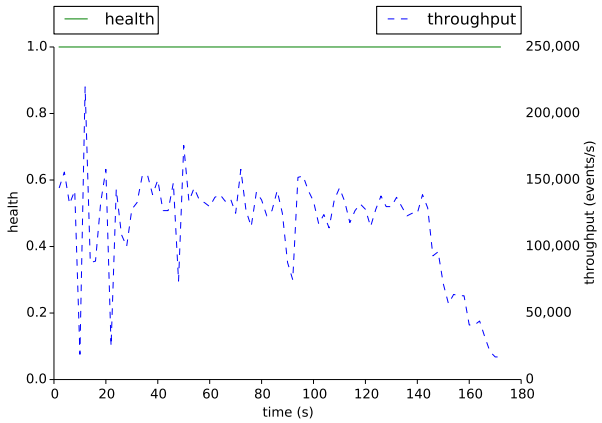
Table 6.3: Overall statistics of the query "rainfall observed once an hour" run on the batch and speed layer with a throttled input stream and one node failure every 45 second interval for 15 seconds applied to the SRBench Bertha data set.

Figure 6.3 highlights the results of the batch layer (a, b) and the speed layer (c, d) with node failures. The throughput histogram of the speed layer shows a very irregular behavior in figure 6.3c. Processing stops at the same time as a node failure occurs and resumes within a period of no node failures. During the short time the speed layer processes data, the throughput exceeds the throughput of the experiment without node failures and throttling. The input speed is throttled with regard to the system time as described in Section 4.2. Therefore the processing may catch-up with the coordinator and process at the highest throughput possible. A node failure always affects the throughput of the computation, but the throughput does not increase every time a node recovers from its failure. It is possible that multiple periods with a health of 1.0 may pass before the speed layer continues processing.

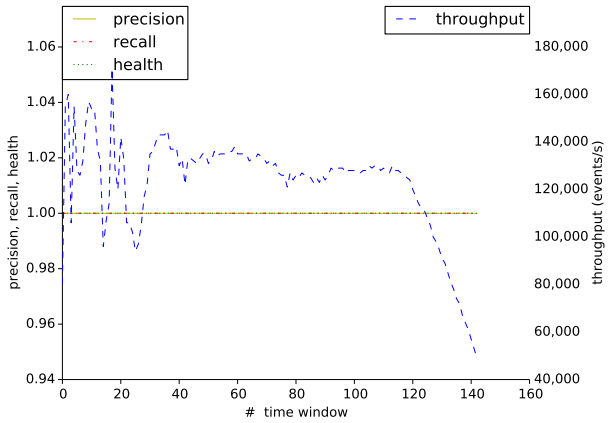
The precision of the speed layer slightly zigzagged throughout the experiment with sudden drops at the beginning of every node failure. The first two node failures had the biggest impact on the recall. The first node failure caused a steep decrease and the recall bottomed out at 0.12. The node failure between the 60th and 100th time window had a lower impact on the recall and a slower throughput.

Node failures in the batch layer cause sudden drops in throughput followed by a steep increase. The throughput peak is 300,000 events/s and is followed by a relatively long period of no throughput. This demonstrates the catching-up effect of the batch layer after a time of low throughput. Figure 6.3a shows two slightly bigger drops in health in periods with node failures. This is the surprising result of unplanned system failures during the experiment.

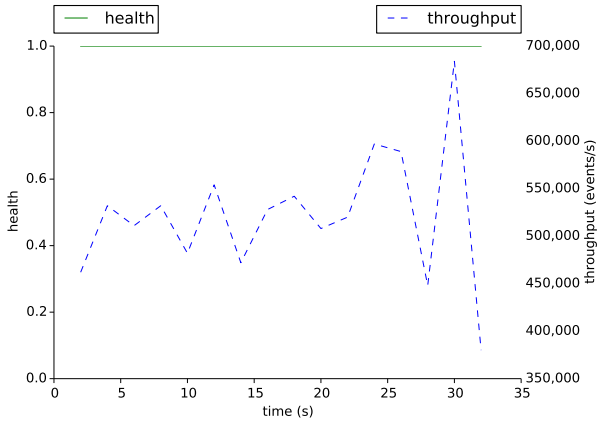
The precision of the recall remains stable at 1.0 during the whole experiment. Figure 6.3b shows different effects caused by node failures. The first two node failures caused rapid fluctuation in the throughput, but did not affect the precision. The third node failure leads to a considerable increase of throughput and recall. The increase of recall to the peak of over 1.6 indicates the appearance of duplicate results. This effect is further described in Section 7.1.3.



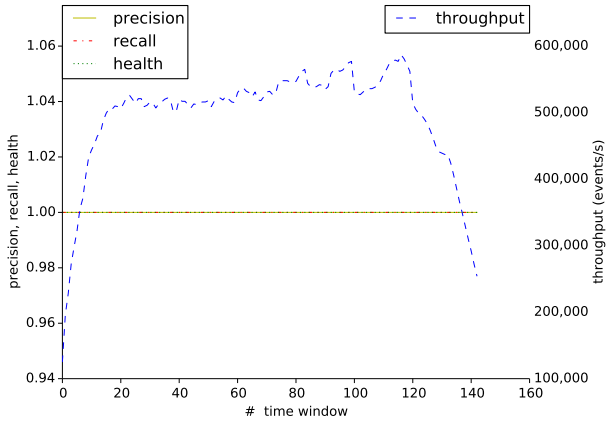
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



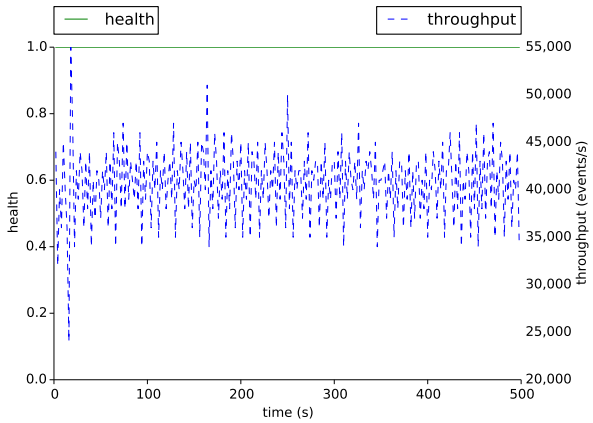
(c) Speed Layer Throughput



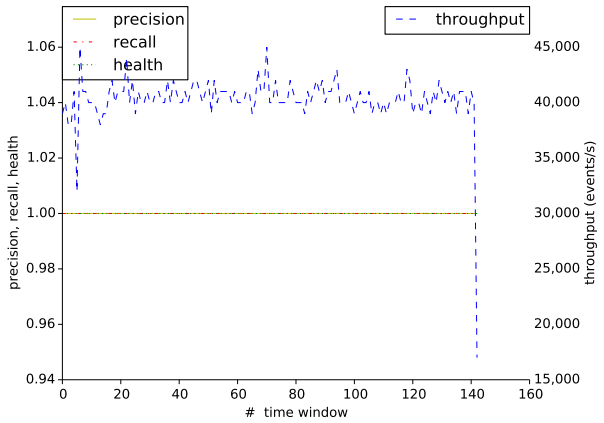
(d) Speed Layer Precision, Recall, Health

Figure 6.1: Results of the query "rainfall observed once an hour" applied to the data set SRBench Bertha without node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

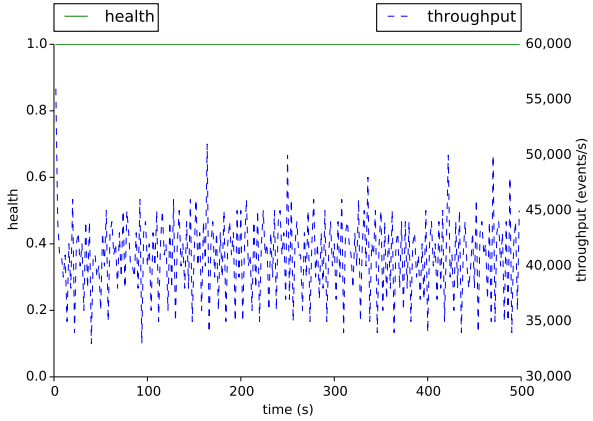
50



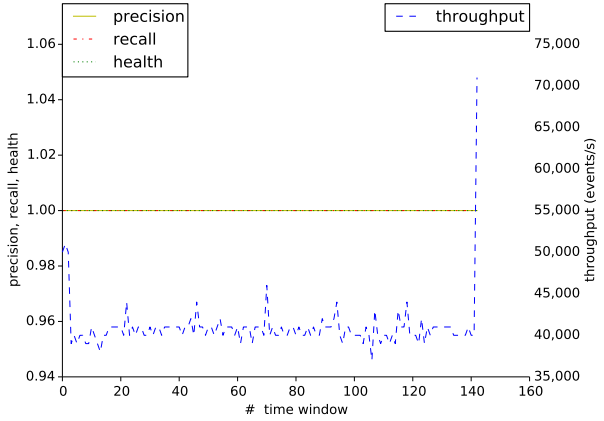
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



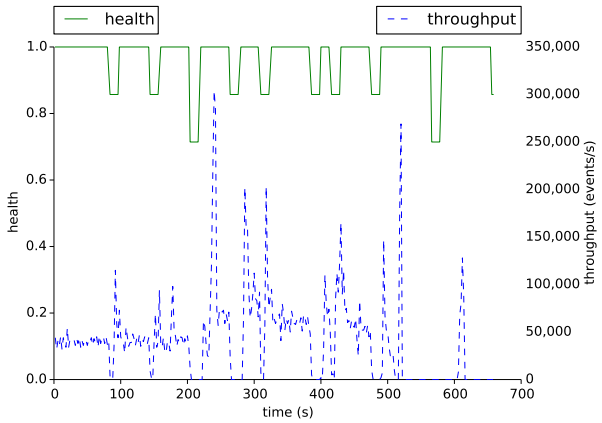
(c) Speed Layer Throughput



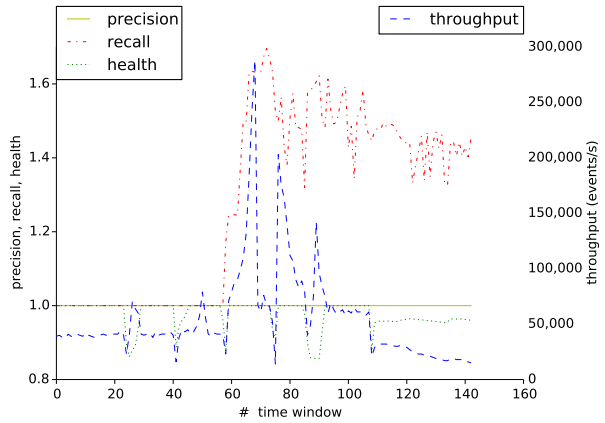
(d) Speed Layer Precision, Recall, Health

Figure 6.2: Results of the query "rainfall observed once an hour" applied to the data set SRBench Bertha with a throttled event input speed. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

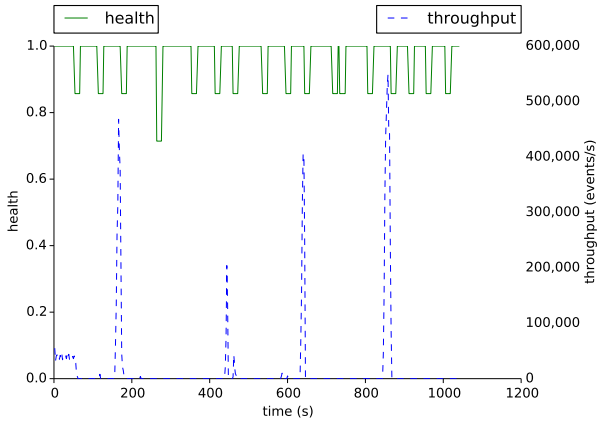
51



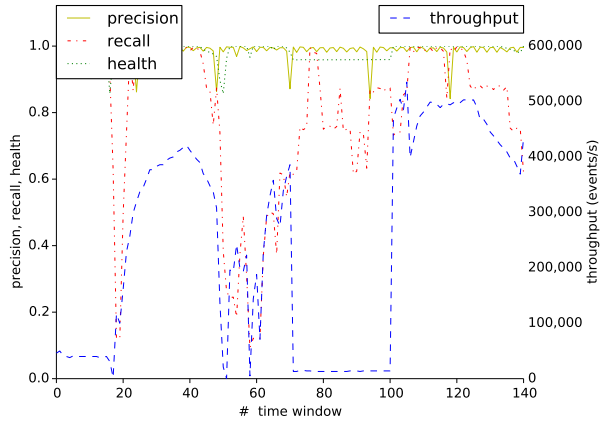
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



(c) Speed Layer Throughput



(d) Speed Layer Precision, Recall, Health

Figure 6.3: Results of the query "rainfall observed once an hour" applied to the data set SRBench Bertha with a throttled event input speed and node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

6.2.2 Broken Station Detection

This query tests the engines capability to detect missing data points. A station is defined as broken when it suddenly stops producing data. Information about the stability of stations is an important factor that can be inferred from the absent of data.

This query shows results similar to the query "rainfall observed once an hour". Therefore only the performance run and the node failure scenario is further described.

Performance Run

This experiment measures the results with the maximum possible input speed for the batch and speed layer. Table 6.4 highlights the overall key performance indicators of this experiment. In comparison with the query "rainfall observed once an hour" the speed layer is 1.4 times slower with the same data set. This indicates an increase of complexity to solve the task. The batch layer shows similar results in both queries. Figure 6.4 shows the key performance indicators of the performance run.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	174.057 s	avg: 115,000 (18.70 MB/s) max: 141,000 (22.939 MB/s) min: 14,000 (2.277 MB/s)	1.0	1.0	1.0
Speed	53.505 s	avg: 355,000 (57.76 MB/s) max: 429,000 (69.794 MB/s) min: 189,000 (30.748 MB/s)	1.0	1.0	1.0

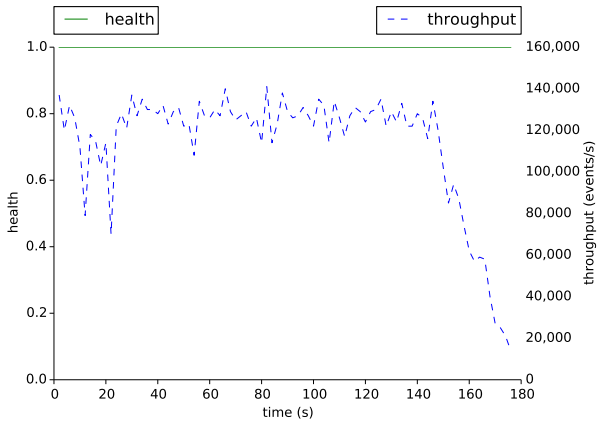
Table 6.4: Overall statistics of the query "broken station detection" run on the batch and speed layer without node failure simulation applied to the SRBench Bertha data set.

Node Failure

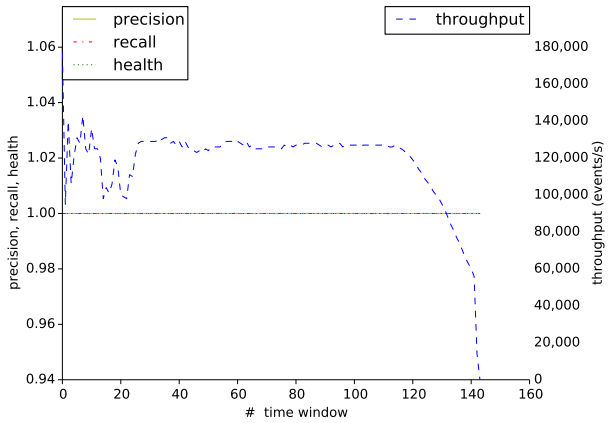
The node failure was tested using a throttled input source. Figure 6.5 highlights the key performance indicators of a throttled throughput applied to this query. In comparison with the query "rainfall observed once an hour" the throughput of the batch layer is almost twice as fast and the recall over the result set is over 2. Consequently the batch layer processed the amount of the data set twice during this experiment, while all generated results are precise. The substantial increase of recall and throughput started after the second node failure (see figure 6.6a). The first node failure did not cause an increase of recall, but a slight rise of throughput. This is further analyzed in Section 7.1.3.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	527.049 s	avg: 81,000 (13.17 MB/s) max: 347,000 (56.453 MB/s) min: 0	1.0	2.178	0.963
Speed	611.320 s	avg: 26,000 (4.23 MB/s) max: 442,000 (71.909 MB/s) min: 0	0.999	0.970	0.965

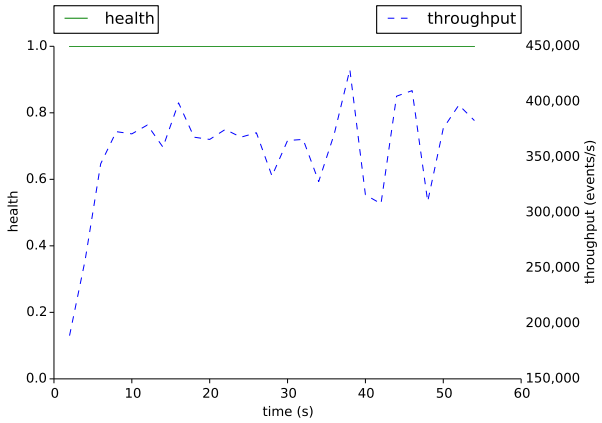
Table 6.5: Overall statistics of the query "broken station detection" run on the batch and speed layer with a throttled input stream and one node failure every 45 second interval for 15 seconds applied to the SRBench Bertha data set.



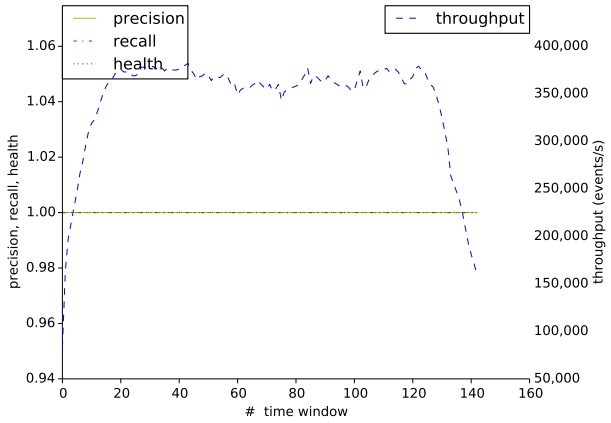
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



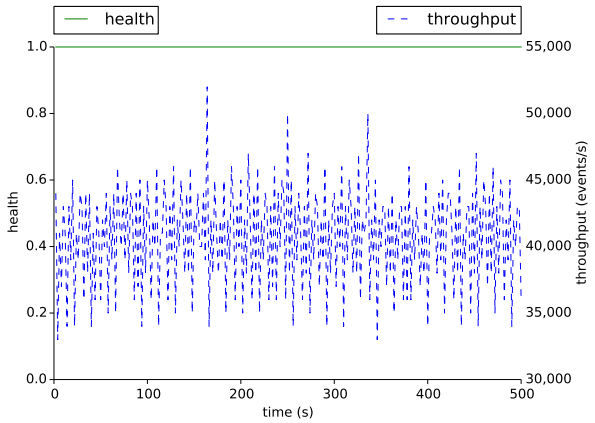
(c) Speed Layer Throughput



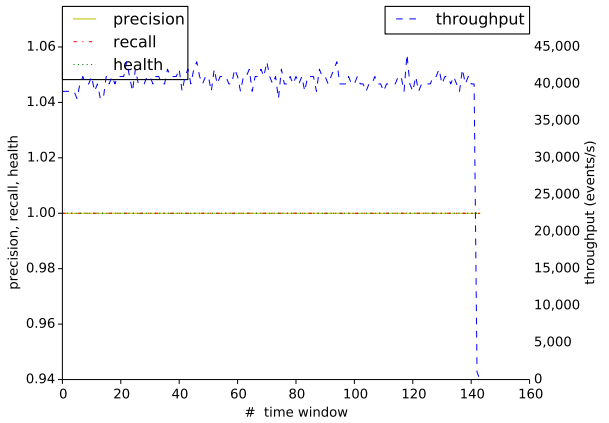
(d) Speed Layer Precision, Recall, Health

Figure 6.4: Results of the query "broken station detection" applied to the data set SRBench Bertha without node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

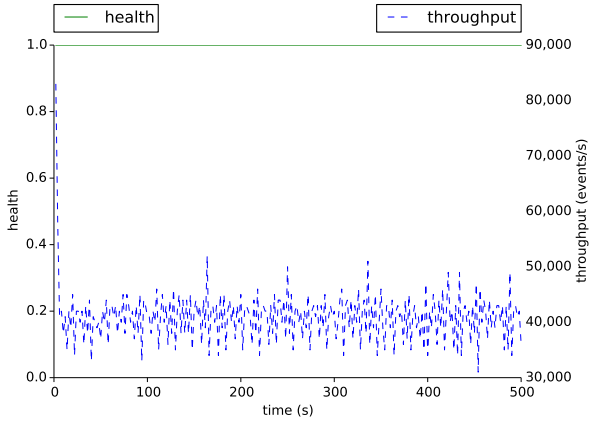
55



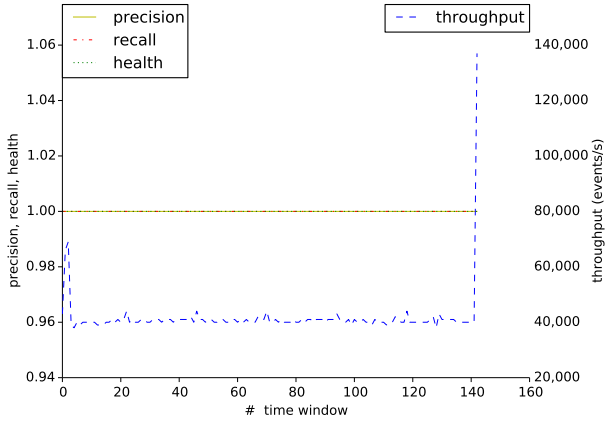
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health

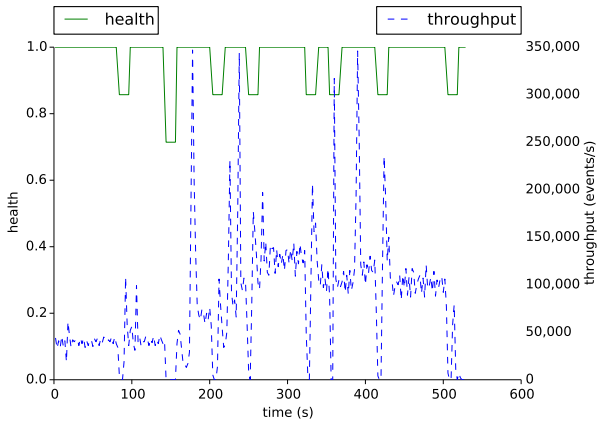


(c) Speed Layer Throughput

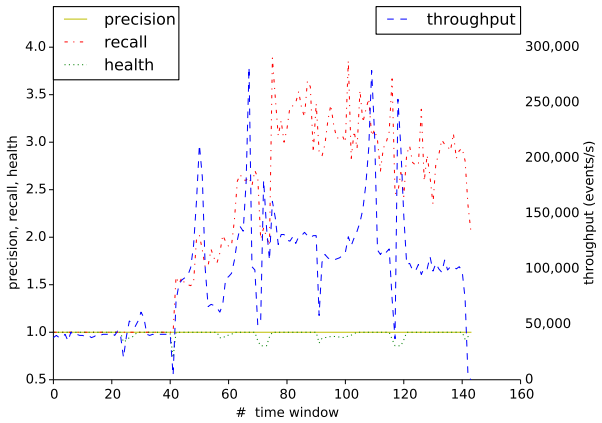


(d) Speed Layer Precision, Recall, Health

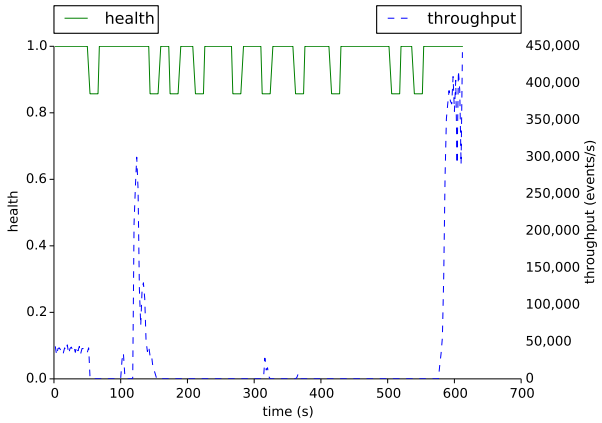
Figure 6.5: Results of the query "broken station detection" applied to the data set SRBench Bertha with a throttled event input speed. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.



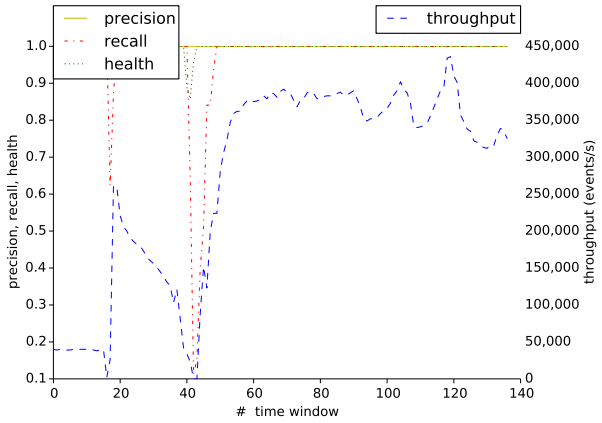
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



(c) Speed Layer Throughput



(d) Speed Layer Precision, Recall, Health

Figure 6.6: Results of the query "broken station detection" applied to the data set SRBench Bertha with a throttled event input speed and node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

6.3 DEBS Grand Challenge 2014 Data Set

Two queries were tested on the DEBS Grand Challenge 2014 data set: (i) load prediction and (ii) average load. The first query uses global storage in order to generate load predictions based on temporarily stored data. The latter query aggregates the load over a time window of 15 minutes.

6.3.1 Load Prediction

This query tests the ability of this architecture to aggregate events within time windows and provide predictions of future load. These predictions rely on temporary data. The speed layer stores such temporary data in a high performance key-value store described in Section 4.1.4 and the batch layer uses its internal local storage described in Section 4.1.2.

Two different setups were used for the experiment: (i) a performance run shows the key performance indicators of this query with the maximum input speed and no node failures and (ii) the node failure run shows the systems behavior when simulating node failures every hour for 5 minutes according to the mechanism explained in Section 4.5.2.

Performance Run

This experiment measures the maximum throughput possible with no node failure simulation. Table 6.6 shows the overall statistics of the batch and speed layer. On average the speed layer processed more than three times as much events per second as the batch layer while both layers produced fully accurate results. Also the recall is 1.0 for both layers. Both layers processed all events and the corresponding results were all precise.

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	46,901.208 s	avg: 86,000 (9.061 MB/s) max: 223,000 (23.494 MB/s) min: 0	1.0	1.0	1.0
Speed	14,534.837 s	avg: 278,000 (29.289 MB/s) max: 505,000 (53.204 MB/s) min: 26,000 (2.739 MB/s)	1.0	1.0	1.0

Table 6.6: Overall statistics of the query "load prediction" run on the batch and speed layer without node failure simulation using the DEBS data set.

Figure 6.7 illustrates the performance run. The throughput histogram of the batch and speed layer show a considerable decrease in throughput step by step during the experiment. Each decrease corresponds to the end of one partition. This is further analyzed in Chapter 7. The peak of the speed layer throughput is at 505,000 events/s compared to the peak of the batch layer at 223,000 events/s. The speed layer shows four sequential decreases in throughput starting after 10,000 seconds and reaches its

lowest point after 12,000 seconds. The batch layer shows a similar throughput curve after 30,000 seconds with three sequential drops.

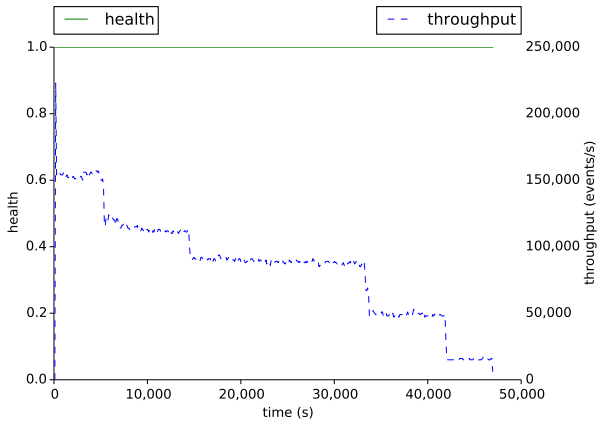
Node Failure

The node failure simulation shows the results of the batch and speed layer with one node failure every hour and a recovery time of 5 minutes. This experiment tests the engines ability to recover from node failures and shows its influence on the accuracy of the results. Table 6.7 shows the overall results of this experiment. The average throughput of the speed layer is similar to the performance run above. However the throughput of the batch layer is higher than in the performance run. In addition the batch layer shows a recall of more than 1.0 and a precision of 1.0. Consequently the batch layer produced more results than the baseline and all results were precise. Figure 6.8a shows a considerable increase of throughput after certain node failures. Other node failures cause slight fluctuations. This is further analyzed in Section 7.1.3. Figure 6.8b shows a sharp increase of recall and throughput after 250 and 2,300 time windows that match the occurrence of the first node failure. The figure shows a decrease of health only in the first 10 time windows and then remains stable around 0.998. Therefore almost all time windows were calculated while at least one node was not available. This effect occurred due to the partitioning mechanism that does not split the data set into equal sized buckets as described in Section 5.3.5.

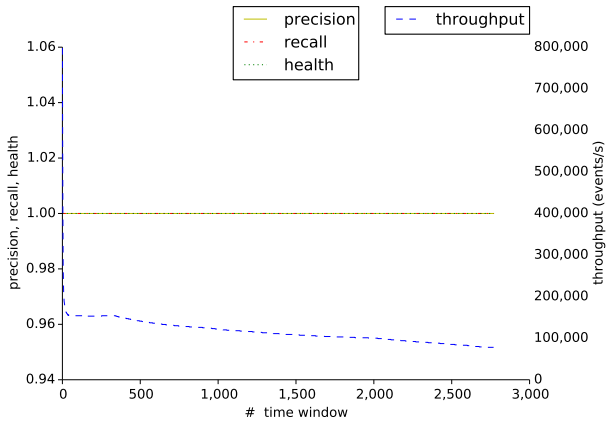
The speed layer finished the experiment 3.4 times faster than the batch layer, but the precision dropped to 0.411 with a recall of 0.41. Figure 6.7d shows the similarity of the precision and recall curve. The curves are aligned except for a few fluctuations in the precision. This experiment tested the engines ability to predict the load of a house. Therefore it is important to analyze the variance in addition to the precision and recall. Figure 6.9 shows the average variance as defined in Section 6.1 for each time window and the recall includes results with a lower difference than the average variance within the time window. The variance shows only slight fluctuations between 0.0 and 0.05 with two exceptions. Therefore results were not accurate, but an average variance of 0.0169 highlights a small difference to the baseline.

Layer	Time	Throughput (events/s)	Precision	Recall	Variance	Health
Batch	51,350.092 s	avg: 117,000 (12.327 MB/s) max: 323,000 (34.029 MB/s) min: 8,000 (0.842 MB/s)	1.0	1.619	0.0	0.998
Speed	14,728.679 s	avg: 275.000 (28.973 MB/s) max: 527,000 (55.522 MB/s) min: 0	0.411	0.41	0.0169	0.998

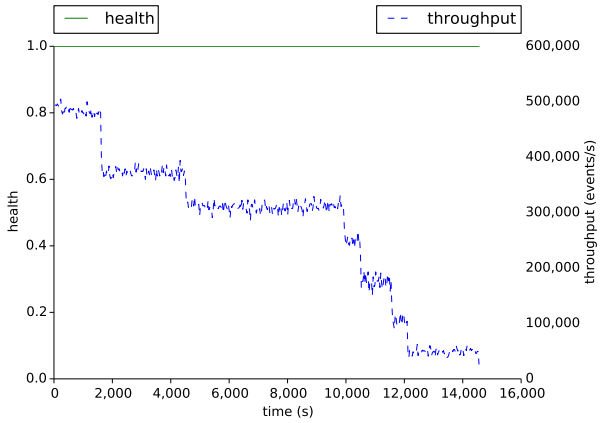
Table 6.7: Overall statistics of the query "load prediction" run on the batch and speed layer without node failure simulation using the DEBS data set.



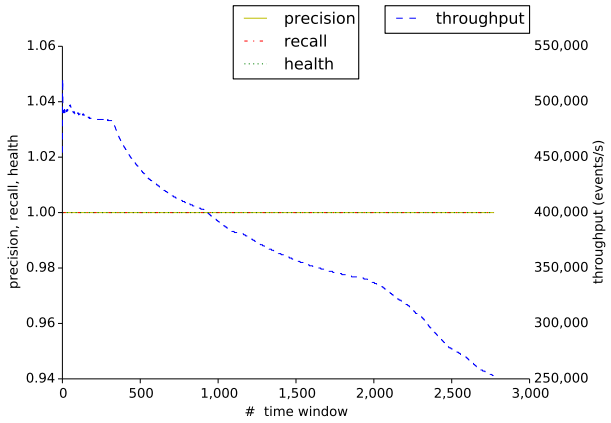
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health

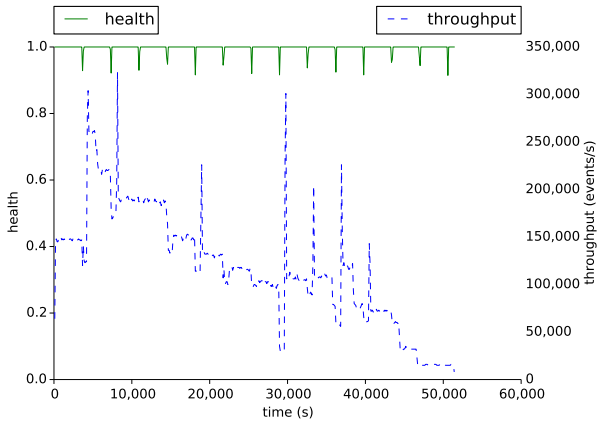


(c) Speed Layer Throughput

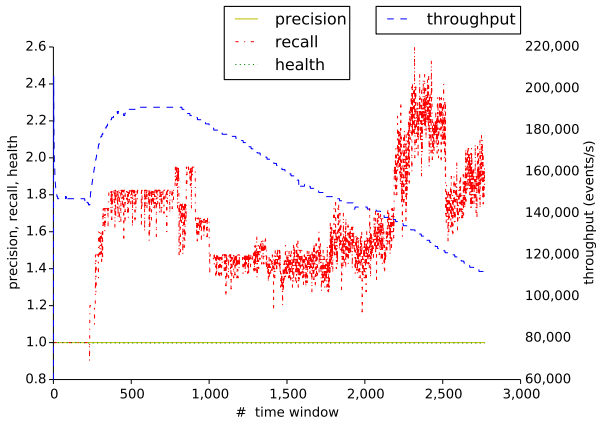


(d) Speed Layer Precision, Recall, Health

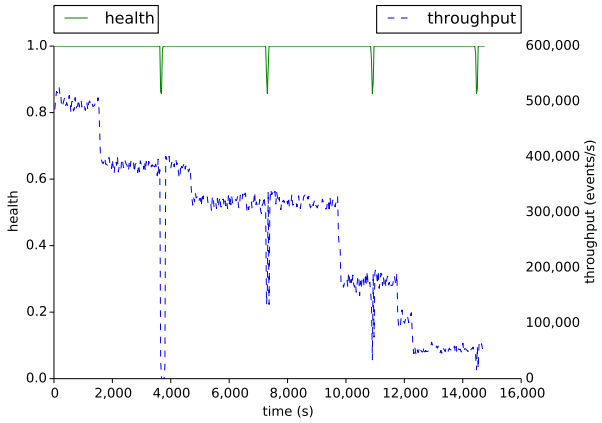
Figure 6.7: Results of the query "load prediction" applied to the data set DEBS without node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.



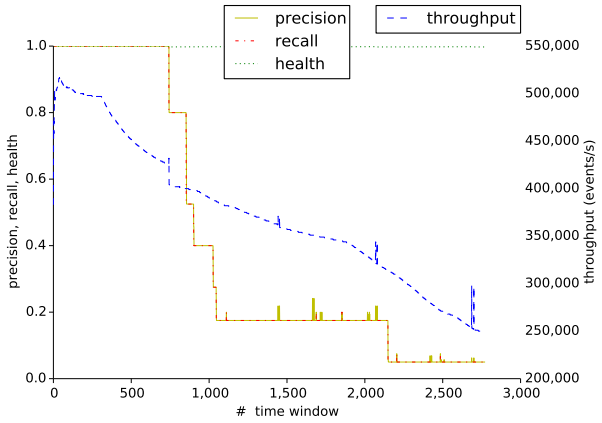
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



(c) Speed Layer Throughput



(d) Speed Layer Precision, Recall, Health

Figure 6.8: Results of the query "load prediction" applied to the data set DEBS with node failures. Figure a and c show the throughput/health histogram. Figure b and c highlight precision, recall, health and throughput for each time window.

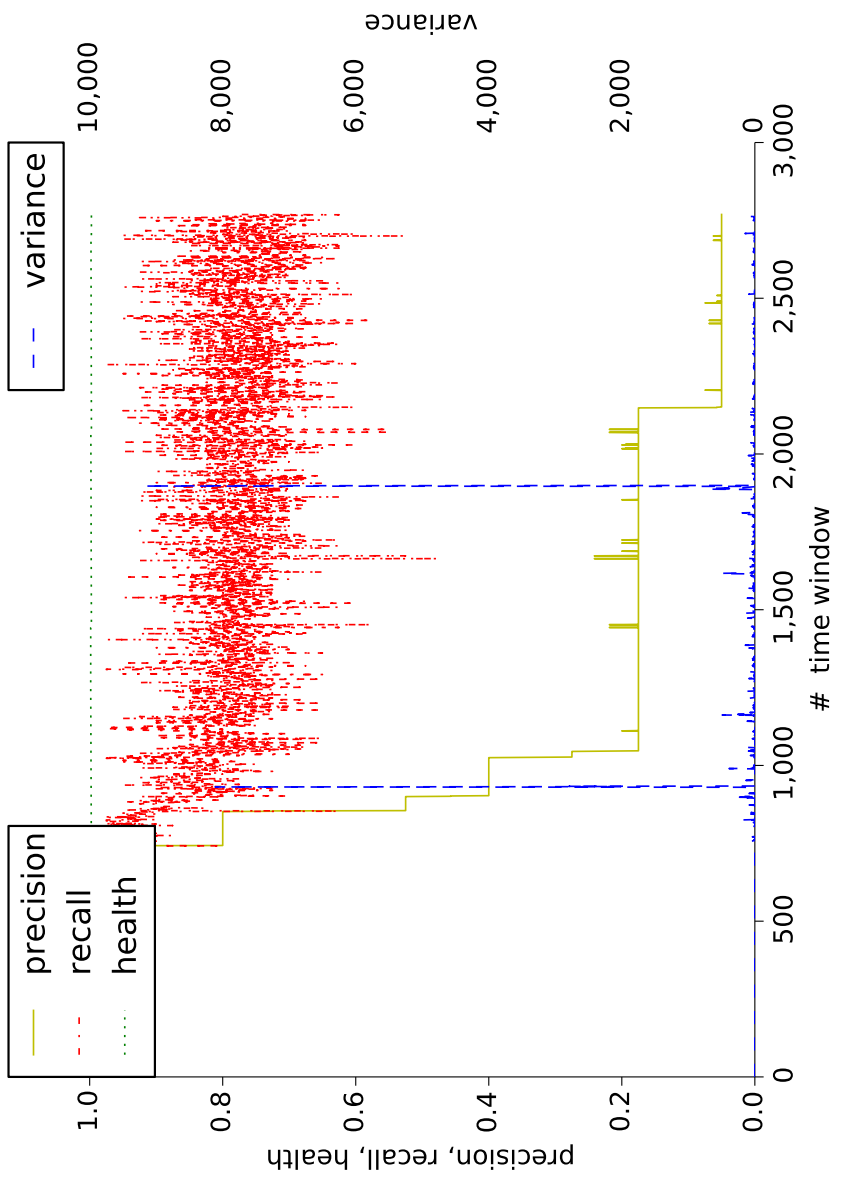


Figure 6.9: KPI of the query "load prediction" applied to the DEBS data set with node failures. In contrast to figure 6.8d this graph shows the variance and the recall excludes only values higher than the average variance of a time window from the result set.

6.3.2 Average Load

This query tests the engines ability to aggregate events within a time window of 15 minutes. In comparison with the query "load prediction" described in Section 6.3.1 this query does not rely on global state to calculate results; failures in results caused by stored data are not possible. In addition this query does not involve lookups to a key-value store for each time window.

First, the performance run experiment is presented which computes the results at the maximum possible throughput of this architecture. Second, the results of this query with node failures are described. The focus of this section lies in highlighting the difference between the results of query "load prediction" and "average load".

Performance Run

Table 6.8 highlights the overall statistics of this experiment. The performance run shows results of the batch layer similar to the results of the query "load prediction". The speed layer is slightly faster compared with the query "load prediction" and shows a higher average throughput. Figure 6.10 illustrates the KPIs of this experiment. The same decline in throughput is observable as highlighted in figure 6.7. Precision and recall is 1.0 for both the batch and speed layer. Although this query did not include a lookup of temporarily stored values the results are similar to the query "load prediction".

Layer	Time	Throughput (events/s)	Precision	Recall	Health
Batch	47,618.753 s	avg: 85,000 (8.955 MB/s) max: 185,000 (19.490 MB/s) min: 4,000 (0.421 MB/s)	1.0	1.0	1.0
Speed	13,455.517 s	avg: 301,000 (31.712 MB/s) max: 518,000 (54.574 MB/s) min: 45,000 (4.741 MB/s)	1.0	1.0	1.0

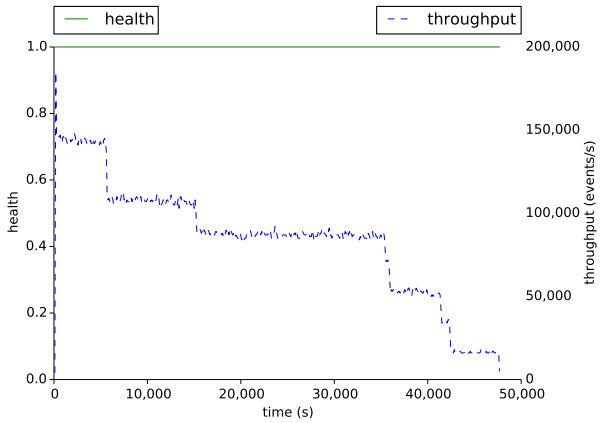
Table 6.8: Overall statistics of the query "average load" run on the batch and speed layer without node failure simulation using the DEBS data set.

Node Failure

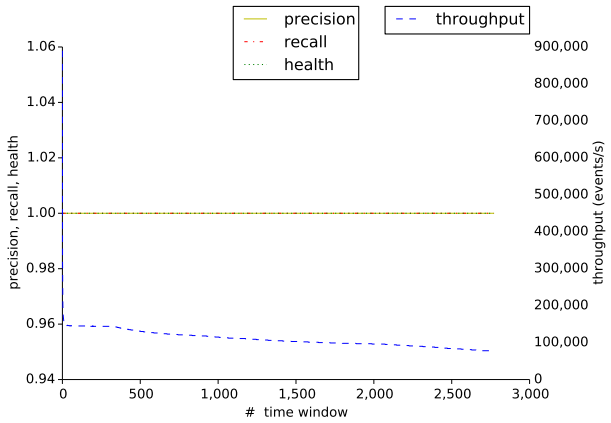
The node failure simulation highlighted in table 6.9 and figure 6.11 show similar results to the query "load prediction". Contrary to the "load prediction" query this query did not include temporarily stored data and therefore failures in results are not populated over multiple windows. However the precision and recall of the speed layer have only marginally improved. The variance curve in figure 6.12 fluctuates only slightly between 0.0 and 0.01 as compared to figure 6.9.

Layer	Time	Throughput (events/s)	Precision	Recall	Variance	Health
Batch	47,371.585 s	avg: 114,000 (12.600 MB/s) max: 291,000 (30.658 MB/s) min: 30,000 (3.161 MB/s)	1.0	1.625	0.0	0.998
Speed	13,615.096 s	avg: 297,000 (31.290 MB/s) max: 521,000 (54.890 MB/s) min: 0	0.419	0.418	0.0161	0.998

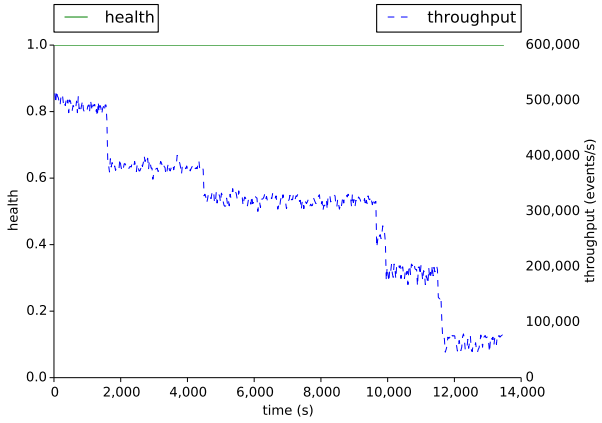
Table 6.9: Overall statistics of the query "average load" run on the batch and speed layer with node failure simulation using the DEBS data set.



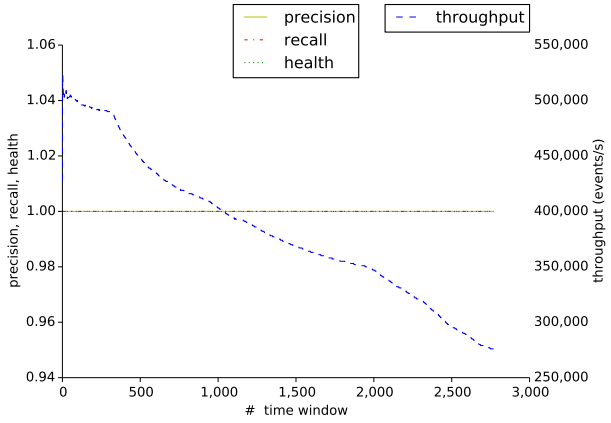
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health

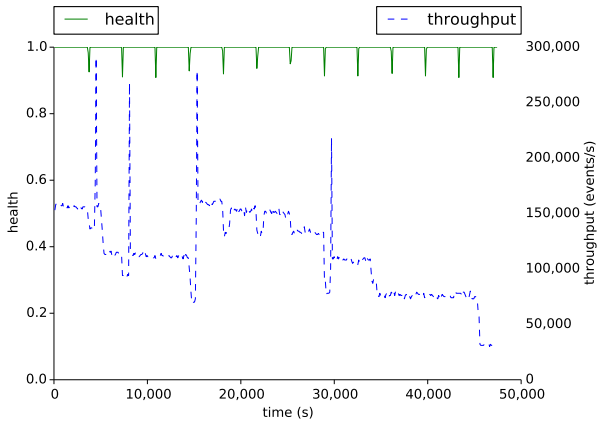


(c) Speed Layer Throughput

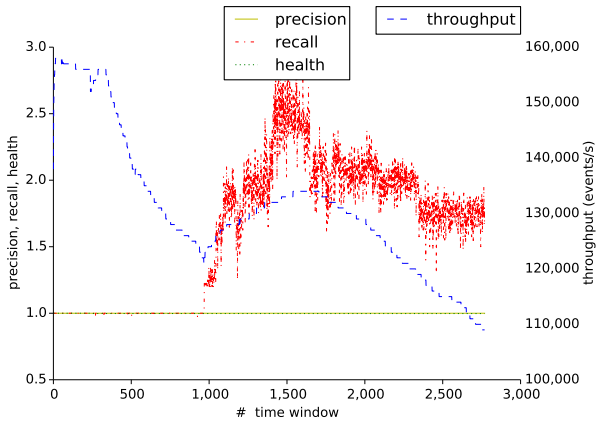


(d) Speed Layer Precision, Recall, Health

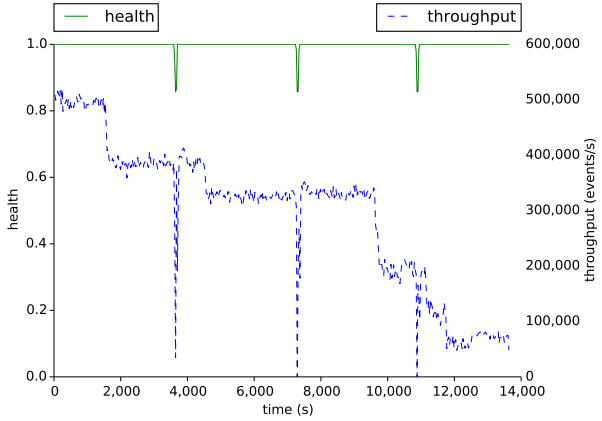
Figure 6.10: Results of the query "average load" applied to the data set DEBS without node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.



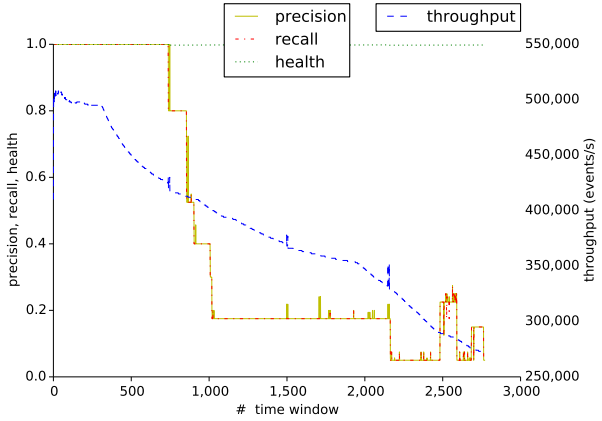
(a) Batch Layer Throughput



(b) Batch Layer Precision, Recall, Health



(c) Speed Layer Throughput



(d) Speed Layer Precision, Recall, Health

Figure 6.11: Results of the query "average load" applied to the data set DEBS with node failures. Figure (a) and (c) show the throughput/health histogram. Figure (b) and (d) highlight precision, recall, health and throughput for each time window.

Figure 6.12: KPI of the query "load prediction" applied to the DEBS data set with node failures. In contrast to figure 6.11d this graph shows the variance and the recall excludes only values higher than the average variance of a time window from the result set.

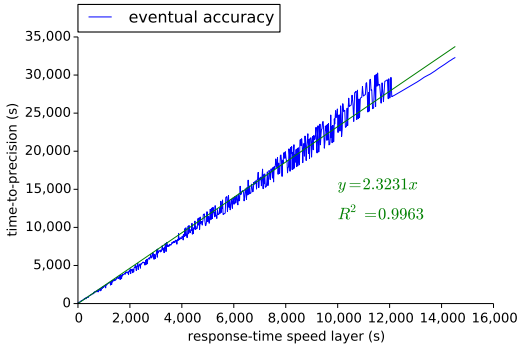
6.3.3 Eventual Accuracy

The lambda architecture provides eventual accuracy guarantees. An inaccurate result of the speed layer will eventually be replaced by the correct result of the batch layer. The two queries tested with node failure simulation described above are great examples to analyze this mechanism. Figure 6.13 shows the time it takes the batch layer to replace the possibly inaccurate results of the speed layer (time-to-precision) for both queries. Time-to-precision includes all results and makes no distinction between inaccurate and accurate. The batch layer will eventually replace all results, because there is no exact way to know which ones are inaccurate before the batch layer starts computing.

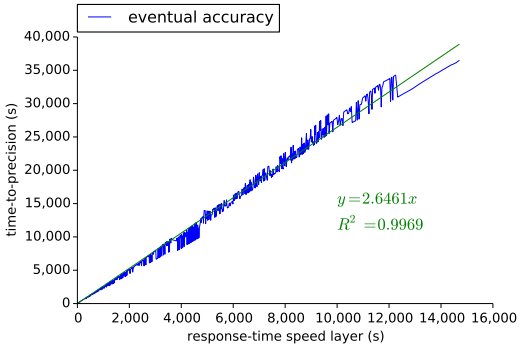
The skewed partitions of the DEBS data set do not have an impact on the eventual accuracy, because the speed layer and the batch layer operate on the same data set and therefore the same preconditions are imposed to both layers. The steady decrease in performance described in Section 6.3.1 is reflected in both layers and the time-to-precision is measured in relative terms.

The time-to-precision linearly increases with the response-time of the speed layer. The impact of node failures is clearly observable in both queries. The time-to-precision curve increases slightly faster in scenarios with node failures as highlighted with the linear regression analysis in figure 6.13.

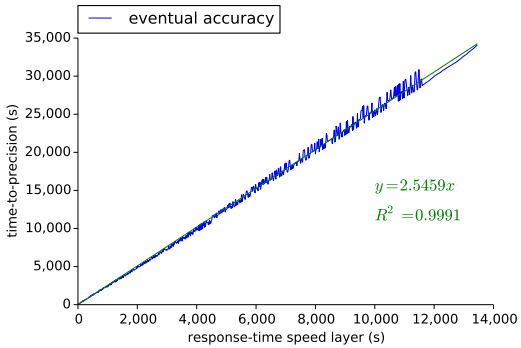
The result is further described with the query "load prediction". More than 4 Billion events were processed within 4.09 hours. The produced results include 59.9% inaccurate results that show an average variance of 0.0169 and in total only a very small fraction of the results was missing (recall is 0.001 smaller than precision). It took an additional 10.17 hours to replace all the inaccurate results with the accurate results of the batch layer and include the results that were not computed by the speed layer.



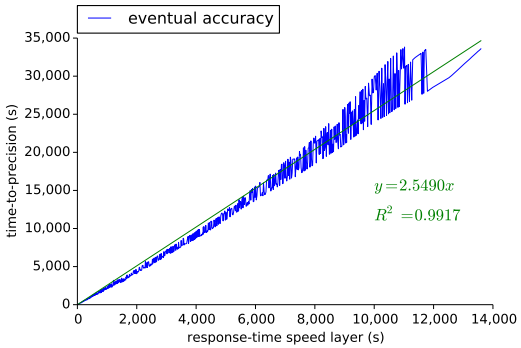
(a) Query "load prediction" Performance Run



(b) Query "load prediction" Node Failure



(c) Query "average load" Performance Run



(d) Query "average load" Node Failure

Figure 6.13: Highlights the eventual accuracy promise of the lambda architecture measured on the DEBS data set. The x-axis measures the response-time of the speed layer to produce results. The y-axis shows the time-to-precision e.g. the time it takes the batch layer to overwrite the possibly inaccurate results of the speed layer.

7

Discussion

The discussion interprets the results of the experiments and compares the concepts of the designed architecture to the proposed lambda architecture of Marz [44]. The speed layer computes results with high throughput, but these are inaccurate when node failures are introduced into the system. The batch layer on the other hand provides accurate results that are not diminished by node failures. The results described in chapter 6 prove that the combination of these capabilities is efficient and demonstrates the trade-off imposed by the guarantee of eventual accuracy.

7.1 Batch Layer

The results described in Chapter 6 prove the reliability of the batch layer in processing different tasks and data sets. Throughout all experiments the precision and recall of the results of the batch layer did not drop below 1.0. However introducing node failures into the system caused an increase of recall higher than 1.0. The following discussion highlights three key aspects. First, the message delivery guarantee of the batch layer is described and its impact on the recall is analyzed. Second, the micro-batch processing mechanism is analyzed and its shortcomings are compared to the recomputation approach. Third, the reaction to node failure and its recovery method is discussed.

7.1.1 Message Delivery Guarantee

The batch layer relies on the stream processing system Samza and its deeply integrated messaging queue Kafka. Both Kafka and Samza only provide at least once message delivery guarantee. As a consequence duplicate events may be sent to the computing task. The results described in Chapter 6 show a sudden increase of recall after node failures, but not every of them caused an increase of recall. This indicates two different effects of Samza: (i) the application master suffers a timeout or (ii) computing tasks are destroyed. The latter case will result in a restart of the corresponding tasks and the recovery to the last committed checkpoint. This causes duplicate results in case the node fails after the result is stored and before the checkpoint is committed. Samza does not provide a recovery mechanism for this scenario. Tanenbaum and Van Steen [54, chap. 8.5] propose a three-phase commit mechanism to solve such problems.

YARN will restart the application master of Samza in case it is destroyed due to node failures, but YARN only recognizes such a failure after a certain timeout. Therefore each task that is still alive will continue to produce results. When the application master is restarted it will stop all running tasks and submit new processing tasks. This goes back to the problem described above where a task is killed before it commits the latest checkpoint. In addition Samza will not use the latest committed checkpoint for each partition to restart a task, but roll back further to make sure no data is lost during a node failure.

A three-phase commit mechanism can reduce the probability of duplicate results. However, Samza's recovery strategy of the application master will always cause duplicate results and therefore a duplicate result check is required. A distributed storage with eventual consistency, such as the configuration of MongoDB during the experiments, does not provide a safe environment to prevent storing duplicate entries.

7.1.2 Micro Batch Processing

This architecture uses micro batch processing in contrast to the proposal of Marz [44] to use a one-time batch process to compute the results of the batch layer. The latency to receive the result based on a certain event may decreases by using a micro-batch processing in comparison with a one-time batch process. Although all results produced by the batch layer are accurate new problems arise such as duplicate results. The method to be selected for the batch layer strongly relies on the constraints of the computational task. If the batch layer relies on data with stream imperfections, such as delayed data, the micro batch processing method not only has to provide a recovery mechanism in case of failure, but also has to roll back in order to recalculate results based on delayed data. Stream imperfections are limitations of this architecture and further described in Section 8.4.

The experiments showed that the micro batch processing method delivers accurate results for the applied queries, but the recomputation approach would improve the systems capability to handle duplicate results and delayed messages.

7.1.3 Node Failure Recovery

YARN manages the discovery and recovery of node failures. A node failure is detected when no heartbeat is received during a configurable timeout. In case of failure YARN negotiates new resources with the application master or if the application master failed YARN restarts the application master. YARN sets such timeouts by default in the magnitude of minutes and timeouts below 5 minutes cause cluster instability. A system with low latency constraints has to enforce short timeouts in order to detect node failures. The batch layer of this architecture does not imply such a constraint and therefore the timeouts are not optimized.

Every node failure leads to a considerable decrease in throughput (see figure 6.8a, 6.11a, 6.3a and 6.6a). A node failure not only stops the computing tasks running on the failing node, but also stops the Kafka broker. The impact of node failures to a Kafka

cluster is explained in Section 4.5.1. In case a Kafka broker fails the Kafka cluster needs time to possibly reelect new leaders for partitions or decide to skip a replication broker until it gets available again. This not only impacts the tasks failed due to node failure, but also influences the tasks running on healthy nodes. As described in Section 5.2 Kafka was started on 7 nodes and configured with a replication factor of 3. Therefore one node failure impacts at least two other nodes and leads to a short timeout of over 40 % of the Kafka cluster. All tasks are stopped and restarted if the failing node was running the application master of Samza.

After a node failure and a corresponding drop in throughput the batch layer may catch up with the produced data and the throughput increases dramatically. Figure 6.8a shows node failures followed by an increase of throughput and node failures without any impact on throughput. In case a message consumer is destroyed, but the producer can still send messages at the same speed, the consumer lags behind and may catch-up after a restart.

7.2 Speed Layer

The speed layer is optimized for real-time computing and does not impose any message delivery guarantees. In particular messages lost during node failures or other unexpected behavior are not recoverable. The results of the experiments show a considerable higher throughput in the speed layer than in the batch layer. The batch layer relies on persistent messaging and failure recovery while the speed layer discards messages during failures and is not subjected to the overhead enforced by storing each message in a persistent manner and managing recovery points. As a result the occurrence of node failures cause a loss of accuracy in the results. The damage imposed by the loss of messages depends strongly on the computational task. This section further analyses the quality of service and node failure recovery.

7.2.1 Quality of Service

The maximum throughput of the speed layer over all experiments is 684,000 events/s as illustrated in figure 6.1c and even with this amount of throughput the speed layer produced accurate results. However the throughput of the speed layer is not constant over all experiments e.g. the query "rainfall observed once an hour" shows an average throughput of 521,000 events/s, while the query "broken station detection" only reaches an average of 355,000 events/s during the performance runs. The possible throughput strongly correlates with the complexity of the computational task. In order to apply QoS constraints on the speed layer the impact of the computational task has to be deeply considered. Also, unexpected failures in the computation nodes may lead to different outcomes regarding the precision and recall of the result. The comparison of these KPIs from the DEBS data set with the SRBench data set shows that the latter experiments were less influenced by failures with regard to precision and recall. The precision curve in figure 6.3d fluctuates between 0.82 and 1.0, while a node failure during the query "load prediction" applied to the DEBS data set initiates a steady decrease of precision.

The query "rainfall observed once an hour" generates a boolean result that is either true or false, while the "load prediction" query generates a floating point result. The latter was further analyzed with the variance of the difference to the correct result as illustrated in figure 6.9. The average variance of the results is 0.0169 and indicates only very small impreciseness over all time windows. In case the computational task allows for higher latency the speed layer could be enhanced to provide a certain degree of recovery guarantees.

The current implementation of the speed layer does limit the QoS constraints to the maximum throughput possible with no message delivery guarantee. However, this architecture can be extended to allow fine-grained QoS constraints. To leverage the real-time computing aspect of the speed layer it is possible to empower the speed layer to only process events within certain latency constraint and withdraw any event that exceeds this latency. There are two options for such a behavior: (i) the in-memory queue of the coordinator can be extended to accept a retention policy for messages similar to the zeromq implementation or (ii) the spout embeds a mechanism to only forward messages within an adequate retention policy.[1] Such an implementation strongly depends on the QoS constraint of the computational use case and therefore needs further analysis regarding its impact on the precision of the results.

The precision and recall of the query "load prediction" tested under node failures show a consistent decline (see figure 6.8c). Marz [44] proposal of the lambda architecture includes a serving layer, which integrates the results from the batch and speed layer. The serving layer provides a way to access results from the batch layer with low latency. The serving layer was not implemented in this architecture, but could lead to better results regarding the query "load prediction". This query takes data from the past three days into account to generate the prediction of the load. Hence the results of both layers could be accessed in order to generate more precise results. This was not implemented in the experiments, since they were run in an offline processing fashion with the highest throughput possible and not according to the actual timestamp of the data. When dealing with real-time data the introduction of an additional serving layer that allows for fast access to the results of the batch layer may produce better results. This can be realized when the batch layer can keep up the pace with the necessary computational requirements.

Section 6.3.3 shows the effect of eventual accuracy. The inaccurate results from the speed layer are eventually replaced by the precise results of the batch layer. Throughout the experiments the time-to-precision increased steadily. A possible hypothesis is that the time-to-precision increases in the beginning and at a certain point evens out to reflect the performance difference between the batch and speed layer. However, the time-to-precision does not even out, but rather linearly increases with the response-time of the speed layer. The experiments with node failure simulation show a slightly faster increase in time-to-precision due to the recovery mechanism of the batch layer that first rewinds back to the last checkpoint and then continues processing.

[1] http://zeromq.org/

7.2.2 Node Failure Recovery

Node failure simulations were run in two different flavors as described in Section 5.4. Experiments run with the SRBench data set show a fast drop in throughput and only limited capabilities to recover from node failures. Figure 6.6c further illustrates this behavior and also shows that the speed layer was not able to recover from node failure for more than 100 seconds. The nimbus service of Storm is responsible for detecting node failures and possibly starting the task on another machine. It uses a different implementation than corresponding component of the batch layer. The running topology freezes after a node failure and within a time range of 10 to 30 seconds the tasks of a topology are reshuffled to different nodes. Storm not only reshuffles the tasks of the failed machine, but takes all tasks into account. The same behavior can be observed when the node becomes available again, which explains the throughput histogram 6.6c and 6.3c. The speed layer shows a higher processing time than the batch layer in table 6.3 and 6.5 due to the topology freeze after a change in the cluster.

The node failure experiments applied to the DEBS data set show different results than the SRBench data set. The experiments were run with a higher node failure interval and therefore the topology freeze did not impact the throughput as severely. But the node failure causes noticeable lower precision and recall as described in Section 7.2.1.

7.3 Data Partitioning

The data partitioning using a hash bucket algorithm as described in Section 5.3.5 causes skewed partitions in the DEBS data set. The throughput histograms in Section 6.3.1 and 6.3.2 show a sequential decrease of throughput during the experiment, due to skewed partitions. The partitioning key of the DEBS data set was defined according to the query to solve. The partitioning method divided the data set into 8 partitions marked by the house number ranging from 0 to 40. The SRBench data set shows a better result regarding the partitioning method. The data set was divided into 8 partitions marked by the station name. Each data set holds the data of hundreds of different stations.

8

Limitations

This chapter describes the limitations of this architecture based on the concepts described in Chapter 4 and the results presented in Chapter 6. Each identified limitation is discussed in the following sections. First, the limitation is identified and its importance is highlighted. Second, the nature of the limitation is further analyzed and the choices that caused the limitation are justified. Third, a suggestion to overcome each limitation is provided.

8.1 Single Point of Failure

The architecture has a single point of failure in the speed layer and batch layer. In both cases the master node of the orchestration service is concerned, but they show different reactions to a node failure. The batch layer relies on the YARN resource manager (see section 4.1.1) and the corresponding master service called resource manager. The community recently introduced high availability features for the resource manager, but due to the tight integration within the architecture an update of was not possible during the time frame this thesis.[1] A failure of the node that runs the resource manager would provoke the complete shutdown of all computing tasks of the batch layer. The speed layer relies on Storms internal orchestration service called nimbus. Nimbus does currently not provide high availability features, but it follows a different approach compared with the YARN resource manager. In case the node running nimbus is unavailable all worker nodes will resume processing and wait for the nimbus service to become active again. However, during this time the computing tasks are not managed by the nimbus service and in case of a second node failure the tasks will not get restarted. The community currently works on a solution to overcome this limitation.[2]

Due to the lack of high availability features in YARN and Nimbus the master node of the experiments was not concerned in node failure simulations. Hence, the ZooKeeper service was not run in a high availability quorum although it offers these features. Running ZooKeeper in a high availability setup could further influence the performance measurements.

[1] https://issues.apache.org/jira/browse/YARN-149
[2] https://issues.apache.org/jira/browse/STORM-166

8.2 Concurrent Node Failures

The experiments did not include concurrent node failure simulations. The batch layer for example could produce inaccurate results in case of multiple node failures at the same time. A theoretical discussion is provided for the batch layer in Section 4.3.4, for the speed layer in Section 4.4.2 and for the coordination layer in Section 4.5.1. However, this discussion is based on the promise of the underlying services and is not qualified with experiments. The design of the node failure simulation discussed in Section 4.5.2 enables clients to specify the amount of concurrent node failures to simulate. Therefore the architecture provides the capabilities to further investigate concurrent node failures, but during the time frame of this thesis it was not possible to present such results.

8.3 Partitioning

The following assumptions were made regarding the partitioning of the data as described in Section 5.3.5: (i) the highest number of parallelism throughout all experiments is 8 (ii) the number of partitions and therefore the number of parallelism can not be changed during runtime and (iii) the data sets are always partitioned using a constant hashing algorithm. The number of parallelism noticeably influences possible throughput in the experiments. In particular the message queue of the batch layer shows a considerable decrease of performance for each additional partition.[3] Due to time restrictions and resource limitations 8 was chosen as the maximum number of parallelism. However, the design of the architecture is not bound to a certain number of parallelism.

The number of partitions cannot be changed during runtime, because the underlying stream processing system Samza distributes tasks according to the partitioning scheme during bootstrap. An increase of partitions during runtime is desired in order to scale horizontally when the data volume increases. Currently the only solution to overcome this problem is to restart the computation job. This will cause higher latency in the batch layer, but since the batch layer always lags behind the speed layer this problem is minor.

The data partitioning method does not guarantee equal distribution of data and therefore influences the scalability of the computational tasks as described in Section 7.3. The algorithm to partition the data described in Section 5.3.5 enables clients to specify a partitioning key and abstracts the partitioning logic throughout the components of the architecture. The method was tested with the SRBench data set, but not with the DEBS data set. There is no one-fits-all solution for the problem of data partitioning and pre-studies are necessary to determine an appropriate method. Hence the partitioning component of this architecture should enable clients to provide their data specific

[3] A pre-study based on the SRBench Bertha data set concluded the following throughput to produce messages with respect to the number of partitions: 1 Partition 50,000 events/s, 8 partitions 20,000 events/s and 20 partitions 1,000 events/s. Kafka has to keep a file handler open for each partition it manages. If one Kafka broker manages multiple partitions, the corresponding 8 file handlers will be always open and only allow sequential writes for certain batch sizes.

implementation to use throughout all modules of this architecture.

8.4 Stream Imperfections

Conventional database systems allow queries to span over the full data set, but stream oriented systems process data as it flows through the system. Events may arrive delayed or out-of-sequence. The batch layer of this architecture computes results using an incremental algorithm that processes new incoming data in a stream oriented fashion. The batch layer as described in Section 7.1.2 currently discards delayed events. The data sets used to run the experiments were sorted by time and did not include any delayed events, but possible use cases for this architecture may include delayed data. Depending on the computational task the delayed message may be processed without access to historical data. For example the query "rainfall observed once an hour" can include the delayed message in its result set. The query "load prediction" on the other hand has to access the full time window of the delayed message in order to provide accurate predictions. This involves a book keeping mechanism to roll back to the messages of the corresponding time window. Alternatively, the serving layer may be used to store all relevant data in order to provide predictions. For example a query that has to calculate an average value over a 1 hour time window could store the sum and count of events and therefore allows for updates in case of delayed messages. However, this solution has to be further assessed with regard to the volume of storage necessary to produce results.

9

Future Work

This thesis provides an implementation design of the speed and batch layer for the lambda architecture and shows its corresponding key performance indicators for a selected set of use cases. Further research is necessary in order to qualify and quantify the eventual accuracy guarantee of this architecture. A selected list of possible future work is highlighted.

The experiments in Chapter 5 show the trade-off of the eventual accuracy guarantee provided by this architecture. It is necessary to further investigate in this promise to better understand its impact. A desirable outcome includes the time-to-precision for different levels of accuracy. A possible hypothesis is that with higher accuracy in the speed layer the time-to-precision would decrease due to the impact of fault-recovery on the efficiency of the speed layer.

The batch layer is designed to use an incremental algorithm to process new data in batches. Recent work shows the relation of batch sizes to performance and introduces a mechanism to dynamically adjust the batch size according to the observed workload [23]. The model is based on one producer and one consumer communicating through a messaging queue with batch support. Its adjustment algorithm is based on fixed-point iteration and overcomes the challenges of changing workload, noise and unpredictable operational problems. The dynamic adjustment of the batch size proves to reduce latency in their experimental design. However, it is necessary to further analyze the impact of the batch size to the latency in a distributed environment with more processing steps and multiple machines, because such a scenario involves more network activity and impacts the latency considerably.

The batched message processing paradigm also requires further research of scalability. As highlighted in Section 7.1.3 the batch layer rolls back to the latest recovery point in case of failure and starts reprocessing. In case of failures in the application master, the recovery point is set further back than the last checkpoint to ensure precise results. In such a scenario it would be possible to continue processing from the last known checkpoint and to leverage the scalability of the system by starting new tasks to recompute the data further back than the latest checkpoint.

A theoretical discussion on the scalability of the coordination, batch and speed layer is provided in Chapter 4, but it is yet unclear how the scalability will affect the throughput and latency of the system by increasing or decreasing the following properties: (i) number of machines (ii) number of partitions (iii) number of computational tasks (iv)

replication factor and (v) size of the message. For example small messages influences the performance considerably, due to the operational overhead of the systems and the replication factor increases the bandwidth used to coordinate between different machines.

The speed layer of this architecture processes messages with the maximum throughput possible and delivers precise results if no node failures are involved. Certain use cases and the respective volume of data further constrain the speed layer to guarantee results within a certain maximum latency. Such a scenario involves further control mechanism in the speed layer as discussed in Section 7.2.1. This architecture provides the possibilities to simulate burst in data as described in Section 4.2.1 and controls the data flow of the incoming messages. However, more control mechanisms are needed in order to set fine-grained QoS constraints on messages and the corresponding computational result. Lohrmann et al. [41] analyzed common design patterns of open source stream processing systems with regard to the QoS goal latency. In particular the output buffers and thread/process model influence the latency of the studied systems. They introduce a model to trade throughput in favor of latency by introducing two new techniques: (i) adaptive output buffer sizing and (ii) dynamic task chaining. It allows clients to specify upper latency constraints for critical series of vertices and edges within the processing graph. These constraints are supplied during the bootstrap of the task and do not adapt to changes of the users needs. Storm for example does not allow the user to define a maximum latency for a computational task and the user has to fine tune configuration options that impose these requirements. Work is in progress to automatically adjust configuration parameters in order to provide lower latency.[1] The lambda architecture imposes further constraints regarding the latency and messages may be dropped in favor of fast response times. The real-time processing system should be able to react to changing requirements and allow for more fine-grained control to optimize the trade-off between latency and accuracy.

Latency is not only influenced by the output buffer size and the thread/process model, but also by the network layout and the distribution of tasks. The proposed architecture leverages different services and frameworks with loose coupling. The producer/consumer pattern is used to synchronize these services, but improvements on the communication channels between the services are necessary. For example location awareness is a desirable property for the distribution of tasks. For example a computing task of the batch layer should be located physically close to the Kafka leader of the partition it consumes from. Also communication within services is not optimized. For example a Storm topology with multiple computing steps may distribute the tasks of one pipeline on different machines leading to higher latency through network communication. Fischer et al. [29] propose a graph-partitioning based method to minimize the total number of messages sent over network within a Storm topology. To minimize the network layer of the lambda architecture the task scheduling of the speed layer and the interfaces to other services are equally important.

[1] https://issues.apache.org/jira/browse/STORM-31

10

Conclusions

The lambda architecture promises a new solution to cope with an ever increasing volume of data and the need to make decisions based on real-time analytics. Current research in this field centers around the idea to optimize the horizontal scalability and to lower the latency of distributed systems in order to fulfill these needs. The lambda architecture takes a different approach and introduces the idea of eventual accuracy. Only limited research is available that discusses the impact of the conflict of goals imposed by this promise and there is no reference implementation that forms the basis for further analysis. The contribution of this thesis is to take a step forward and broaden the discussion of the lambda architecture.

The designed architecture is based on open source stream-processing, orchestration and provisioning software and the following implementation is provided for the two computing layers of the lambda architecture. The batch layer follows an incremental approach that first persistently stores incoming messages in a high availability message queue. This queue provides replay capabilities to recover from unexpected behavior. The computation tasks are bound to the partitioning scheme of the queue and both are highly scalable. The speed layer consumes messages from an in-memory queue that does not provide recovery mechanisms. In case of unexpected behavior such as machine failures the speed layer drops messages and continues with the most recent data available.

The architecture includes components to coordinate, monitor and measure the computing jobs. The coordination component provides the interface to send data into the system. The orchestration component manages the resources and includes an automatic deployment of the architecture. The analysis of the results and the monitoring of the processes is managed in the service layer.

An evaluation of experiments including the SRBench and DEBS Grand Challenge 2014 data set measured the capabilities of the designed architecture and stressed its behavior on an unreliable infrastructure. The results of the batch layer were throughout all experiments completely accurate and replaced the inaccurate results from the speed layer caused by node failure. The speed layer on the other hand generated results up to five times faster than the batch layer.

The main limitations of the designed architecture are the incapability to handle stream imperfections such as delayed messages and the partitioning of messages based on consistent hashing does not guarantee equal distribution of data.

The imposed QoS constraints of the designed architecture are currently not configurable. Future research is necessary to better understand the impact of timely but inaccurate results and investigate the possibilities to dynamically adopt QoS constraints based on the accuracy and latency level.

References

[1] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al. (2005). The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289.

[2] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139.

[3] ACM (2014). Debs grand challenge 2014. *http://www.cse.iitb.ac.in/debs2014/?page_id=42*, Retrieved 2014-08-14.

[4] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044.

[5] Andrade, H., Gedik, B., and Turaga, D. (2014). *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

[6] Apache Foundation. Apache kafka configuration table. *http://kafka.apache.org/08/configuration.html*, Retrieved 2014-08-14.

[7] Apache Foundation. Config class. *https://storm.incubator.apache.org/apidocs/backtype/storm/Config.html*, Retrieved 2014-08-14.

[8] Apache Foundation. Performance results. *http://kafka.apache.org/07/performance.html*, Retrieved 2014-08-14.

[9] Apache Foundation. Storm, distributed and fault-tolerant realtime computation. *https://storm.incubator.apache.org*, Retrieved 2014-08-14.

[10] Bernhardt, T. and Vasseur, A. (2007). Esper: Event stream processing and correlation. *ONJava, in http://www. onjava. com/lpt/a/6955, O'Reilly.*

[11] Beyer, K. S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.-C., Ozcan, F., and Shekita, E. J. (2011). Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*.

[12] Bockermann, C. and Blom, H. (2012a). The streams framework. Technical Report 5, TU Dortmund University.

[13] Bockermann, C. and Blom, H. (2012b). The streams framework. Technical report, Technical Report 5, TU Dortmund University, 12 2012.

[14] Brewer, E. A. (2000). Towards robust distributed systems. In *PODC*, page 7.

[15] Carlson, J. L. (2013). *Redis in Action*. Manning Publications Co.

[16] Cattell, R. (2011). Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27.

[17] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM.

[18] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM.

[19] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[20] Chodorow, K. (2013). *MongoDB: the definitive guide.* " O'Reilly Media, Inc.".

[21] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). Mapreduce online. In *NSDI*, volume 10, page 20.

[22] Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62.

[23] Das, T., Zhong, Y., Stoica, I., and Shenker, S. (2014). Adaptive stream processing using dynamic batch sizing.

[24] Dean, J. and Ghemawat, S. (2003). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[25] Dean, J. and Ghemawat, S. (2011). leveldb–a fast and lightweight key/value database library by google. *https:// code.google.com/ p/ leveldb*, Retrieved 2014-08-14.

[26] EsperTech Inc. (2014). Esper - complex event processing. *http:// esper.codehaus.org*, Retrieved 2014-08-14.

[27] Etzion, O. and Niblett, P. (2010). *Event processing in action*. Manning Publications Co.

[28] Fan, W. and Bifet, A. (2013). Mining big data: current status, and forecast to the future. *ACM SIGKDD Explorations Newsletter*, 14(2):1–5.

[29] Fischer, L., Scharrenbach, T., and Bernstein, A. (2013). Network-aware workload scheduling for scalable linked data stream processing. In *International Semantic Web Conference (Posters & Demos)*, pages 281–284.

[30] George, L. (2011). *HBase: the definitive guide*. " O'Reilly Media, Inc.".

[31] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.

[32] Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59.

[33] He, B., Yang, M., Guo, Z., Chen, R., Su, B., Lin, W., and Zhou, L. (2010). Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM.

[34] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11.

[Hypertable Inc.] Hypertable Inc. Hypertable. *http://hypertable.com/*, Retrieved 2014-08-14.

[Kreps] Kreps, J. Idempotent producer. *https://cwiki.apache.org/confluence/display/KAFKA/Idempotent+Producer*, Retrieved 2014-08-14.

[37] Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*.

[38] Kwon, Y., Balazinska, M., and Greenberg, A. (2008). Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment*, 1(1):574–585.

[39] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

[LMAX Trading] LMAX Trading. Disruptor library. *https://github.com/LMAX-Exchange/disruptor*, Retrieved 2014-08-14.

[41] Lohrmann, B., Warneke, D., and Kao, O. (2014). Nephele streaming: stream processing under qos constraints at scale. *Cluster computing*, 17(1):61–78.

[42] Love, R. (2013). *Linux system programming: talking directly to the kernel and C library*. " O'Reilly Media, Inc.".

[43] Manning, J. (2004). *Apache Storm*. Signet.

[44] Marz, N. (2013). *Big Data: Principles and best practices of scalable realtime data systems*. O'Reilly Media.

[45] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30.

[46] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM.

[47] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM.

[Project Netty] Project Netty. Netty project. *http://netty.io/index.html*, Retrieved 2014-08-14.

[49] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. (2013). Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM.

[50] Robak, S., Franczyk, B., and Robak, M. (2013). Applying big data and linked data concepts in supply chains management. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1215–1221. IEEE.

[51] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE.

[52] Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, 34(7):491–541.

[53] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47.

[54] Tanenbaum, A. and Van Steen, M. (2007). *Distributed systems*. Pearson Prentice Hall.

[55] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

[56] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.

[57] Wanderman-Milne, S. and Li, N. (2014). Runtime code generation in cloudera impala. *IEEE Data Eng. Bull.*, 37(1):31–37.

[58] Ye, F., Wang, Z.-J., Zhou, F.-C., Wang, Y.-P., and Zhou, Y.-C. (2013). Cloud-based big data mining & analyzing services platform integrating r. In *Advanced Cloud and Big Data (CBD), 2013 International Conference on*, pages 147–151. IEEE.

[59] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer.

[60] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10.

[61] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association.

[62] Zhang, Y., Duc, P. M., Corcho, O., and Calbimonte, J.-P. (2012). Srbench: a streaming rdf/sparql benchmark. In *The Semantic Web–ISWC 2012*, pages 641–657. Springer.

List of Figures

List of Tables