Data-driven Transfer Optimizations for Big Data in the Industrial Internet of Things

vorgelegt von
M. Sc.
Niklas Bernhard Semmler

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 11. Juni 2021

Berlin 2022

Abstract

In the last two decades, the Internet of Things (IoT) has grown from a mere vision to everyday reality. Its fundamental idea is that devices become interconnected with each other and digital services. The consumer side of the IoT, the Consumer Internet of Things (CIoTs), has become omnipresent in the form of wearables, virtual assistants, and smart home solutions. The industrial side of the IoT, the Industrial Internet of Things (IIoT), has received less attention from the general public. The IIoT takes the shape of industrial-grade devices, from trucks to industrial robots, that are equipped with sensors and networking chipsets. It promises to reduce waste, increase machine lifespans, improve energy efficiency, and enable mass customization.

The CIoT predominantly creates big data sparsely across wide areas, e.g., distributed over many households. CIoT applications collect and process this data in the cloud. In contrast, the IIoT predominantly creates big data at industrial facilities that are densely populated with devices. Because these industrial facilities are often connected to the cloud by low-bandwidth access networks, IIoT big data cannot be entirely transferred to the cloud. Simultaneously, industrial facilities are often equipped with limited computing resources. This creates a data-compute asymmetry where most data stays at resource-constrained industrial facilities, and only a fraction is transferred to the resource-rich cloud. Unmitigated, the network bottleneck delays the installation of IIoT applications. This thesis introduces software solutions that reduce the impact of the network bottleneck.

Systems processing IIoT big data face complexity from both the data sources and application requirements. On the one side, the data is generated by inherently hierarchical and distributed industrial processes and retains these qualities. On the other side, IIoT applications have diverse requirements on data access and processing (e.g., requiring database-like access to historic IIoT big data or processing recent IIoT big data as data streams). This work proposes a high-level architecture that connects both sides using novel computing primitives. Our novel computing primitives flexibly aggregate and combine data across hierarchies and locations. As part of our architecture, we introduce data-driven transfer optimizations to reduce the impact of the network bottleneck. The remainder of the thesis presents three case studies that implement data-driven transfer optimizations for different data processing frameworks.

In our first case study, IIoT applications in the cloud access a data store at an industrial facility. They face a trade-off between processing individual queries at the industrial facility and transferring raw data to the cloud. We introduce online replication strategies that make fine-granular choices based on data access patterns. In our second case study, an IIoT application identifies the top-k most relevant objects (e.g., machine failures) across multiple industrial facilities. We introduce a new fixed-phase distributed top-k algorithm. This algorithm uses fewer phases than related work while simultaneously reducing the data transfer volume compared to the state-of-the-art. In our final case study, IIoT applications process data streams using dataflow programs. Dataflow programs process data by moving it through an operator graph. A sudden rise in the data input rate or a software or hardware failure risks to increase the dataflow program's latency and decrease its throughput. We introduce a load shedding solution that mitigates this risk and simultaneously balances the data loss with the loss of previously done work. Our work enables IIoT applications for resource and bandwidth-constrained industrial facilities.

Zusammenfassung

In den letzten zwei Jahrzehnten hat sich das Internet of Things (IoT) von einer bloßen Vision zur alltäglichen Realität entwickelt. Die Grundidee des IoT ist, dass sich Geräte untereinander und mit digitalen Diensten vernetzen. Die Konsumentenseite des IoT, auch bekannt als Consumer Internet of Things (CIoT), ist in der Form von Wearables, virtuellen Assistenten und Smart-Home-Lösungen allgegenwärtig. Die industrielle Seite des IoT, auch bekannt als Industrial Internet of Things (IIoT), hat in der breiten Öffentlichkeit weniger Beachtung gefunden. Inzwischen werden Industriegeräten, von Lastwagen bis hin zu Industrierobotern, mit Sensoren und Netzwerk-Chipsätzen ausgestattet. Die Vernetzung dieser Geräte innerhalb des IIoT verspricht Abfälle zu reduzieren, die Lebensdauer und Energieeffizienz von Maschinen zu erhöhen und Produktionsflexibilität zu ermöglichen.

Das CIoT erzeugt Big Data vorwiegend spärlich über weite Bereiche, z.B. über viele Haushalte verteilt. Diese Daten werden zumeist in der Cloud gesammelt und dort verarbeitet. Im Gegensatz dazu erzeugt das IIoT Big Data überwiegend in Industrieanlagen mithilfe einer großen Zahl von vernetzten Geräten. Da diese Industrieanlagen oft über Zugangsnetze mit geringer Bandbreite mit der Cloud verbunden sind, kann IIoT-Big-Data nicht vollständig in die Cloud übertragen werden. Gleichzeitig sind die Industrieanlagen häufig mit begrenzten Rechenressourcen ausgestattet. Dadurch entsteht eine Daten-Verarbeitungskapazität-Asymmetrie, bei der die meisten Daten in den ressourcenbeschränkten Industrieanlagen verbleiben und nur ein Bruchteil in die ressourcenreiche Cloud übertragen wird. Ungemildert verzögert der Netzwerkengpass die Installation von IIoT-Anwendungen. Diese Arbeit stellt Softwarelösungen vor, die die Auswirkungen des Netzwerkengpasses reduzieren.

Systeme, die IIoT-Big-Data verarbeiten, sind an zwei Fronten mit Komplexität konfrontiert. Auf der einen Seite werden die Daten durch inhärent hierarchische und verteilte industrielle Prozesse erzeugt und behalten diese Eigenschaften bei. Auf der anderen Seite haben IIoT-Anwendungen unterschiedliche Anforderungen an den Datenzugriff und die Datenverarbeitung (z.B. die Behandlung von IIoT-Big-Data als Datenbank oder als Datenstrom). In dieser Arbeit wird eine High-Level-Architektur vorgeschlagen, die beide Seiten mithilfe neuartigen Computing Primitives verbindet. Diese neuartigen Computing Primitives aggregieren und kombinieren Daten flexibel über Hierarchien und Standorte hinweg. Basierend auf dieser Architektur führen wir datengesteuerte Übertragungsoptimierungen ein, um die Anzahl und das Volumen des Datenaustauschs zwischen Industrieanlagen und der Cloud zu begrenzen. Im weiteren Verlauf der Arbeit stellen wir drei Fallstudien vor, die datengesteuerte Übertragungsoptimierungen für verschiedene Datenverarbeitungs-Frameworks implementieren.

In unserer ersten Fallstudie greifen IIoT-Anwendungen in der Cloud auf einen Datenspeicher in einer Industrieanlage zu. Sie stehen vor der Wahl entweder individuelle Abfragen in der Industrieanlage zu verarbeiten oder die Rohdaten in die Cloud zu übertragen um sie dort zu verarbeiten. Wir stellen Online-Replikationsstrategien vor, die auf der Grundlage von Datenzugriffsmustern fein-granulare Entscheidungen treffen. In unserer zweiten Fallstudie identifiziert eine IIoT-Anwendung die top-k-relevantesten Objekte (z.B. Maschinenausfälle) über mehrere Industrieanlagen hinweg. Wir stellen einen neuen verteilten Top-k-Algorithmus mit einer festen Anzahl von Phasen vor. Dieser Algorithmus reduziert die Anzahl der Phasen und das Datenübertragungsvolumen im Vergleich zum gegenwärtigen Stand der Technik. In unserer letzten Fallstudie verarbeiten IIoT-Anwendungen Datenströme mithilfe von Datenflussprogrammen. Diese Programme Daten, indem sie sie durch einen Graphen von verbundenen Operatoren bewegen. Ein plötzlicher Anstieg der Dateneingangsrate oder ein Software- oder Hardwarefehler kann die Latenzzeit der Programme erhöhen und ihren Durchsatz verringern. Wir stellen eine Load Shedding-Lösung vor, die dieses Risiko abmildert. Zusätzlich balanziert unsere Lösung gleichzeitig den Verlust von Daten mit dem Verlust von zuvor geleisteter Arbeit. Diese Dissertation ermöglicht datenhungrige IIoT-Anwendungen für ressourcen- und bandbreitenbeschränkte Industrieanlagen.

Acknowledgements

This thesis would not have been possible without my supervisors, colleagues, friends, and family's continuous support and motivation. I use this opportunity to thank those who helped me along this path and apologize in advance should I have omitted anyone by name.

I started my time at the research group INET as student worker for Ingmar Poese. He gave me the opportunity to attend my first conference at SIGCOMM 2012 in Helsinki. Without his example to combine research and industry, I would not have pursued a Ph.D.

From the beginning, my supervisor Anja Feldmann has shown me that a good understanding of the data is the beginning, middle, and end of any research project. Her ability to immediately grasp a solution's weakness from a simple plot has been truly inspiring. In later years, my de-facto co-supervisor Georgios Smaragdakis proved to me multiple times that the determination and the many hours of work are deciding factors of getting projects done. Watching his work ethic has been incredibly motivating.

Since the start of my Ph.D., Philipp Richter and Matthias Rost have been permanent fixtures that motivated and pressured me to move forward. Without them and their genuine support, I would not have completed this work. On the other side of the spectrum, my office mate Damien Foucard was always there to listen and exchange crazy ideas without premature judgment. I admire his unwavering calmness and determination to follow his own path.

When I moved from the university to continue my research at SAP, I was welcomed by a tight-knit group of colleagues that accepted me as one of their own. The group included Thomas Bodner, Daniel Johannsen, David Knacker, Anton Niadzelka, Christian Krause, Hannes Rauhe, Hendrik Radke and Josie Rueckert. I am particularly grateful to Hannes for patiently listening to so many iterations of the same work week after week.

After a large part of the INET research group moved to Saarbrücken, Thomas Zinner took over the research group. Even when I was technically not part of INET anymore, he made sure that I always felt welcome. With Thomas, the Ph.D. student Susanna Schwarzmann arrived at INET. In the last two years, she has become my closest comrade. Hang in there!

Over the years, I had the opportunity to work with many student workers. They contributed to this work and many other projects that, unfortunately, never made it beyond the prototype stage. Thank you Robert Krösche, David Herzog, Paula Breitbach, Kajetan Maliszewski, Jan Heyd and David Guzmán. I extend my special thanks to Hendrik von Kiedrowski, who somehow always had the time to help me out.

To my colleagues at INET and particularly Enric Pujol and Jawad Saidi I am thankful for sharing this path with me. I also thank Franziska Lichtblau, Florian Streibelt, and Lars Prehn for making me feel welcome at the MPI in Saarbrücken. I extend my gratitude to the system administrators, particularly Sarah Dierenfeld and Christian Struck, for getting my hardware unstuck every so often. And to INET's secretary Birgit Hohmeier-Toure, for helping me through so many administrative hurdles from my first day as INET's student to just very recently.

Finally, I thank my family for putting up with me over this strenuous period and in general. My wife, Albina, for her patience and encouragement. My parents and my brother Jakob for always being there for me. Your resourcefulness in your work and willingness to go the extra mile for family and friends is something I strive towards.

This work would not have been possible without my employment by SAP SE. It was supported by the Software Campus project "DNA: Distribute and Aggregate" (FKZ 01|S12056) and in part by funding from the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158, FKZ 679158).

Publications

Parts of this thesis are based on the following peer-reviewed papers that have already been published. All my collaborators are among my co-authors.

International Conferences

Distributed Mega-Datasets: The Need for Novel Computing Primitives.
NIKLAS SEMMLER, GEORGIOS SMARAGDAKIS, ANJA FELDMANN.
IEEE 39th International Conference on Distributed Computing Systems, 2019.

Online replication strategies for distributed data stores.
NIKLAS SEMMLER, GEORGIOS SMARAGDAKIS, ANJA FELDMANN.
Open Journal of Internet Of Things 5 (1), 47-57, 2019.

Edge replication strategies for wide-area distributed processing.
NIKLAS SEMMLER, MATTHIAS ROST, GEORGIOS SMARAGDAKIS, ANJA FELDMANN.
Proceedings of the Third ACM International Workshop on Edge Systems, 2020.

Contents

1 Introduction

The *Internet of Things (IoT)* is changing not only the consumer but also the industrial environment. The *Industrial Internet of Things (IIoT)* promises to reduce waste, increase machine lifespans and energy efficiency, and enable mass customization. Unlike the *Consumer Internet of Things (CIoT)* that deploys smart devices (e.g., wearables) sparsely over a wide area (e.g., many households), the IIoT deploys numerous smart devices at single industrial facilities. This difference leads to a separate set of challenges. Many industrial facilities house computer clusters with limited processing power and are located at the edge [9] of the Internet, i.e., they connect to the Internet via low-bandwidth access networks. Both factors limit the extent to which data can be locally processed or transferred to the cloud. Simultaneously, the increase in the number of smart devices per industrial facility and data rates per smart device lead to an explosion of data volumes generated by industrial facilities. Over time access networks become the dominating bottleneck (see Figure 1.1), limiting the growth of IIoT applications.

Cloud-native data processing and machine learning frameworks are key enablers of CIoT applications. They join and process data from a massive number of smart devices. Further, they enable the creation of digital twins [96] (rich virtual representations) of smart devices and open new forms of optimizations. The IIoT's network bottleneck and the limited compute resources available at industrial facilities constrain similar achievements for IIoT applications. New data aggregation mechanisms are needed to overcome these obstacles. However, IIoT big data differs from big data produced by web and CIoT services. Datasets at industrial facilities are the product of and stored in hierarchies of industrial processes. Hence, unlike other large datasets (e.g., click logs), these datasets are inherently hierarchical. Furthermore, as the same smart devices and similar industrial processes generate datasets across industrial facilities, they are semantically related. Processing these datasets together can yield significant quality improvements, e.g., for machine learning methods. Data aggregation methods have to account for these properties of IIoT big data.

Data aggregation methods need to come in two forms. First, we need mechanisms that enable the processing of data across the hierarchies of industrial processes. These mechanisms need to be able to flexibly aggregate data in

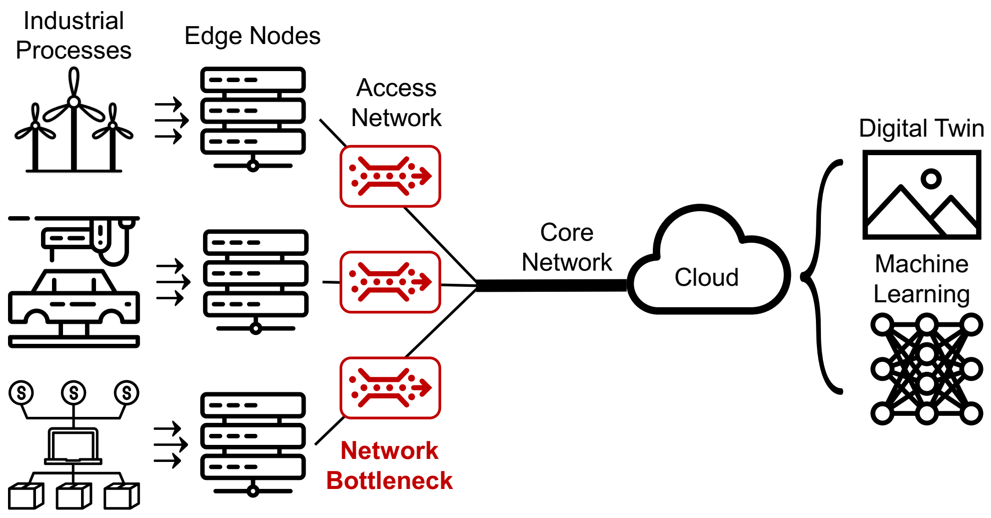


Figure 1.1: IIoT network bottleneck. Across industries, a growing number of smart devices generate massive datasets at industrial facilities. Cloud-native data processing and machine learning frameworks can process this data to produce rich digital twins (virtual representations). Low-bandwidth data access networks connecting edge nodes to the cloud put limits on the data exchange and hinder the deployment of IIoT applications.

response to storage availability and user interest. They also need to enable the exchange and merge of aggregates across levels of the hierarchy (e.g., from individual devices to the local server cluster). Second, we need mechanisms that reduce the transfer volume exchanged between edge nodes (server clusters at industrial facilities) and the cloud. These mechanisms need to exploit opportunities to reduce the transferred data volume beyond the existing use of caching and compression. We focus, for the most part, on the second set of mechanisms.

1.1 Setting

As IIoT applications use a diverse set of data processing frameworks, a single one-size-fits-all solution will not be able to cover all IIoT applications. This work focuses on three types of data processing frameworks that we distinguish by their data transfer patterns: query- and replication-based exchanges, synchronous message passing, and dataflow programs. Query-based data exchanges are used by classical database systems and modern micro-service architectures. They exchange data in response to queries of users or other applications. Additionally, these systems often allow for moving data via replication between multiple servers of the same system for fault tolerance or to exploit data locality. Synchronous message passing is used by systems that follow the Bulk Synchronous Processing (BSP) model [155]. These systems partition data over a set of servers and exchange data among servers in discrete phases. In dataflow programs, a chain of static operators transforms data as it moves from one operator to the next. Contemporary dataflow systems, e.g., Apache Flink [121], distribute and parallelize operators in the form of a (mostly) directed graph of inter-connected processes deployed over multiple servers.

We call the mechanisms that reduce the transferred data volume: data transfer optimizations. Transfer optimizations can be classified into two groups *Avoidance* and *Compression*. Data transfer can be avoided or at least greatly reduced by moving the computation to the data or creating copies of the data (raw data or results) closer to the user. Where the avoidance of transfers is not possible, the data transfer volume is reduced by removing redundancies (lossless compression) or decreasing the precision (lossy compression). Many existing data transfer optimizations (e.g., caching) use data-agnostic worst-case optimized algorithms. They limit the performance drop under adversarial conditions and use very little processing power. However, simultaneously they do not adapt to their environment and thus do not necessarily provide good average-case performance. Instead, this work focuses on data transfer optimizations that exploit patterns in the data distribution, data access records, and resource utilization.

1.2 Research Questions

The Industrial Internet of Things (IIoT) generates rapidly increasing data volumes at industrial facilities. IIoT applications can use this data to optimize industrial processes. Unfortunately, local data processing requires more computing power than industrial facilities have available. Simultaneously, cloud-based data processing is limited by bandwidth-constrained access networks. This double bind slows down the development and deployment of IIoT applications to the pace of network infrastructure upgrades. This leads to the central question of this thesis.

"How can software solutions enable timely processing of IIoT big data?"

To support IIoT applications, we need to reduce the number and volume of data exchanges between edge and cloud. This requires the systematic use of data-driven transfer optimizations that exploit patterns in the data, data access, and resource utilization. Such an undertaking is challenged by the diverse set of applications and the complexity of the data. Similar to contemporary data-driven applications, future IIoT applications are likely to draw from a diverse set of frameworks and approaches (e.g., data stream and data batch processing) to process and store data. Simultaneously, data in IIoT contexts is created at different hierarchical levels (e.g., machine vs floor level) and inherently distributed (i.e., generated over multiple machines and sensors). These challenges call for an architecture that can aggregate data from various sources at different levels and serve data to diverse applications. The design of this architecture poses the first sub-question of this thesis:

Q1: *"What architecture can provide data transfer optimizations for inherently hierarchical and distributed datasets?"*

To answer question **Q1**, we propose an architecture that is broad enough to encompass a variety of aggregation and data processing mechanisms. Naturally, this high-level architecture does not present concrete implementations of aggregation mechanisms. We complement our investigation with a second sub-question that reflects a bottom-up perspective.

Q2: *"What mechanisms can reduce data exchanges beyond worst-case optimized mechanisms?"*

To investigate question **Q2** we investigate the application of data aggregation to three prominent data processing frameworks in the form of three case studies. Each case study proposes a mechanism that exploits patterns in the data, data access, or resource utilization. We compare the mechanisms to frequently used static data aggregation mechanisms.

1.3 Contributions

This thesis makes two contributions. We introduce a general architecture for IIoT big data and present three case studies covering prominent data processing frameworks 1.1. First, we propose a general architecture for IIoT big data:

- The emerging IIoT big data has structural differences to big data. Industrial processes that create IIoT big data are usually part of a production hierarchy. Each level produces data and needs to process data to feed local control loops (e.g., failure detection). This necessitates a hierarchical data processing architecture. Simultaneously, raw sensor data is often so massive that it cannot be transferred as a whole. Hence, data needs to be summarized in response to the demands of various applications. We introduce an architecture based on hierarchically distributed data stores that summarize data and exchange data summaries. The data summaries are created and adjusted by novel compute primitives. Data-driven transfer optimizations reduce the data transfer volume exchanged between industrial facilities and the cloud. With this architecture, we translate the problem posed by IIoT big data into smaller surmountable sub-problems.

Second, widely different IIoT use cases use different application frameworks. We present case studies for three frameworks: (a) client-server queries and replication (b) multi-phase synchronous message exchanges (c) dataflow systems.

a) Online replication strategies

  - For database-like systems, the aggregate volume of transferred query results determines the data transfer volume. Future queries can be processed locally by replicating data to the query origin, removing the need to transfer their results. However, the replication of data introduces its own data transfer volume. We propose online replication strategies that manage the trade-off between transferring query results and replicating raw data in the face of an uncertain future. We investigate the application of online replication strategies to two scenarios. In the first scenario, a single set of data partition is accessed. Strategies can use the time series of past accesses to decide whether or not to replicate. Our strategies are able to reduce the data transfer volume by between 18% to 51%. In the second scenario, data partitions are accessible for limited time windows and are afterward exchanged with new data partitions. This allows strategies to learn from access behavior in previous time windows. Here our strategies reduce the data transfer volume by 22% on average.

b) Distributed top-k for power-law distributions

  - Distributed top-k is a mechanism to create global top lists for distributed data among multiple servers. We use knowledge of the data distribution and the users' error constraint to reduce data transfer volume and the number of data exchanges. In this work, we exploit two insights, (1) most top-k operations are executed over power-law distributions, and (2) top-k is an approximate tool that allows for a slight error. Our solution saves 13% in data transfer volume and 1 round-trip or 5% in data transfer volume and 2 round-trips for a real-world dataset compared to related work.

c) Load shedding for highly parallel stream processing systems

  - Stream processing systems are used when IIoT applications need to process data in or near to real-time. Many of the contemporary stream processing systems are based on the dataflow model. These systems exhibit a performance problem when an operator processes data at a slower pace than it receives it. The effect is a slow-down of earlier operators in the operator chain and is caused by queue saturation. The effect is called backpressure. Unmitigated, it increases the latency and can further decrease throughput. The common solution to backpressure is to scale the problematic operator. This option is often not available for resource-constrained industrial facilities. Instead, we use load-shedding to reduce end-to-end latency by more than a magnitude and avoid further degradation of the throughput in some cases by 23%. Our solution balances both data loss and wasted work.

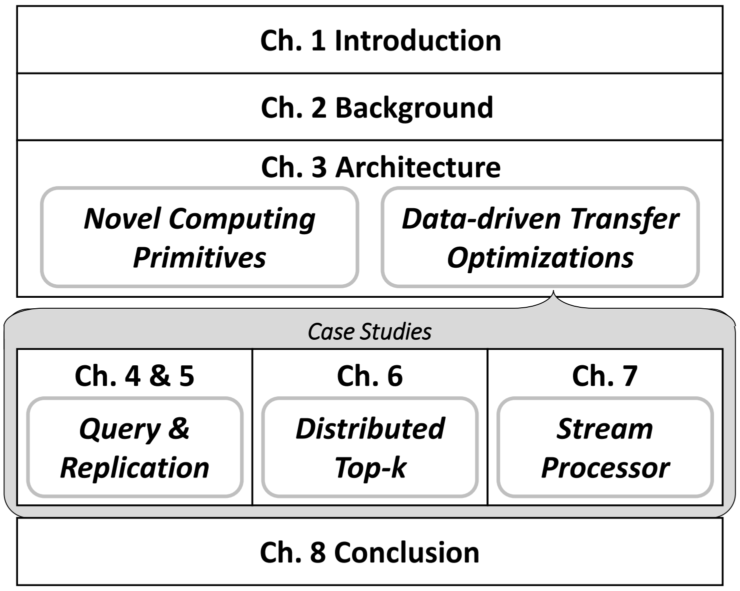


Figure 1.2: Thesis Overview

1.4 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 discusses concepts that form the foundation of the thesis. Chapter 3 presents an architecture to process IIoT big data in a timely and cost-efficient manner. As part of this work, we discuss the challenges that IIoT big data create and how the architecture covers them. Chapter 4 introduces online replication strategies. These strategies use existing user access behavior to decide whether replicating a partition can reduce the data transfer volume for a given time window. Chapter 5 adds online strategies that learn over past time windows. Chapter 6 introduces a multi-phase algorithm for the commonly used top-k operation on distributed datasets. Chapter 7 applies the principles of our approach to data streams for a dataflow system. Chapter 8 summarizes our contributions, points out possible future work, and concludes the thesis.

2 Background

This chapter describes the background of this work. We discuss the term IIoT, how it differs from existing terms, what technologies play a major role, and give examples of IIoT use cases. Then, we describe how growth measures indicate that the network bottleneck will become more severe in the future. Finally, we give a historical account of data storage and processing technologies.

2.1 The Industrial Internet of Things

The Industrial Internet of Things (IIoT) is a concept of fully connected industrial processes that enable new forms of automation. Central themes of the IIoT include the combination of data from globally distributed machines, machine learning to identify and learn behaviors that are too subtle or complex for human operators, and the use of machine-to-machine communication to facilitate rapid and autonomous interactions. These changes will enable greater efficiency, more automation, faster failure detection and correction, and faster reconfiguration of industrial processes to meet the changing market demand. In the following, we discuss other terms that refer to the developments in the industrial sector. Next, we introduce technologies that have been integral in enabling the IIoT. Finally, we describe concrete use cases that show the potential and challenges of the IIoT.

2.1.1 Disambiguation

A number of terms are used to refer to the current development in the industry. Here, we explain the differences between the terms and justify why we have chosen the term IIoT over existing alternatives. Many of the terms attempt to position themselves in comparison to industrial history. From a historical standpoint, the first and second industrial revolutions are clearly defined by the concurrent emergence of a number of key technologies. The first industrial revolution was triggered by the development of steam power. The first factories and machine tools were build around steam engines to harness their power. The second industrial revolution was triggered by the utilization of electricity. The ease by which electricity can be transported allowed new factories to be built around the production work-flows. The development of the assembly line is possibly the most famous example of this era. In comparison to these two revolutions, the term third and fourth revolutions are more contended.

In his book *The Third Industrial Revolution* [130], Jeremy Rifkin claims that the third revolution is related to globalization and a focus on renewable energies. In contrast, the executive chairman of the World Economic Forum Klaus Schwab [138] sees the third industrial revolution in the use of electronics and information technology. He claims that we are entering the *Fourth Industrial Revolution*, a digital revolution that is “blurring the lines between the physical, digital, and biological spheres”. The term is related to the high-tech strategy coined *Industrie 4.0* of the German government [32], first introduced in 2011 [156]. The term is used to refer to a vision that encompasses automation of the full value chain of mostly manufacturing-oriented businesses. McKinsey uses the similar term *Industry 4.0*[22]. However, the number 4.0 in their context does not refer to previous industrial revolutions but rather to three recent developments: (1) Lean manufacturing, (2) labor outsourcing to low-wage countries, and (3) the recent introduction of manufacturing automation. Their term *Industry 4.0* refers to “the next phase in the digitization of the manufacturing sector”.

From another point of view, the changes in the industrial sector can be perceived as the influx of technologies that have matured as part of Internet services and the Internet of Things. The term *Internet of Things* was coined by Kevin Ashton [10] and refers to things that are connected to the Internet and often entails the management of these things by applications or the interaction of things without the involvement of humans. General Electric went in this direction and coined the term *Industrial Internet* [99] to refer to the merge of the Internet with industrial processes. Later, the term Industrial Internet of Things [70] was coined to refer more specifically to the merge of the Internet of Things with industrial processes (as opposed to the Consumer Internet of Things). Compared to the various attempts to define a new industrial revolution, which have originated in economic and political backgrounds, the *IIoT* focuses on a narrower technical perspective. For this reason, we are using the term IIoT in this work.

2.1.2 Industrial Automation

The IIoT has its roots in several inventions and concepts that were developed over the last sixty years. Figure 2.1 gives a rudimentary timeline. For the manufacturing industry, three technologies stand out that drastically increased automation. The first is the programmable logic controller (PLC), which the US automotive industry introduced it in 1969. It simplified the control and automation of industrial processes. In the same year, the “Stanford arm” [135], a 6-dimensional robotic arm, revolutionized industrial robotics. Its increased flexibility allowed the application to a far greater set of use cases than its predecessors. A third notable invention is the introduction of 3D printing. The first patent for the printing of both metals and plastics was created in 1984.

For the transportation industry, the introduction of RFID chips and GPS systems changed how goods and vehicles were tracked all over the globe. Mario Cardullo created a predecessor of the current chips [35], which form the backbone of many low-energy tracking systems in the logistics and manufacturing industries. For GPS (Global Positioning Satellites), which forms the backbone of navigation systems worldwide, it was the US Department of Defense [110] that introduced it in 1993. Not only did it have a large impact on tracking the movement of goods, but it also changed automation in the agricultural sector.

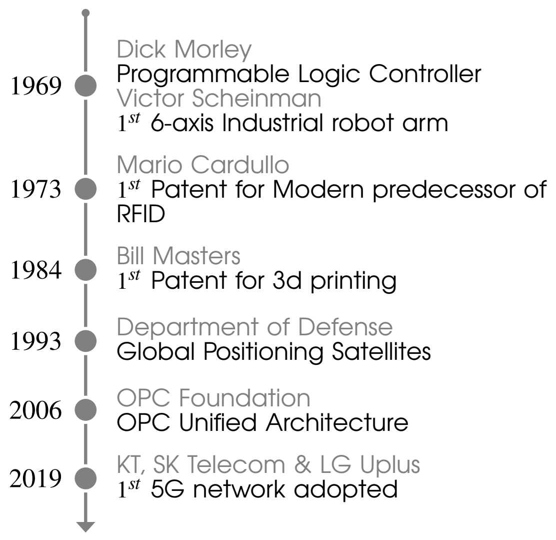

1969
Dick Morley
Programmable Logic Controller
Victor Scheinman
1st 6-axis Industrial robot arm
1973
Mario Cardullo
1st Patent for Modern predecessor of RFID
1984
Bill Masters
1st Patent for 3d printing
1993
Department of Defense
Global Positioning Satellites
2006
OPC Foundation
OPC Unified Architecture
2019
KT, SK Telecom & LG Uplus
1st 5G network adopted

Figure 2.1: History of the industrial internet of things.

A breakthrough in integrating machines from different vendors was established in 2008 when the OPC Foundation standardized the OPC unified architecture for machine-to-machine communications and industrial automation [100]. It differed from previously existing architectures by its extensibility and its interaction with the cloud.

For mobile devices such as autonomous vehicles, the introduction of the new communication standard $5G$ is bound to play a significant role. $5G$ offers greater bandwidth, lower latency, and increased computing power at cell towers. Individual $5G$ towers may become another edge of the internet for mobile applications. In April 2019, three carriers in South Korea [114] adopted the first $5G$ network. Verizon followed an hour later with a $5G$ network in the US.

In this thesis, we mainly focus on the analytical processing power that the IIoT requires. However, there would be no need if flexible production systems such as industrial robots, 3D printers, and autonomous cars were not available. Satellite networks for navigation and the emerging fifth-generation mobile networks form the more extensive infrastructure for these systems. With the OPC Unified Architecture in place, existing machinery has been designed for at least a limited form of connectivity.

2.1.3 IIoT Use Cases

In this subsection, we look at the current industry's state and its challenges in transitioning to the IIoT. We discuss the changes that IIoT brings to three industrial sectors: Energy, Manufacturing, and Transportation.

2.1.3.1 Energy

Renewable energy facilities such as solar and wind parks usually require the flexible management of equipment over a wide area. This process can benefit from automation. Particularly offshore wind parks need to quickly react to changes in the climate and the emergence of microclimates to harvest energy and protect the equipment. At the same time, human crews often deploy with significant delays. Offshore wind parks consist of tens to hundreds of wind wheels and secondary systems. They produce a wide range of data from turbines, electric power transformers, multiple cameras, authorization modules, and many more devices, often at a time scale of milliseconds. This data is often so large in volume that it cannot be in its entirety transferred to the shore but is temporarily stored in local micro data centers and then aggregated. However, essential functionality, such as the timely shut down of wind wheels, can benefit from extensive data processing, and so can other aspects such as the development of next-generation wind wheels.

2.1.3.2 Manufacturing

The IIoT promises faster adjustments to the market, higher production rates, and lower energy use to the manufacturing industries. 3D printers and flexible industrial robots realize faster adjustment to market demand. Both allow recalibration to new production routines in a short time. Cutting humans out of the loop and coupling machines tighter while at the same time allowing for more flexible behavior, e.g., reaction to faults realizes higher production rates. Lower energy use is realized by flexibly scaling production up and down. These seemingly conflicting developments rely on the increase of sensors, e.g., individual machines containing hundreds of sensors and the increased computing power of industrial machines. Both require an adaptation of the factory IT infrastructure to process data. A central aspect is analyzing factory workflows from sensory data to identify faults and opportunities for optimization.

2.1.3.3 Transportation

For the transportation industry, the IIoT promises increased automation of the movement of airplanes and cars. This is especially interesting for airlines that manage large fleets (e.g., Lufthansa with 700 or American Airlines with 1,300 planes). Apart from predictive maintenance, the IIoT offers the prediction of delays, increased fuel efficiency, and the improvement of the next generation of airplanes. For this purpose, new generations of airplanes have up to 24.000 sensors, where older models include only a few thousand or even a few hundred sensors. The data is collected aboard the plane and then moved to storage across multiple paths. Some part of the data is stored on the famous black boxes in case of emergencies. Other urgent data is delivered by mobile connection directly to the airline. The airline collects the full dataset by manually retrieving storage units at regular (weekly) intervals. For a few years, cloud platforms exist [125] that allow sharing the data between multiple stakeholders (e.g., vendors, airlines, regulators). In the future, data retrieval will likely be automated, with edge nodes at airports serving as (temporary) data stores.

Another application of the IIoT is car-sharing. Car-sharing fleets have grown substantially in metropolitan areas around the world in recent years. Car-sharing offers cities a reduction of the number of cars and thereby the necessity to provide parking space. Over the long-term, car-sharing solutions may transform into providers of autonomous vehicles, which do not require the active participation of a human driver. Connectivity can benefit the synchronization of vehicles and the deployment of new updates on the fly. The continued adaptation of $5G$ is an essential step towards this goal. Also, when autonomous vehicles take over taxi- and delivery-services, they will require orchestration that can only be realized with continuous connectivity.

2.2 Origin of IIoT's Network Bottleneck

We base our work on the hypothesis that not all data generated at the (industrial) edge can be transferred to the cloud. We base this hypothesis on the assumption that data production rates grow faster than bandwidth for edge nodes. In the following, we break the reasoning behind this assumption down.

1 Data production rates by the IIoT grow rapidly.

- 1.1 Individual IIoT devices produce increasingly large data volumes.
  - 1.1.1 Sensors are becoming cheaper.
  - 1.1.2 More sensors are sold.
  - 1.1.3 Computing hardware is becoming smaller, allowing for more powerful sensors and machines.
- 1.2 More IIoT devices are deployed.
  - 1.2.1 More IIoT devices are sold.
  - 1.2.2 More IIoT devices are connected.

2 The capacity of access networks grows slower than the capacity of core networks.

- 2.1 The increasing data transfer volume is putting more pressure on access networks.
- 2.2 Increasing the capacity of access networks is more costly than the capacity of core networks.

3 IIoT applications (e.g., predictive maintenance) require increasing data volumes.

- 3.1 Future IIoT applications will depend on machine learning to identify patterns.
- 3.2 Machine learning starts to perform well when large data volumes become available.

We have summarized supporting evidence for the argument's points in Table 2.1 (annual growth rates) and Table 2.2 (laws observed on annual growth). Unfortunately, the data is often ambiguous. For example, while some evidence points to the faster growth of core networks compared to access networks (35 % [152] to 20 % [144]), other evidence points to the growth of access networks at around 50 % [44, 115]. Additionally, the overall data seem to grow slower than the network (26 % to 40 % [147, 146]), but the critical question is how this growth is spread over edge nodes and the cloud. For Item 2.2, it is also challenging to identify reliable information on the cost structure of upgrading networks. While it makes intuitive sense that upgrading access networks in the vicinity of cities takes more effort than upgrading backbone cables that are placed in the vicinity of highways or less crowded places, the numbers to support this intuition are not readily available. That both core and access networks include a wide variety of network types further complicates matters. Similar complications arise in the classification of IIoT machine learning applications. While there is anecdotal evidence of their success, hard evidence is harder to come by, even more so as they are often prone to companies' confidentiality agreements.

To conclude, we have summarized both the argument and the evidence supporting the argument that forms our work's motivation. The evidence does not indicate that all IIoT use cases will face the problem but indicates that many will. Where the problem occurs, our contributions in this work may be of help.

Table 2.1: Yearly growth of trends influencing the growth of the IIoT.

Trend	Annual Growth	Point
IoT devices		
Image sensor sales	5 % [142]	1.1.2
Sensor Prices	-7 % [105], -8 % [145], -12 % [54]	1.1.1
Industrial robot sales	15 % [88]	1.2.1
3D Printer sales	25 % [7]	1.2.1
Number of connected devices	24 % [54]	1.2.2
IoT connections	19 % [105]	1.2.2
Data		
Overall Data	26 % [147], 40 % [146]	2.1
Enterprise Data	84 % [54]	2.1
Mobile Data	41 % [75]	2.1
Bandwidth		
International bandwidth	35 % [152]	2
Access bandwidth	20 % [144]	2

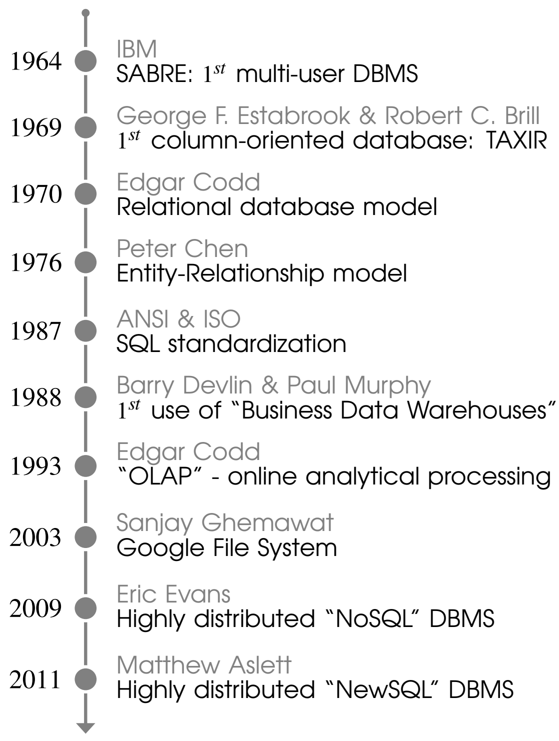

1964
IBM
SABRE: 1st multi-user DBMS
1969
George F. Estabrook & Robert C. Brill
1st column-oriented database: TAXIR
1970
Edgar Codd
Relational database model
1976
Peter Chen
Entity-Relationship model
1987
ANSI & ISO
SQL standardization
1988
Barry Devlin & Paul Murphy
1st use of "Business Data Warehouses"
1993
Edgar Codd
"OLAP" - online analytical processing
2003
Sanjay Ghemawat
Google File System
2009
Eric Evans
Highly distributed "NoSQL" DBMS
2011
Matthew Aslett
Highly distributed "NewSQL" DBMS

Figure 2.2: History of data warehouses.

2.3 Evolution of Data Storage and Processing Systems

To better understand the software for storing and processing Big Data, we describe a few of the significant milestones over time in the next two subsections. Apart from very early historical examples, we focus entirely on shared-nothing architectures, i.e., architectures where all communications between two computers have to pass over a computer network. We first focus on the development of data storage methods for large analytical datasets. Next, we discuss models for programming a distributed system. We do not claim completeness, but we believe discussed technologies give a good understanding of the historical development.

2.3.1 Distributed Data Storage

Figure 2.2 depicts the timeline of storage technologies. We chose to start from the point of the first networked commercial database: SABRE [140]. Booking agents of American Airlines used SABRE across the US to manage airline bookings. This multi-user aspect is essential for today's infrastructure, where a diverse set of applications access the same data. A distinguishing feature of databases meant for analytical processing in contrast to databases

Table 2.2: Laws behind the IIoT.

Name	Technology	Annual growth	Point
Moore's law [106]	Number of transistors per circuit	41 %	1.1.3
Edholm's law [44]	Bandwidth and data rates	59 %	2
Nielsen's law [115]	(high-end) user bandwidth	50 %	2

used for transactions, e.g., bookings, is their column-oriented storage layout. This layout stores numbers of a column (e.g., booking prices) in sequential order, which speeds up the processing performance. TAXIR [57] was the first column-oriented database that had this feature.

Prior to 1970, programs and users who wanted to access data had to understand the data's physical layout. This approach changed in 1970 when Edgar Codd introduced his relational model [46]. Peter Chen's introduced the Entity-Relationship model, which simplified and unified the structuring of data, in 1976 [43]. Both presented a semantic abstraction from data that led to the development of the querying language SQL [87], which was standardized by the year 1987.

The developments in the 1970s spurred the use of databases in different areas of business administration. As companies incorporated multiple databases, they found themselves in a situation where critical information was distributed among different systems. (Business) Data warehouses [55] were soon developed to collect data across multiple databases and process it together.

Edgar Codd helped define analytical processing by introducing OLAP (online analytical processing) and 12 rules for evaluating OLAP products [47]. The concept of slicing multi-dimensional data is at the heart of OLAP. A few years later, the term "Big Data" emerged to denote datasets that are too large to be processed by a single machine.

In the 2000s, the web took off, and it became apparent that new databases were needed to store website contents and click logs. Initially, this led to large volumes of unstructured data, distributed file systems were developed to store this data [71]. Later, the persistent need for structured access and change of this data help to form the NoSQL [59] movement. This movement focused on the development of databases that relied on simplified models and relaxed existing consistency models to increase scalability. While the early NoSQL databases were a success, it became quickly apparent that the limitations in consistency and query language slowed application development. NewSQL [11] tried to bridge the NoSQL databases' performance with the consistency guarantees of earlier relational databases.

Today we benefit from these developments in that we can now use highly scaleable distributed databases with varying semantic models for the analytical processing of sensor data.

2.3.2 Parallel Programming Models

While single-core computers drastically improved in speed in the 1980s and 1990s, this process slowed down in the first decade of the 2000s. Since then, systems have increasingly relied on parallel computing to process larger and larger data volumes. Unfortunately, the development of parallel programs turned out to be more complex than their counterparts. A number of frameworks and programming models were developed to simplify the process. Figure 2.3 depicts a timeline of these developments.

The groundwork of concurrent processing was laid in the 1960s. Dijkstra introduced the first paper on the topic in 1965, where he introduced the concept of mutual exclusives. In 1966, Flynn introduced a taxonomy classifying different types of parallel computing [67]. One year later, Amdahl discussed the potential speed-up that can be achieved by using parallel computing for a previous sequential problem [6]. The law governing the speedup became later known as Amdahl's argument and Amdahl's law [132].

Several attempts at defining models for the implementation of concurrent programs were made over the years. One of the first has been Bert Sutherland's Ph.D. work on dataflow programming [149]. In dataflow programs, data is transformed in consecutive while moving across a chain of operators distributed among a set of machines. A different solution was proposed by Hewitt et al in the form of the Actor model [80] in 1973. In the actor model, autonomous agents are distributed across a set of machines and interact with other actors and users via asynchronous

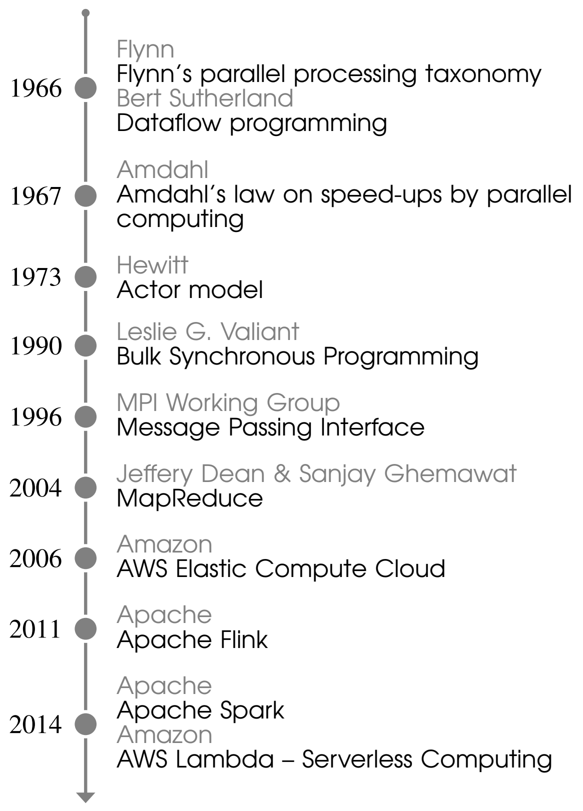

1966
Flynn
Flynn's parallel processing taxonomy
Bert Sutherland
Dataflow programming
1967
Amdahl
Amdahl's law on speed-ups by parallel computing
1973
Hewitt
Actor model
1990
Leslie G. Valiant
Bulk Synchronous Programming
1996
MPI Working Group
Message Passing Interface
2004
Jeffery Dean & Sanjay Ghemawat
MapReduce
2006
Amazon
AWS Elastic Compute Cloud
2011
Apache
Apache Flink
2014
Apache
Apache Spark
Amazon
AWS Lambda – Serverless Computing

Figure 2.3: History of concurrent computing.

messages. Each actor represents an independent part of the full program logic. The Bulk Synchronous Parallel model proposed in 1990 by Valiant [155] forms a third solution. In this formalization, independent processors exchange data in regular supersteps. A different attempt was made in 1966 by the MPI working group [157]. Instead of introducing a single abstraction, they introduced a library of low-level parallel data exchange mechanisms.

In 2006 Amazon introduced the first public cloud. It offered its users the possibility of renting commodity computers for limited time windows [20]. The cloud gave a broad audience access to computer clusters. It also created the need for simple frameworks to orchestrate large numbers of commodity machines for a single analysis. The MapReduce framework by Jeff Dean and Sanjay Ghemawat [52] has been one of the first to fulfill this need. In its open-source implementation as Apache Hadoop [126] it spurred the emergence of a data processing ecosystem including frameworks such as Apache Flink (2011) [121] and Apache Spark (2014) [101]. To take a share of the market for data processing and storing, cloud providers started to provide managed instances of these projects and offer similar proprietary systems. In 2014, Amazon went a step further by offering serverless [77] computing. In this framework, users can upload their functions and only have to pay for their use. Amazon decides both their location of deployment and the degree of parallelism.

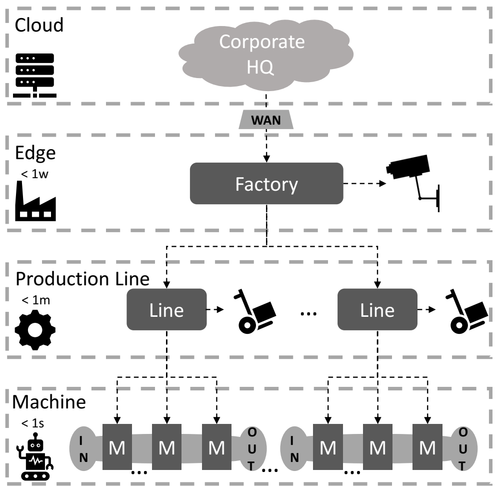
3 Architecture for IIoT Big Data

This chapter describes an architecture for the effective management of IIoT big data. This architecture aims to enable a wide range of simultaneously running IIoT applications in a resource-constrained edge environment. To that end, the architecture ensures that information from physical processes is communicated to their digital representations in a *timely* and *dependable* fashion.

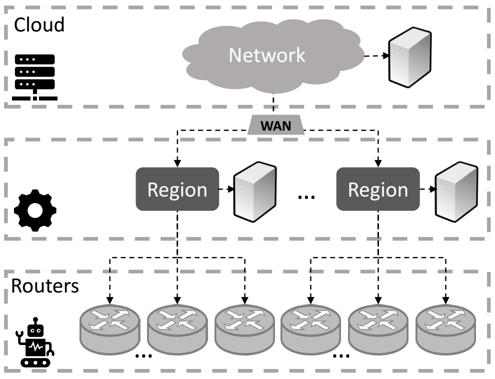
Industrial processes, in general, generate heterogeneous, often non-stationary, and non-uniform, distributed data streams. Their data varies over different scales of time (from sub-millisecond to hours, and days, spanning easily 6-7 orders of magnitude) and space. An architecture has to produce data summaries to cater to multiple applications in parallel. As a starting point for our architecture, we discuss two prominent use cases, a smart factory setting and a network management use case. In both settings, data, physical processes, and their models are highly distributed [102]. Furthermore, both need to provide similar guarantees, including consistency, real-time accuracy, and data privacy, and features, including data sharing and joined processing, across multiple applications. Yet, they are subject to the constraints of the available computing and bandwidth resources as well as security and data privacy restrictions. We identify nine challenges that are common among both use cases.

To process IIoT big data in a coordinated yet distributed fashion in real-time, we propose novel *computing primitives*. The primitives summarize data in a form that supports (a) support arbitrary queries on the data, (b) combining summaries gained from different locations or at moments in time, (c) adjusting the aggregation granularity, (d) adapting to variations in data and queries. Further, they (e) make use of domain knowledge to provide meaningful levels of aggregation. Simultaneously, they use minimal resources and address data lineage, quality criteria, and time constraints. See Saidi et al. [134] for a concrete example of a computing primitive.

Computing primitives ensure that storage space is efficiently used but do not regulate the network usage. For this reason, we propose *data-driven transfer optimization*. They ensure that the data exchange between locations is



(a) Smart Factory.



(b) Network Monitoring.

Figure 3.1: Two settings with IIoT big data: Smart Factory (left) and Network Monitoring (right). The dotted lines represent the lines of (partial) control from higher to lower hierarchy levels.

restricted to avoid congestion. We will expand on the concept of these data-driven transfer optimizations in the remainder of the thesis.

This chapter presents a vision of how IIoT big data can be successfully handled. We introduce an architecture, the concept of computing primitives that ensure efficient use of storage space and transfer optimizations for efficient network use. Throughout the remainder of this chapter, we use two cases, namely, Smart Factory [162, 128] and Network Monitoring [134].

3.1 Use Cases

We discuss two use cases in more detail to pinpoint the central challenges of IIoT big data. Table 3.1 summarizes the challenges.

3.1.1 Use Case: Smart Factory

Traditionally, a factory consisted of single-purpose machines, which followed a rigid sequence of instructions and interacted with each other, e.g., by moving goods over a conveyor belt. As a result of this, monitoring was limited to a small number of sensors with limited capabilities. This factory design limited the machine's operation range and necessitated constant supervision and frequent interventions by human operators.

Drastic changes have happened since then, e.g., the introduction of collaborative robotic arms [122] and autonomous forklifts [49]. Robotic arms extend the range of movements that a machine can perform and simplify the interaction with human operators. Autonomous forklifts extend the reach of factory automation into the warehouse. Both are examples of recent innovations enabled by a combination of high-resolution sensors (e.g., 3D cameras) and a multitude of lower-resolution sensors. Consequently, their data rates have exploded, e.g., a single 3D camera can produce 52 GB/h of uncompressed data, and a high-resolution camera can produce 17.5 GB/h of uncompressed data.

The data rates have increased to the degree that they often require dedicated data processing equipment enabling new, more complex behavior and richer interactions. Still, limited compute and storage resources in the machine's vicinity and limited sharing of data across machines constraint these innovations.

A long-awaited step in the evolution of factories has been the move towards dark factories (or lights-out manufacturing), i.e., factories that use no light as no humans are present. This idea has been around since the 1980s [81] and was realized in the early 2000s [116]. The next step in the evolution will be the smart factory that achieves fully automated processes while also increasing the processes' flexibility to better react to market demand. It will rely on *rapid local decision-making* while respecting today's factory setups (particularly at the machine and production line level), and at the same time, it will be able to *continuously adapt* to insights gained across *factory lines* and *different factory locations*. This adaptation will enable improvements to the efficiency of existing processes, e.g., adjustments to degrading machine mechanics, and enable new processes, e.g., for mass customization.

In Figure 3.1, we illustrate (on the left) the typical hierarchical structure of a factory. Machines connected by a conveyor belt or related technology (the production line) are located at the lowest level. The controller of the production line is one level above. Besides the control of the respective machines, this level may also control supporting processes, such as the movement of materials and products. All production lines are monitored and managed on the factory level, which may leverage additional data (e.g., from factory cameras). Some part of the gathered data/information may be exported into the cloud, where it can be combined with additional resources (e.g., for Enterprise Resource Planning or ERP).

Given the ever-increasing data flood from all sensors, *mega-sized datasets* arise at different points in the factory. As a whole, they form an IIoT dataset. Various applications require this data to be processed in different ways. While some require the data to be processed immediately as data streams, others require it to be stored, either temporarily or almost permanently, to answer (interactive) queries at a later point, e.g., on the history of produced goods for supply-chain management. Similarly, applications have different requirements: they differ in the degree of precision that they require, e.g., in terms of measurements over time; whether they require data from a single IIoT dataset or multiple IIoT datasets; what timeliness they require, e.g., decision making at the machine or factory level may require results between 1 second and 1 minute respectively.

Many applications can be enabled by better use of the factories' data. To name but a few: (a) *predictive maintenance*, the analysis of operational data belonging to a type or class machines of machines to predict failures and schedule maintenance operations accordingly; (b) *supply chain management*, procedures for tracing product failures back to the material used in the production steps or to variations in the production process itself; (c) *process mining*, the review of production processes attained by combining operational data and enterprise data to identify sources for efficiency gains.

3.1.2 Use Case: Network Monitoring

Today's de facto communication medium is the Internet, a network of networks. To cope with the increasing demand and complexity, network operators have to manage their networks. Network management requires an accurate view of the network based on the continuous analysis of network data, i.e., network monitoring.

In the past, a coarse-grained view of the network was sufficient. Today, network operators must have a fine-grained view of their networks. They have to continuously keep track of their network activity both over relatively large time windows, e.g., days or busy hour (6 pm to 12 am), for network provisioning or to make informed peering decisions as well as over smaller time windows, e.g., minutes, to identify and rectify unusual events, e.g., attacks or network disruptions. To that end, they typically rely on either flow-level or packet-level captures from routers within their network. As gathering such data for every packet is often too expensive at high-speed links, packets are sampled, e.g., 1 of every 10K packets [109]. Still, even the resulting datasets can exceed multiple Terabytes per day and router. Thus, sending this data to a central location may or may not be possible, e.g., due to data protection regulations, or desirable, due to bandwidth restrictions. Instead, each dataset produced by a set of routers forms a "local/regional" *IIoT dataset*. These datasets, combined with additional network configuration data, e.g., topology, network element state, and routing configuration, form an *IIoT dataset*.

Many problems that network operators face can be resolved by analyzing an IIoT dataset or a subset of IIoT datasets. It can help to (a) determine network trends, e.g., popular network applications or traffic sources; (b) compute traffic matrices for planning network upgrades, (c) investigate performance or DDoS incidents, i.e., identify affected network parts and possible sources, (d) perform dynamic traffic engineering, by aggregating flow statistics across time and sites, (e) answer interactive queries on the state of the network.

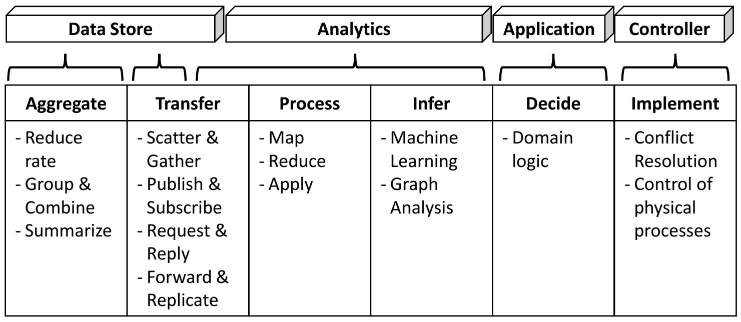
3.2 Architecture

In both use cases, IIoT datasets emerge, and similar challenges have to be addressed. Table 3.1 summarizes the key challenges and associates each of these challenges with the operation of smart factories (middle column) and network monitoring (right column). These challenges motivate us to propose the following novel data processing architecture.

The design of our architecture leverages the following observations: (a) data has to be filtered and aggregated before computation (Challenge 3), (b) data from different data streams, with varying rates and characteristics, have to be combined (Challenge 2), (c) processing has to enable local (Challenge 4) and far-reaching control loops (Challenge 6), (d) processing has to be spread out over a hierarchy of physical processes, resources, and restrictions (Challenges 1, 7), (e) computation has to be modular to include different aggregation, analytics, and application logic (Challenges 8 and 5) and (f) data has to be stored at different aggregation granularity for future, yet unknown, queries (Challenges 9).

Table 3.1: Challenges of IIoT big data and examples from both use cases.

#	Challenge	Smart Factory	Network Monitoring
1	Increasing computation requirements	High-resolution camera feed	High-speed traffic inspection
2	Large number of devices producing data streams	Streams of sensor data	Streams of flow data
3	Massive combined data rates	Machine and factory-level sensors	flow exports from switches, routers, etc.
4	Rapid local decision making	Machine control	Repair of network failures
5	High data variability	Differing sensor types	Logs, flows, packets
6	Analytics require full knowledge	Predictive maintenance	Traffic engineering & provisioning
7	Hierarchical structure	Machines, production lines, factories	Devices, regions, networks
8	Varying requirements across applications	Maintenance *vs.* process optimization	Attack mitigation *vs.* load balancing
9	A priori unknown queries	state of production	network state



Data Store		Analytics		Application	Controller
Aggregate	**Transfer**	**Process**	**Infer**	**Decide**	**Implement**
- Reduce rate - Group & Combine - Summarize	- Scatter & Gather - Publish & Subscribe - Request & Reply - Forward & Replicate	- Map - Reduce - Apply	- Machine Learning - Graph Analysis	- Domain logic	- Conflict Resolution - Control of physical processes

Figure 3.2: The feedback loop in four building blocks.

Flexible yet resource-efficient data summarization is at the heart of extracting timely information from IIoT big data. For this purpose, we propose novel *computing primitives* that construct summaries of the data, which can be combined across the hierarchy.

3.2.1 Building Blocks

We base our architecture on four building blocks shown in Figure 3.2. These are *Data Stores* ("collect & aggregate"), *Analytics* ("transfer & process"), *Applications* ("model & learn"), and *Controllers* ("resolve conflicts & decide"). In the following, we describe each building block in more detail. For an illustration of the architecture, see Figure 3.3.

Data Store: A data store aggregates data using one or multiple instances of *computing primitives*, referred to as aggregators. We describe data stores in depth in Section 3.3 and computing primitives in Section 3.4.

Applications: To satisfy the varying needs of the users of IIoT big data, we envision a range of different applications. Each application embodies the decision logic for a single purpose. Applications can be long-running processes or enable short-term queries. They function as an interface for the users to gather information from the data stores. Thus, they can serve monitoring or reporting purposes. We envision that most applications run on a compute cluster, either at the edge or in a datacenter. Applications can be purely local, e.g., one that supervises the temperature of all machines of a specific type, or global ones, e.g., one that tracks the efficiency of different factories in different countries. Other application examples are process mining and predictive maintenance. Applications have two ways to interact with the physical world. They can either contact the controller to manipulate physical processes directly or install *triggers* in the data store, to influence future behavior. As the name suggests, triggers are triggered by events and then signal a controller. We envision that the latter can be used for simple conditions that need real-time reactions while the former is used to detect complex situations and may require complex actions, e.g., an update of the controller's logic.

Analytics: Many applications require processing beyond the computing primitives capabilities that can be installed within the data stores. They may require computation ranging from big data analytics, e.g., primitives that exploit "pleasingly parallel" computation (MapReduce, etc.), which can run on various devices, to more complex computations, even those that require specific hardware (e.g., GPUs, TPUs or FPGAs). In principle, we envision Analytics as a toolset that includes machine learning, graph processing, and big data systems such as Flink [121], Spark [101], etc. But, it does not have to stop here. Instead, it can also include visualization and statistics toolkits, e.g., R [83] or MATLAB [154], or tools to build interactive data visualizations, using, e.g., Shiny [23].

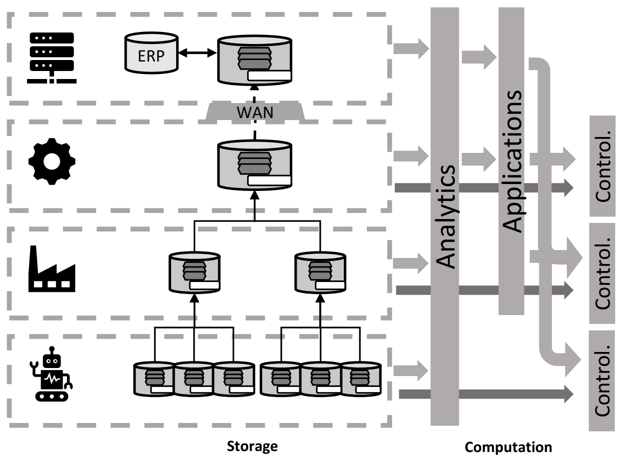


Figure 3.3: High-level view of the Architecture.

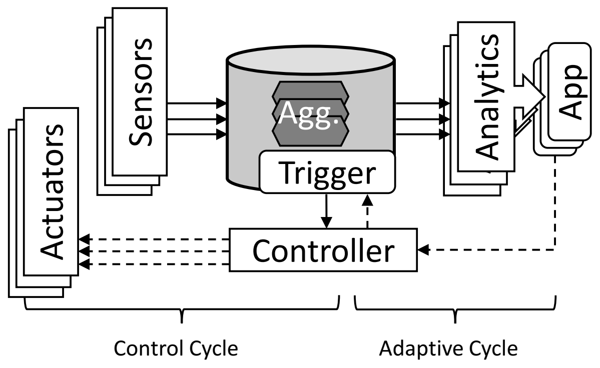
Controller: For operating at production speed, machines may not wait for input from applications. Yet, some validation may be necessary to avoid failures, e.g., raising a robot arm beyond its highest point. Thus, we envision a local control logic, the controller. The controller monitors the machine, e.g., using a set of patterns installed at the data stores as triggers. When a trigger matches, the controller is activated and regulates the machine accordingly. The controller's logic is installed and updated by individual applications.

3.2.2 Data and Control flow

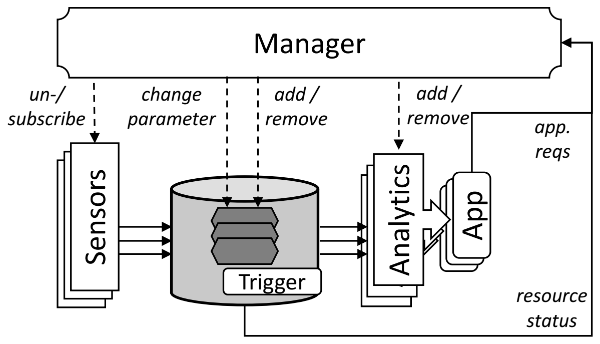
The Manager component manages the architecture. We describe both the data and the control plane in turn.

Data flow: Having described our system's components, we now turn to explain how data flows through the system and fires consecutive actions. Data is first pushed from the sensor (based on a request or an initial subscription) to the data store. In the data store, data is aggregated according to a chosen set of *computing primitives*. If any registered trigger matches a data summary (including possibly raw data storage), it activates the controller that regulates the respective machine(s). Independent of whether triggers are activated, the data store sends data to any registered Analytics pipeline. The pipeline performs pre-processing (e.g., using MapReduce), data transfer (scatter and gather semantics), and inference (e.g., using a Machine Learning algorithm). A pipeline feeds the processed data to one or possibly many applications. In turn, the applications decide whether to install new rules in the controller and forward the data for monitoring or reporting purposes. Conflicts between rules are resolved locally at the controller.

Control flow/Manager: Its configuration is a major challenge in realizing the architecture. Figure 3.4b shows the control plane. The Manager assigns and adapts resources according to the varying application needs. Each application records the application requirements in terms of the required data source and aggregation format (e.g., sample or histogram) and the required precision (e.g., sample rate or bin size). The manager then uses this information to decide (a) what data should be kept from which sensors (b) what computing primitive should be installed, (c) how the computing primitives should be configured, and (d) what analytics is deployed within the infrastructure (from the level of the machine up to the datacenter). Besides the storage within the data stores (shown in Figure 3.4b), the Manager tracks the availability of network bandwidth and computing nodes across the architecture. In summary, the manager controls all components of the architecture.



(a) Data Plane.



(b) Control Plane.

Figure 3.4: Single-level view of the Architecture.

Hierarchy: So far, we described the architecture that handles one data store, which corresponds to one single IIoT dataset. In the case of multiple IIoT datasets, each IIoT dataset is stored in its own data store. Further data stores exist to merge and aggregate data from multiple IIoT datasets, depending on the need of the applications. We have depicted an example of a hierarchy in Figure 3.3. In this example, the data stores responsible for combining data are located closer to a center of the infrastructure, i.e., closer to compute clusters, while the other data stores are located closer to the majority of physical processes. The manager decides what data stores should be deployed based on the applications' needs and connects the Analytics pipelines with the respective data stores.

3.2.3 Security, Privacy, Lineage, and Integration

Security & Privacy: To ensure privacy, the Manager can limit the set of summaries shared with the analytics component and at what granularity they. A local Controller may still use other summaries and more precise data. Security can be achieved by encrypting data along the Analytics pipelines, requiring updates to the Controller to be certified to ensure authenticity, and by requiring authorization before interaction with the Manager.

Lineage: An unavoidable problem in systems interacting with and processing sensor data is faulty or missing data. To address this problem, we need to track data as it moves through and is transformed by the system. This process is referred to as "lineage". Data lineage can, e.g., be used to identify faulty sensors or retract erroneous rules. Data lineage [159, 50] can be differentiated into schema-level and instance-level lineage. Schema-level lineage tracks the transformation of data from sensors to applications with respect to changing formats and the locations of transformations. It can help to identify how data came to its current format, but it cannot give information on

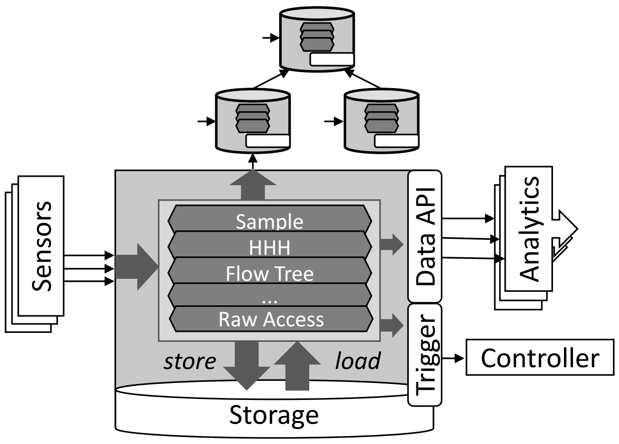


Figure 3.5: Structural view of a Data Store.

specific results. Instance-level lineage tracks individual items as they move through the system. It can be used to see how faulty data propagates, but it usually comes at a high cost (high overhead). The identification of a lineage mechanism for each computing primitive and analytic component at the envisioned data rate and with the flexibility of this system is an unsolved challenge.

Integration: Another problem that we have not touched upon is the integration of data beyond the operational data. We envision that applications will also make use of business data, e.g., the Enterprise Resource Planning database. This can be realized by the interaction between the business data and a data store located at a central data center.

3.3 Data Store

We show our vision for the data store in Figure 3.5. The data store selects and collects data from sensors and then feeds their data into aggregators, instances of computing primitives that have subscribed to the respective data streams. Queries received by the data store are broken into sub-queries and are forwarded to the respective aggregator. Sub-queries for aggregators stored at other data stores are forwarded or resolved on a local replicate. The main responsibility of the data store is to ensure that the storage and network resources are efficiently used:

Storage: Note that the data store is the only entity in our architecture, which stores data. All other elements might cache data or intermediate results (depending on their implementation), but they are not expected to persistently store raw or summarized data. Hence, when a data store chooses to delete data, it cannot be recovered. Thus, storage space has to be carefully allocated and managed.

We identify three basic strategies for storing data in the data store: (1) storage with predefined expiration, (2) storage using a round-robin mechanism, and (3) storage using a round-robin mechanism and hierarchical aggregation. The first strategy gives application developers the guarantee that data is stored for a fixed time duration. Note, choosing the time period optimally in advance may be difficult. The second strategy optimizes the use of storage in the sense that it fully utilizes the storage. In this case, the duration that the data is stored depends on the data rate. The third strategy is a combination of the first and second one in the sense that older data is not expired but aggregated to a coarser granularity with a smaller footprint and then stored. This guarantees long-term storage but at the price of reduced detail due to aggregation. More sophisticated strategies may regard stored data as a storage investment that has to be traded off against future queries that it will help answer. These strategies may compress or delete data that is deemed of lesser interest in the long term.

Network Transfers: Different data stores gather data at different locations. Yet, some analysis requires their joint processing. Thus, data in one data store may have to be combined with data from other data stores to process queries. In this case, the data store has the choice of (1) shipping the query to the data or (2) *replicating* the respective aggregator(s). A basic strategy for this decision is to replicate the data produced by an aggregator when the data it holds has been accessed at least n number of times by a remote data store, when at least b of its bytes have been transferred or when it has created a data transfer volume of at least p percent of its own storage volume. Each of these strategies is heuristic in nature. More sophisticated strategies can be developed using predictions of future accesses. We address this problem in more detail in Section 3.5.

3.4 Computing Primitives

A major challenge in realizing our architecture is the efficient summarization of data across multiple sources and locations to answer a priori unknown queries. This is accomplished by computing primitives, which can be used by the individual data store to create data summaries, aggregates of raw data from the incoming data streams. Computing primitives can use aggregation methods from simple statistics over time bins (e.g., sum, mean, median, and standard deviation) and sampling methods to more complicated streaming algorithms (e.g., heavy hitter detection or even hierarchical heavy hitter detection).

Yet, none of the above methods are suited to unleash the full capabilities of our proposed architecture as they do not support any of the following: support arbitrary queries, enable the combination of data summaries, have an adjustable level of aggregation, self-adapt to incoming data and queries and take domain knowledge into account, to create more meaningful summaries. Thus, we need novel computing primitives. In the following, we discuss these desired properties in more detail.

3.4.1 Design Properties

Support arbitrary queries: Each computing primitive needs to enable arbitrary queries, particular a priori unknown queries, on its respective data summaries. While the format of the queries depends on the particular data organization, each primitive should permit flexibility, e.g., in the degree of precision.

Can combine summaries: For integration into the hierarchy of data stores, computing primitive should be able to combine data summaries. Each summary represents a single time interval and a collection of data streams at a single location. Hence, a combined data summary can answer queries over data from multiple locations (including differences between the locations).

Adjust the level of aggregation granularity: In most cases, the raw data is produced at rates that are too high for storage or timely processing. Therefore, computing primitives should aggregate data into summaries that can be stored and timely processed. Furthermore, to deal with volatility in the rate of incoming data streams, computing primitives should be able to adjust their level of aggregation granularity over time.

Self-adaptive: Every summary produced by a computing primitive increases the storage and processing footprint of the data store. The summary should continuously re-organize the data it stores and its level of aggregation granularity according to the incoming data streams and queries. If the manager were to know all future requests in advance, e.g., because the application set is fixed, it would be straightforward for the manager to choose the appropriate aggregation level for the computing primitive. Yet, most of the time, this information is not available. Therefore, computing primitive should ideally be able to adjust the granularity on demand. Where this is not possible due to the aggregation method, the applications may be forced to specify at which aggregation level they want to operate.

Uses domain knowledge: Knowledge of the data domain can help create more meaningful computing primitives where aggregation has a semantic relationship to the data. Such computing primitive can enable queries to express their desired level of aggregation in terms of the domain.

3.4.2 Toy Example

A toy example of a computing primitive can be an aggregator that uses random sampling. This primitive can produce a data summary in the form of a sampled time series and has the following properties:

- **Query:** It enables queries on a time series, e.g., by selecting all data points in a given time frame that exceed a given value.
- **Combine:** Two summaries, i.e., time series, can be combined by combining individual data points from the respective time series.
- **Aggregate:** The level of aggregation can be changed by adjusting the sampling rate of the time series.
- **Self-adapt:** The time granularity required by incoming queries and the rate of the incoming data can be used to adjust the sampling rate.
- **Domain knowledge:** This computing primitive using random sampling is an example of aggregation without domain knowledge.

For a more detailed example we refer the reader to work by Saidi et al. [134].

3.5 Data-driven Transfer Optimizations

Conventional transfer optimizations (see Subsection 1.1) have been designed with a focus on data independence and optimal worst-case behavior. Less thought was given to exploiting patterns in the data (in the broadest sense) and towards the average-case performance. This approach is not only symptomatic for computer networks but has also been applied to database and algorithm design in general [133]. One of the main reasons for this approach was the limited available processing power that could be expended on identifying patterns. The continuing increase in processing power and the improvement of machine learning algorithms are opening new opportunities for algorithms and systems that improve average-case performance while retaining similar worst-case performance. Work by Tim Kraska et al. [95, 94] shows the power of this approach for database area. We believe that a data-driven approach can help to reduce data transfer volume further.

In our architecture, each data store can maintain multiple computing primitives that retrieve and combine data from other data stores. Due to the size of the data, this can lead to congestion in the network, particularly in the connection among multiple edge locations or one edge location and the cloud. In the following, we discuss the four design properties of data-driven transfer optimizations. We then give examples for data-driven transfer optimizations that show some of these properties.

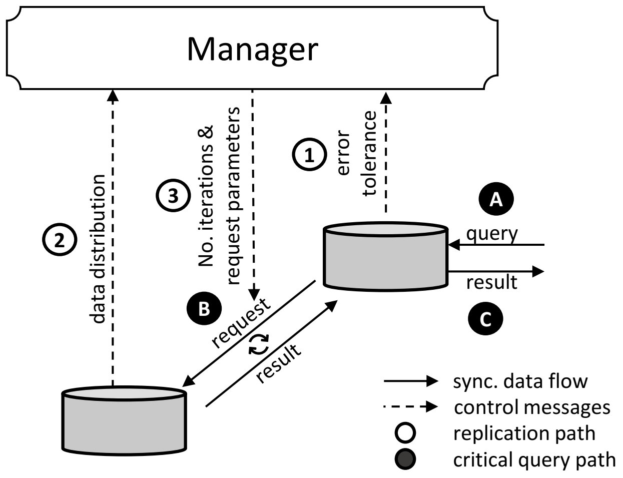


Figure 3.6: Optimizing iterative data exchanges between two data stores.

3.5.1 Design Properties

The design properties of data-driven transfer optimizations are:

- **Monitor network performance:** Data-driven transfer optimizations monitor the network performance to guide their internal behavior.
- **Mine data and data access:** Data-driven transfer optimizations mine the data and data access for patterns that are opportunities for a reduction of data transfer.
- **Adjust data precision:** Data-driven transfer optimizations adapt the data precision according to the availability of capacity. Where applications allow and circumstances require, transfer optimizations reduce the precision of the data to optimize network performance.
- **Avoid or compress**: Data-driven transfer optimizations can take the form of compressing existing data transfers or avoiding transfers altogether (e.g., by replication).

Optimally a data-driven transfer optimization expresses all of the properties. However, not all of them are applicable to all applications and environments. In the following, we describe three data-driven transfer optimizations for different data processing frameworks. We develop these data-driven transfer optimizations further in the next chapters.

3.5.2 Example: Client-Server Query-Result Exchanges

Many applications are likely to use database-like queries to access summaries and historic IIoT big data. Internally, the architecture can process incoming queries by forwarding it to the data store closest to the data or by replicating data from this data store to the data store closest to the application. The cost of each option depends on the frequency by which data is accessed. Figure 3.7 illustrates this example.

When a data store receives a query for combined data A, it must first check whether the respective data is locally available. In the default case, the data store receiving the query queries the data store B holding the data. Next, it collects the result C before it can process D and answer the query E. The time spent on retrieving data from another data store increases the latency of the individual query and increases network utilization. The former can

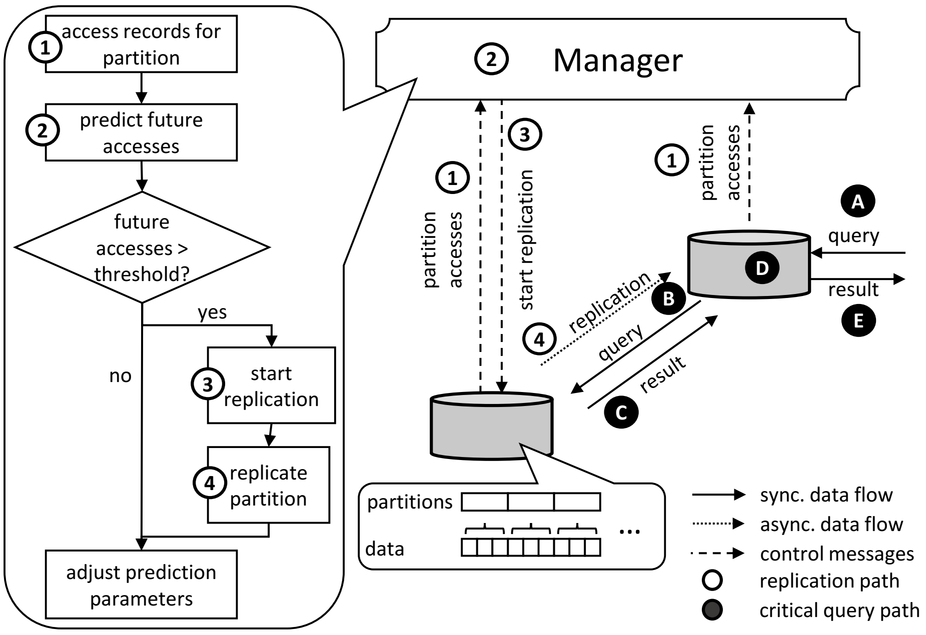


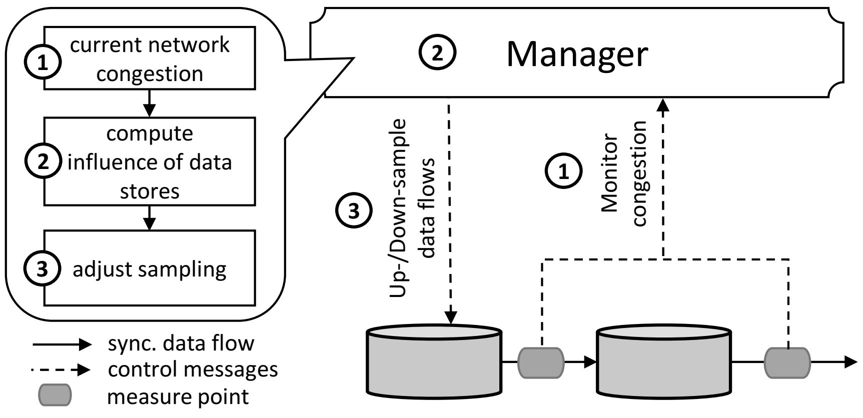
Figure 3.7: Optimizing data transfers with adaptive replication.

become a problem for applications that have low latency requirements, while the latter can degrade third-party application performance.

Our architecture (see Section 3.2) splits the data hold by any data store in partitions. The manager tracks access to each data partition ① and records the access time and the respective data transfer volume. The manager uses this information to classify partitions and predict future data transfers ②. If the predicted future access exceeds the replication cost, the manager decides to initiate the replication of a partition. The data is replicated to the first data store ④, and future queries are resolved locally at this store. We discuss strategies to cope with the uncertainty of future accesses in Chapters 4 (single time window) and 5 (multiple time windows).

3.5.3 Example: Iterative Message Exchanges

Another data transfer pattern for applications accessing multiple distributed data stores is the iterative selection and pre-processing of data in the form of iterative message exchanges. In this type of data exchange, the cloud attempts to narrow down the data summary that it actually needs over the course of multiple phases. See Figure 3.6 for a representation of this theme. In Ⓐ, a query comes into the data store on the right. This data store does not contain the data, so it has to collect it from the data store on the left. Therefore, it sends multiple requests Ⓑ to the left data store. With every result, the right data store can further narrow down the data it needs. In the end, it returns the result Ⓒ to the origin of the query. To reduce the number of iterations and the data transfer volume in contrast to a data-agnostic worst-case algorithm, the right data store can ascertain the error tolerance of the query and communicate this to a central manager ①. The left data store can estimate the distribution of the data and also send this to the central manager ②. Using this information, the central manager can then decide the best interactions between the two data stores in terms of the number of exchanges (or phases) and the requests themselves ③. Chapter 6 discusses distributed top-k algorithms that are a simple example of this kind of data exchange and optimization.



① current network congestion
② compute influence of data stores
③ adjust sampling
② Manager
③ Up-/Down-sample data flows
① Monitor congestion
sync. data flow
control messages
measure point

Figure 3.8: Optimizing data transfers by down-/up-sampling.

3.5.4 Example: Dataflow Program

Applications that require knowledge of new data in real-time or near real-time rely on some type of data stream processing. Many stream processors are based on the dataflow model. They implement programs as a chain of operators that process data and forward it to the next operator. For these programs, an unpredictable rise in data generation rates or software and hardware faults can lead to situations where an operator processes and transfers data slower than it receives data. This leads to congestion, the decrease of end-to-end latencies, and under some circumstances further reduction of the program's throughput. As resource-constrained environments, industrial facilities can often not remove congestion by scaling the respective operators. Instead, they can use load shedding and accept a loss in precision for more predictable end-to-end latency. Figure 3.8 shows such an approach for our architecture.

At step ① the Manager continuously monitors congestion of the network, possibly at the application-level. This information is integrated into an internal model in step ②. Depending on the severity of the congestion, actions are taken to down-sample the ingested data, when congestion increases, or up-sample the data when no congestion is monitored for a prolonged time interval (step ③). We discuss this approach in more detail in Chapter 7.

3.6 Related Work

At this point, standardization bodies, researchers, and the industry have developed architectures with widely different perspectives on the IIoT. Here, we give an overview of the most prominent architectures and describe their differences from our approach. We start with the largest top-down architectures built explicitly for the IIoT. Next, we discuss architectures that focus on the message bus of IIoT systems as the central component. Finally, we discuss IIoT architecture taxonomies.

The two biggest standardization bodies working on the IIoT are the German government initiative Industrie 4.0 established in 2012 (which evolved since from a working group to an industry platform) [124] and the Industrial Internet Consortium (IIC) established in 2014 [85]. The platform Industrie 4.0 created the Reference Architectural Model Industrie 4.0 (RAMI 4.0) [103, 163]. RAMI 4.0 includes a three-dimensional IIoT model that offers a high-level view of IIoT architectures. The three dimensions include the architecture's life cycle ("Life Cycle & Value Stream"), the location-based hierarchy from products to the enterprise to the "Connected World" ("Hierarchy Levels"), and a semantic hierarchy from asset to information to business concerns ("Layers"). The IIC created the

Industrial Internet Reference Architecture (IIRA) [84, 86]. Similar to RAMI 4.0, it creates an IIoT space to better compare and standardize IIoT architectures. At its core, it combines different viewpoints including a business, usage, functional, and implementation viewpoint. Higher-order viewpoints guide the development of lower-order viewpoints, while the lower-order viewpoints provide validation. The views are enriched by considerations of the IIoT architecture's life cycle and differences between industry sectors. A joint whitepaper of Industrie 4.0 and the IIC [107], has pointed to the service-oriented architecture of RAMI 4.0 and view-oriented architecture of the IIRA as the biggest differentiators. Different from RAMI 4.0 [163] and IIRA [84], our architecture covers only the implementation part of IIoT architecture with a focus on the data flows.

At a smaller scope, standardization bodies have built architectures around the communication between applications and devices. In Subsection 2.1.2, we mention the Open Platform Communications (OPC) Foundation and their proposed service-oriented architecture, the Unified Architecture (UA) [100]. Different from RAMI 4.0 and IIRA, the OPC UA focuses on the convergence of operation technology (OT) and information technology (IT) rather than an overarching IIoT architecture. The OPC UAs central element is a message bus for secure and reliable data exchange and information model building blocks for semantic structuring of the data. It provides several services for data exchange including method execution, query look-ups, and publish-subscribe [119]. It also supports local data aggregation [118].

The oneM2M global partnership (established in 2012) created the oneM2M service layer [117] with a similar goal but from a different perspective. Where the OPC UA is an industrial standard first that is built on top of earlier work of the OPC, the oneM2M is heavily influenced by the non-industrial IoT. Similar to the OPC UA, it also provides a data modeling approach and a service-oriented architecture around a central message bus. The service layer is designed as a middle layer between the application layer and the network layer. It provides functionality through fourteen common service functions including security, device discovery, and data management. Similar to other IoT systems, it leverages a resource-oriented architecture and manages data access through CRUD (create, read, update, and delete) primitives. For a comparison of both approaches with a focus on interoperability see [37].

Other architectures besides the oneM2M service layer and OPC UA have been built around the communication channel. Cunha et al [51] discuss the use of the publish-subscribe protocol MQTT as a central communication platform to communicate data from sensors to company systems (e.g., Enterprise Resource Planning). Schel et al [136] describe a manufacturing service bus (MSB). In their design, manufacturing systems and applications connect as services to the MSB. Data in the MSB is routed via manually configurable integration flows in a publish-subscribe manner. These architectures are orthogonal to our approach. Where they focus on device addressing and data modeling, we focus on the flexible aggregation of data and its effect on computation, storage, and network resources.

Other work developed taxonomies to classify IIoT systems. Boyes et al. [28] categorized IIoT systems into industry sector, device location, connectivity, device characteristics, device technology, and user type. Our architecture is closest to their [28] device location-centric view, as our architecture starts from the IIoT device location in the data aggregation hierarchy (see Figure 3.3. In this category, we also find work by Bradicich et al [29] and the Purdue reference model. Bradicich et al [29] split the IIoT architecture into four vertical stages: sensing, data acquisition, Edge IT, and Cloud. The Purdue reference model, as described in [28], separates the IIoT architecture into six horizontal levels. The first four (industrial process, basic control, area supervisory control, and site manufacturing operations & control) are located in the manufacturing zone, while the latter two (site business planning and logistics network & enterprise network) are located in the enterprise zone. While we provide a view of the production hierarchy in Figure 3.3, we do not specify the number or details of each level.

Aziz et al [13] have done a case study for IIoT architectures in the mining industry. They divide the "Industrial Internet" into five layers: business & application (e.g., ERP), analytics (data batches and data streams), information (data repository, schema, etc.), operations (prognostics, optimization, etc.), and control layer (devices, sensing, etc.). Our architecture sits across the middle three layers. Additionally, Aziz et al. [13] identify four cross-cutting Distributed Data Management functions: publish & subscribe, reduction & analytics, one-time & continuous queries and storage, persistence & retrieval. These functions are a subset of the building blocks we describe in Subsection 3.2.1.

The authors of [129] propose a monitoring architecture that has some similarity to the one we propose. The architecture uses data stores over multiple levels. The work differs by the stronger focus on the underlying wireless transfer protocol and the focus on a single monitoring use case.

IIoT architectures are complex structures that involve many stakeholders and technical processes. They evolve to best serve business goals. Our architecture does not aim to cover all stakeholders and scenarios. Instead, we focus on the effect that the growing data generation has on computing, storage, and networking resources. Our proposed computing primitives and data-driven transfer optimizations are meant to reduce this effect. Hence, our architecture is compatible with both approaches that cover a wider spectrum such as RAMI 4.0 or the IIC's IIRA, and approaches that focus on the device integration into a central message bus and the data model such as the OPC UA and the oneM2M service layer.

3.7 Summary

Using two use cases, smart factory and network monitoring, we point out why and how IIoT datasets arise. These datasets cannot be stored or processed locally. Yet, they need to be processed to support the correct interactions with and the necessary control loops of the real world. Indeed, an accurate reflection of the digital world with bounded capacities for communication, storage, computation, and accuracy is challenging.

To address this challenge, we propose an architecture for handling IIoT big data, which consist of a hierarchy of data stores, applications to address the needs of the users via data analytics, controllers to facilitate interactions with the physical world as directed to by the applications, and a manager that controls the data flow.

Further, we highlight the need for novel computing primitives to summarize data efficiently and data-driven transfer optimizations to limit the impact of individual data exchanges on the network. We have introduced three examples for data-driven transfer optimizations that we expand upon in the following chapters.

4 Online Replication Strategies for a Set of Immutable Data Partitions

One use of edge nodes is as databases that return data in response to queries. In this chapter, we introduce a data-driven transfer optimizations for this use case. To account for the situation that edge nodes retrieve data from other edge nodes, we refer to the cloud and edge nodes that retrieve data as front-ends and edge nodes that store data as back-ends (see Figure 4.1). A front-end forwards a query to a back-end, which processes the query using its local data store and *ships* the result back. Alternatively, a front-end can request a back-end to *replicate* data to its local store and then use it to process queries locally.

We focus here on purely analytical use of the data and thus assume that the data is immutable, i.e., that it is not updated by queries. Furthermore, we assume that the data is horizontally split into equally large *data partitions* and that each partition can be replicated independently. Partitions are used by systems to speed up lookups [148], enable parallel data processing [26], and reduce storage [98].

Front-ends that replicate all data can incur a massive data transfer volume due to the size of the data. However, even front-ends that exclusively rely on queries to transfer data can still incur unnecessary data transfer volume if the queries process the same raw data. In this situation, data transfer volume can be reduced by selective replication, i.e., by first replicating the respective data partitions and then processing the queries locally. The contrast between the two approaches is influenced by the popularity of the raw data. The more popular the data is, the more is gained by replicating this raw data.

Data popularity is often skewed, i.e., a fraction of the data is often heavily accessed while the remainder is accessed infrequently or not at all. The popularity skew occurs naturally in the context of sensor data, where most data reflects monotonous behavior and only a small part of the data pertains to anomalous behavior. Therefore, selective replication can, in principle, greatly reduce data transfer volume. Unfortunately, it is not immediately clear how

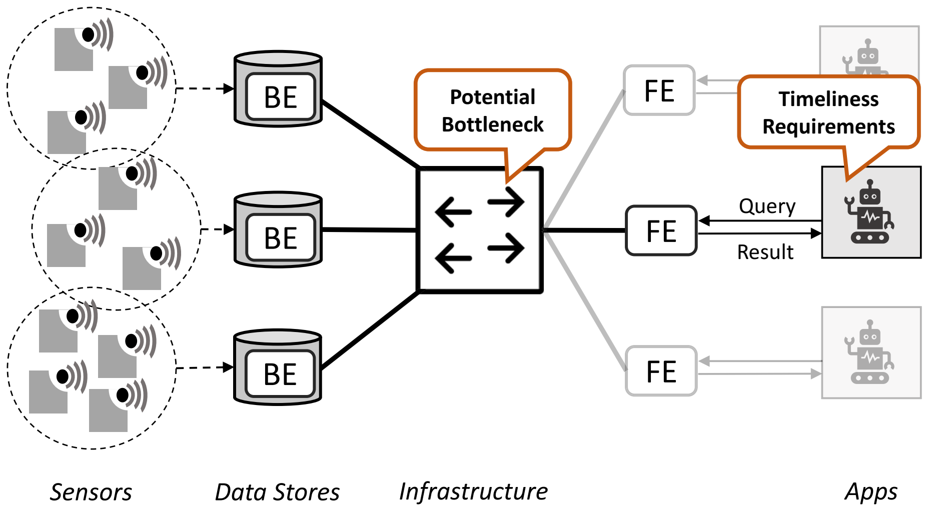


Figure 4.1: IoT scenario with distributed mega-datasets. (Front-ends and back-ends are abbreviated as FE and BE respectively.)

current data popularity translates into future data popularity. If the popularity fades after the replication, then the data transfer volume can be increased rather than reduced. We propose *online replication strategies* to make replication decisions based on past data accesses. We introduce both heuristic and competitive strategies and compare them against the two base cases of shipping all query results or replicating everything. For our evaluation, we use a simulation that we base on two real-world database traces.

Our Contributions

- We formalize the trade-off between shipping query results and replicating data partitions respective to data transfer volume.
- We design a competitive strategy based on the ski-rental algorithm to make replication decisions in the face of uncertainty.
- Our evaluation using two real world-query traces shows that the competitive strategy reduces transfer cost between 18% to 51%.

4.1 Problem Formalization

Traditionally, distributed databases rather ship query results than replicate partitions. Even modern databases make only light use of replication. They rely either on a static replication scheme or require explicit configuration by a human administrator [48, 79]. Here, we use replication as a tool to reduce the data transfer volume, i.e., the data volume that is moved over the network to answer queries and reduce query response times as a side effect. In this preliminary work, we focus on the interaction between a single front-end and a single back-end. We also restrict this model to a read-only workload consistent with the immutable machine and sensor data often produced in the Internet of Things.

In the following, we specify, for every function, its domain and the target set (the co-domain) using the $\rightarrow$ symbol. We specify the function itself using the $\mapsto$ symbol. Where unambiguous, we omit the domain and codomain.

4.1.1 Query Model

In our model, the front-end receives queries over time from the set of queries Q. At any point in time t, it receives query sets Q_t where $Q_t \subseteq Q$. We refer to the history of query sets received up to the point t as $\hat{Q}_t$ and to the set of all queries received up to the point t as $Q_{(t)}$ where $Q_{(t)} \subseteq Q$. We assume that the back-end stores its data in partitions P of fixed size and that the queries can be answered using data stored in the partitions at the back-end. Partitions can be replicated to the front-end. At any point in time t we denote the set of replicated partitions as P_t^R where $P_t^R \subseteq P$. Similar to the queries, we describe the history of replicated partition sets up to time t with $\hat{P}_t^R$ and all partitions replicated at the time t with $P_{(t)}^R$ so that $P_{(t)}^R \subseteq P$. For now, we assume that partitions, which have been replicated at some point in time remain replicated. For convenience, we summarize all variables in Table 4.1.

We assume the use of partitions since partitions are often the result of optimizing for memory/disk accesses. The common assumption is that if a row in a partition is accessed, other rows within the partition are likely to be accessed as well. For the same reason, we see it as an appropriate transfer unit for our model.

In this chapter, we assume that the cost of replicating a partition p is approximated by the size of the partition $size(p)$.

$$size\text{: } P \rightarrow \mathbb{N}^+ \tag{4.1}$$

A query q is answered using data from one or several partitions. We refer to the contributions of each partition p to the query result as the *partition transfer volume* and $result(q,p)$:

$$result\text{: } Q \times P \rightarrow \mathbb{N}_0^+ \tag{4.2}$$

Thus, we can determine the overall contribution of each partition p, or *aggregate partition transfer volume*, to queries up to time t as $record(Q_{(t)}, p)$:

$$\begin{aligned} record\text{: } & Q \times P \rightarrow \mathbb{N}_0^+ \\ & (Q_{(t)}, p) \mapsto \sum_{q \in Q_{(t)}} result(q,p) \end{aligned} \tag{4.3}$$

Table 4.1: Variables for Online Replication Strategies.

Var.	Description
Q	Set of queries over the whole runtime
Q_t	Queries received at time t
$\hat{Q}_t$	Sequence of query sets up to time t: $(Q_1, Q_2, \ldots, Q_t)$
$Q_{(t)}$	Set of queries observed up to time t: $\bigcup_{i=1}^{t} Q_i$
P	Set of partitions of the data
P_t^R	Set of partitions replicated at time t
$\hat{P}_t^R$	Sequence of partition sets replicated up to time t: $(P_1^R, P_2^R, \ldots, P_t^R)$
$P_{(t)}^R$	Set of partitions replicated up to time t: $\bigcup_{i=1}^{t} P_i^R$

4.1.2 Cost Model

The choice of the point in time at which a partition is replicated influences the network cost in terms of data transfer volume. At any moment t, the transfer cost is the sum of the cost of replicating partitions and the cost of shipping query results:

$$\begin{aligned} &\textit{cost: } Q \times P \to \mathbb{N}_0^+ \\ &(Q_t, P_t^R) \mapsto \sum_{p \in P_t^R} \textit{size}(p) + \sum_{p \notin P_t^R} \sum_{q \in Q_t} \textit{result}(q,p) \end{aligned} \tag{4.4}$$

The total cost is simply a summation of the cost up to that moment:

$$\begin{aligned} &\textit{total-cost: } 2^Q \times 2^P \to \mathbb{N}_0^+ \\ &(\hat{Q}_t, \hat{P}_t^R) \mapsto \sum_{i=1}^{t} \textit{cost}(Q_i, P_i^R) \end{aligned} \tag{4.5}$$

4.1.3 Strategies

The optimal point in time at which a partition should be replicated depends on the number of queries that a partition will receive after this point. This information is usually unknown in advance. Thus, our problem has to be solved online [4].

An online replication strategy in our model decides which partition should be replicated at any given moment based on the previously seen queries and its previous decisions.

$$\begin{aligned} &\textit{strategy: } 2^Q \times P \to P \\ &(\hat{Q}_t, P_{(t)}^R) \mapsto P_{t+1}^R \end{aligned} \tag{4.6}$$

Two naïve strategies are to use either only replication or only shipping and disregard any knowledge gained from previous queries.

$$\textit{replicate-only (RO): } (Q_t, P_t^R) \mapsto P \tag{4.7}$$

$$\textit{ship-only (SO): } (Q_t, P_t^R) \mapsto \emptyset \tag{4.8}$$

Our goal is to devise strategies that improve on these basic strategies and come close to the *optimal offline strategy*. The optimal solution is to replicate partitions, whose record accumulates, over the whole course of time, accesses that exceed the cost of replicating the partition—the partition size. For this purpose, we consider finite query sequences of length n. The optimal strategy is then:

$$\begin{aligned}\textit{optimal (OO): } (Q_t, P_t^R) \mapsto \{p | p \in P \\ \wedge\ \textit{record}(Q_{(n)}, p) > \textit{size}(p)\}\end{aligned} \tag{4.9}$$

For some use cases, storing the full access history of queries may prove to be prohibitively expensive. In these cases, only the aggregate partition transfer volume can be stored. The strategies presented in this chapter would still work, although with possibly reduced accuracy.

4.2 Online Replication Strategies

Given that we deal with an online problem where an optimal strategy has to be selected at any point in time without knowledge of the future, we turn to the area of online algorithms [65]. Our replication problem resembles the ski rental problem [92, 65]. In this problem, a skier faces the choice between buying a ski-set and renting a ski-set every day of his skiing career. Buying the ski set has the advantage that no future cost for renting will accrue. However, in the worst case, the skier stops skiing on the same day (for whatever reason). Then, the sum of money spent on buying was almost "wasted". Renting the ski-set in this situation would have been cheaper. But, if she keeps on skiing, the accumulated cost of renting can easily exceed the cost of buying them in the long run. The third option is to switch from renting to buying after a number of skiing days or after a certain sum has been spent on renting. Yet, choosing this threshold to minimize the absolute cost in advance, without knowledge of the future, is impossible. This ski rental problem is similar to our problem, whereby the small renting cost is analogous to the cost of shipping query results, the cost of buying ski is analogous to replicating a partition, and the workload is not known in advance.

Even though future events in the ski rental problem are unknown, the worst-case is known and can be summarized in the form of the competitive ratio. The competitive ratio captures the performance of an online algorithm in comparison to an offline one that has perfect knowledge of the future. It can be shown [92] that buying the ski after spending the same amount on renting means that the skier pays at most twice the sum an all-knowing actor would have spent. Similarly, we too know that replicating a partition after a data volume equal its size has been shipped is at most twice the cost an all-knowing actor would have spent. To capture this, we introduce *threshold strategies* with the parameter threshold τ:

$$\begin{aligned}\textit{threshold}(\tau)\textit{: } & 2^Q \times P \rightarrow P \\ & (Q_{(t)}, P_t^R) \mapsto \{p | p \in P \\ & \wedge\ \textit{record}(Q_{(t)}, p) > \tau\}\end{aligned} \tag{4.10}$$

The threshold strategy replicates a partition once it is responsible for a data transfer volume equal to τ. The *ski rental strategy (SR)* is one example of a threshold strategy where the threshold is the size of the partition. If all partitions have the same size, this is a single value. If they have different sizes, then the threshold corresponds to their individual sizes.

A key difference between our problem and the ski rental problem is that our problem includes multiple partitions. We can either treat them as separate ski rental problems or we can consider them as a single problem. In the latter case, we may be able to estimate the distribution of accesses over the partitions. This approach is related to the constrained ski rental problem [91, 68, 93]. Here, the probability for the number of skiing days is known to follow a given probability distribution. This allows the creation of a threshold-based strategy for the average case in terms of the expected cost [68, 93].

Thus, we estimate the distribution of aggregate partition transfer volumes from the history of accesses (the *record* in our model) which results in the *reactive threshold strategy (RT)*. This is another example of the threshold strategy, but one that uses a threshold $\tau(t)$ which changes over time. We call threshold strategies that use different thresholds over time *dynamic* in contrast to *static* strategies that use only a single threshold. For RT, the threshold is estimated from the existing query history up to present time t.

$$\tau_{RT}(t) \mapsto argmin_{\tau} \sum_{i=1}^{t} cost(Q_i, threshold(\tau)(Q_{i-1}, P_{i-1}^{R})) \tag{4.11}$$

Given a finite query sequence of length n, we can use the same method to compute the optimal threshold for this query sequence. We refer to the strategy that uses this threshold as the *optimal offline threshold strategy*. Its performance is an *upper bound* for all *static threshold strategies*.

$$\tau_{OT}(t) \mapsto \tau_{RT}(n) \tag{4.12}$$

Table 4.3 summarizes all strategies and their parameters.

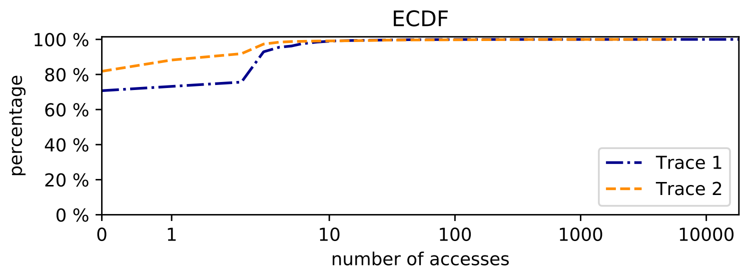
4.3 Datasets

Ideally, we would like to deploy and evaluate our strategies in the real world. However, this is impossible as IoT deployments at this scale are still emerging, and the currently existing ones often do not require a distributed setting. Thus, we do a query trace-based evaluation. For this, the query trace should (a) come from a setting with a partitioned data store and (b) include the size of query results by partition.

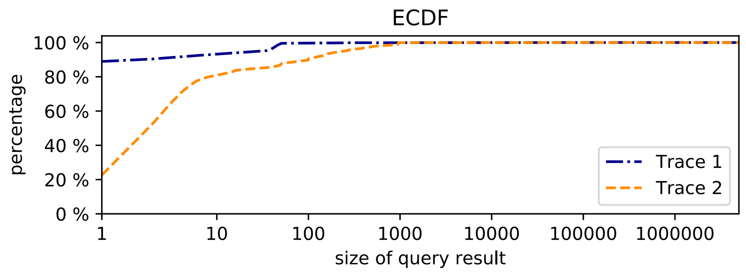
Unfortunately, traces with this degree of detail are rare for a variety of reasons. First, current database systems, typically, only include statistics on how often a specific database partition was accessed. Second, even when they can record per query accesses, this feature is typically disabled for performance reasons. Third, even if such a trace exists, it often contains private, business-critical information and, hence, their owners rarely share them, even for research purposes. Yet, the alternative of relying on synthetic datasets also has the major disadvantage that the data is often much more regular than in the real world [82].

Table 4.2: ERP traces statistics.

Name	Trace 1	Trace 2
Table size in rows [million]	24	100
Number queries [million]	1.28	2.49
Duration [days]	≈ 3	≈ 3
Accesses in rows [million]	34	137
Avg. rows per query	26	55



(a) ECDF of number of times that individual rows are accessed.



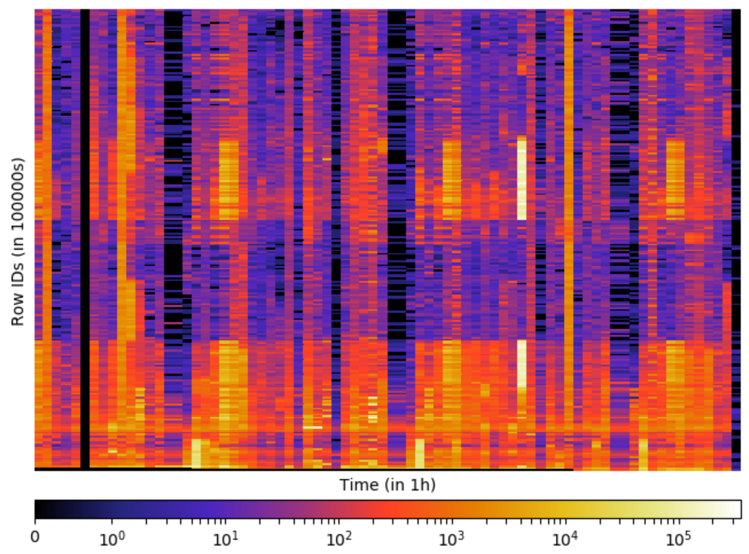
(b) ECDF of result sizes per queries.

Figure 4.2: Trace characterization.

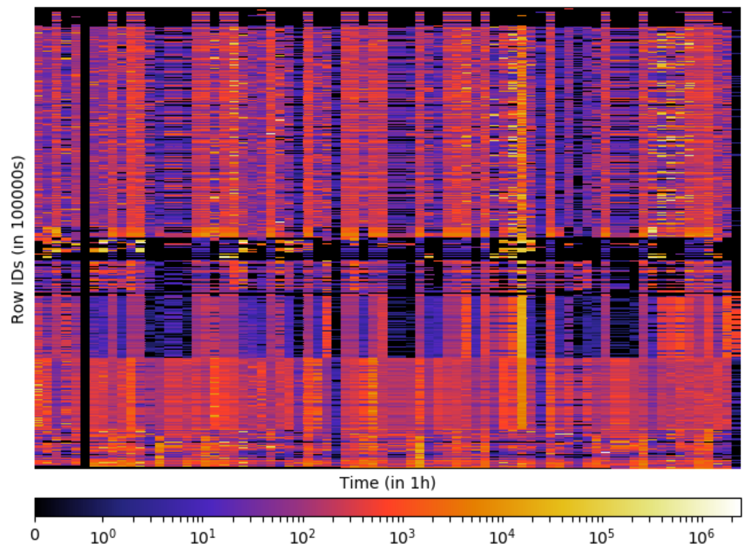
Against all these odds, we were able to get access to two large database query traces with all required details about the queries and the results. Both traces were gathered by Martin Boissier et al. [27] by instrumenting a live production SAP-based Enterprise Resource Planning (ERP) system of a Global 2000 company. Each trace was collected using a two-step process. In the first step, all queries to one table within a live ERP database were recorded for a three-day time period. To reduce the overhead on the production system, this trace was sub-sampled to only contain queries that appeared within the first two minutes of every ten-minute time window. The result of this step is a sequence of queries with their timestamps. In a second step, the queries were re-run against a copy of the live system to determine the size of their result set, including all rows that are part of the result set. Note, these tables, in contrast to our assumed setting, are not immutable per se. However, since the result rows are recorded at the level of row ids and since these row ids are not reused, the data is a decent approximation for our use case. Each trace contains more than 1 million queries and accesses more than 20 million rows, which corresponds on average to 26 resp. 55 row accesses per query for Trace 1/2. See Table 4.2 for a summary.

Even though the traces were recorded from an ERP system rather than an IoT system, we find that the access distribution per row is highly skewed. Figure 4.2a depicts the empirical cumulative distribution function (ECDF) of the accesses per row id for both traces. Amazingly, more than 70%/80% of the rows are never accessed at all. More than 95% of the rows are accessed less than ten times. Only a small fraction (less than 1%) of the rows are very popular with more than 100 accesses. Query result sizes are heavily skewed as well. See Figure 4.2b which does not even show queries with empty result set sizes. The result sets of more than 80% of the queries include less than ten rows. Hereby we note that Trace 1 is significantly more skewed than Trace 2. In conclusion, the access patterns and the result sizes of both traces are highly skewed, which supports our earlier assumption. IoT use cases are likely to be even more skewed and, thus, would likely give even better results for our approach.

Recall our presumption that data is organized in partitions. Yet, the traces do not contain any information about partitions. To nevertheless use the trace for our evaluation, we add partitions to the trace, whereby each partition contains 100K adjacent rows. Similar results (not shown) apply for different partition sizes. When looking at the accesses per partition over the duration of the trace, we noticed some periodic access which resulted in repeating daily (1 AM) high volume access. After closer investigation, we concluded that these are likely to be the result of



(a) Trace 1+.



(b) Trace 2+.

Figure 4.3: Heatmaps of the number of accesses to each partition across time periods.

daily maintenance jobs and eliminated them by replacing them with the accesses from the previous hour. We refer to the cleaned traces as “Trace 1+” and “Trace 2+” respectively. Figure 4.3a and Figure 4.3b show the resulting access patterns as heatmaps (using a logarithmic scale) per partition and hour. Frequent accesses to the same partition result in light color entries, while low frequent ones result in dark color entries. Notice that most entries are dark. However, some rows are much lighter than others. These are partitions that are frequently accessed (heavily used) and, thus, should be replicated. Partitions that are rarely accessed should not be replicated. Rather, the results should be computed at the back-end and shipped to the front-end.

Given that the distribution of row accesses is skewed (see Figure 4.2a), we expect that the distribution of data transfer over partitions is skewed as well. This is indeed the case (see Figure 4.4). Note, we normalized the x-axis with respect to the partition size. For example, for Trace 1+ about 60% of all partitions have data transfers less than

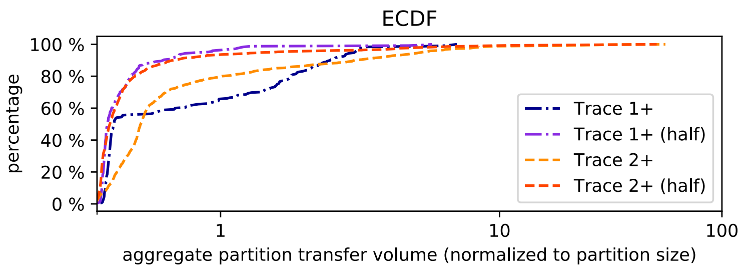


Figure 4.4: ECDF of aggregate partition transfer volumes.

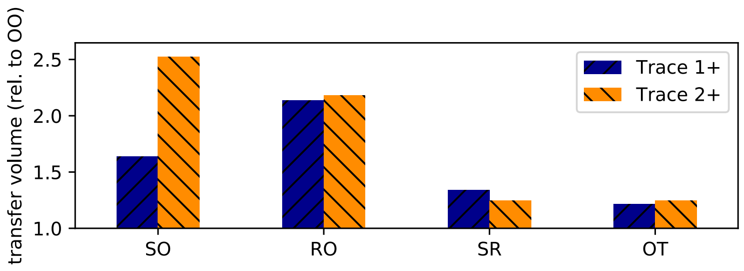


Figure 4.5: Transfer volume for all static threshold strategies normalized to the OO (optimal offline) strategy.

the partition size, i.e., 100k rows. For these, shipping the query results is the “right” strategy, while 40% have a data transfer of more than the partition size. For those, replicating the partitions is the “right” strategy.

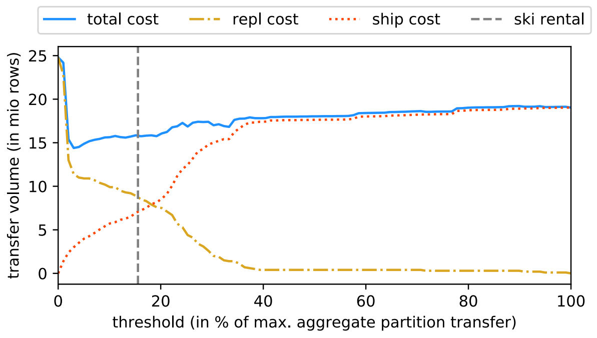
To highlight that this does not only apply to the full trace but also for any sub-sequence of the query trace, we also include the distribution for the first half of the two traces (labeled Trace X+ (half)). All traces show a significant skew. We also see a distribution shift, in the sense that the skew decreases slightly for Trace 2+ and a bit more for Trace 1+ as the trace progresses. Possible explanations are that these are traces from an ERP system and not an IoT application and, thus, still may contain some regular access patterns.

4.4 Evaluation

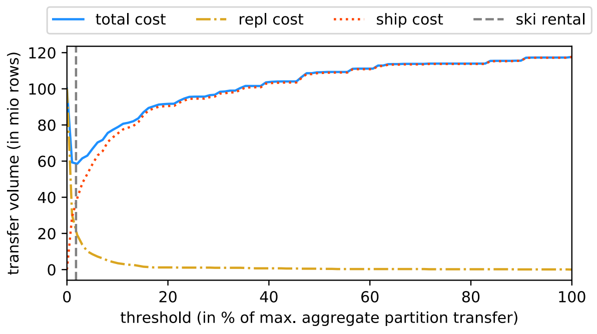
Next, we use the traces to do a *what-if* evaluation of the proposed strategies for our scenario with one back-end (incl. data store) and one front-end. For every strategy, we compute the transfer volume that would have been generated

Table 4.3: Strategy overview.

Short	Strategy	Description
RO	replicate only	Replicate immediately
SO	shipping only	Never replicate
SR	ski rental	Threshold strategy where $\tau = size(p)$
OT	optimal thresh.	Optimal offline threshold
RT	reactive thresh.	Choose threshold that would have worked best for past queries
OO	optimal offline	Lower bound



(a) Trace 1+.



(b) Trace 2+.

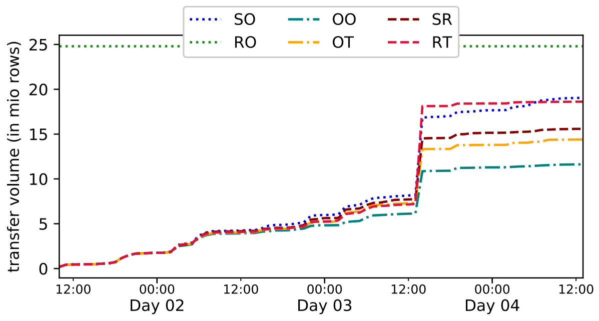
Figure 4.6: Transfer volume [in rows] for the threshold strategy for all possible thresholds (normalized by maximum aggregate partition transfer volume). In addition, we highlight the threshold of the SR strategy by a dashed line. Note, a threshold of 0 corresponds to RO, while a threshold of 100 corresponds to SO.

between the front-end and the back-end, if this strategy had been used. For each trace and each strategy, we compute the total cost for the whole trace duration. Recall Table 4.3 summarizes the strategies.

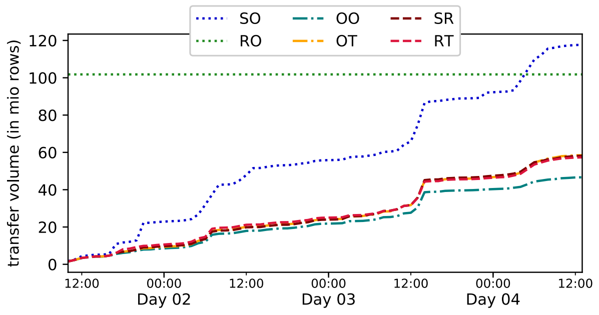
To start, we consider static strategies, i.e., those that use a fixed parameter. These include the naïve strategies ship-only (SO) and replicate-only (RO) as well as the ski-rental (SR) and the optimal threshold strategy (OT). Figure 4.5 shows the results normalized by the cost of the optimal offline strategy (OO). A lower bar corresponds to a smaller transfer volume. Note, all strategies are within a factor of 2.5 of optimal. However, the strategies that combine shipping and replication–SR (ski-rental) and the OT (optimal threshold) strategy–perform much better for both traces. Their overhead vs. the optimal offline algorithm is less than 34%. Indeed, the SR strategy already saves 18% (Trace 1+) and 51% (Trace 2+) compared to SO and saves 37% (Trace 1+) and 43% (Trace 2+) compared to RO. This confirms our intuition regarding exploring ski rental-based strategies. OT further improves upon the SR strategy by about 10% (Trace 1+) and 1% (Trace 2+).

To further explore threshold strategies, we next look at all possible threshold choices in Figure 4.6[1]. Here, the x-axis is the threshold normalized by the maximum aggregate partition transfer volume for each trace, as depicted in Figure 4.4. The transfer volume (“total-cost”) is the sum of the cost of shipping the query results (“ship-cost”) and the cost of replication (“repl-cost”). Note, as the threshold increases the shipping cost increases monotonically while the replication cost decreases monotonically. However, since they are not convex (concave) their sum, the replication cost can have multiple local minima. This plot again confirms our intuition that combining replication and shipping is beneficial (local minima exist) and that the naïve strategies RO or SO are sub-optimal.

[1]RO/SO correspond to using a threshold of zero/infinity resp.



(a) Trace 1+



(b) Trace 2+

Figure 4.7: Transfer volume [in rows] over time for all strategies.

Next, we evaluate how the performance of the strategies changes as the trace length increases, see Figure 4.7. For both traces, the performance of strategies except the naïve ones (RO and SO) are very similar for the first half of the traces. Indeed, up to this point in time, the performance penalty of not knowing the future (compared to the optimum offline) is less than 20%. During the second half of the trace, the access patterns change, which causes the RT strategy to send 57% (Trace 1+) and 28% (Trace 2+) more than the optimum. We suspect that the difference is caused by the distribution shift, as described in Section 4.3.

We propose the RT strategy to improve the OT/SR strategies. However, RT assumes that the access distribution can be approximated from the past. If a distribution shift occurs, this may no longer be the case. Thus, given the shift, it is not surprising that RT does not perform as well as OT and may even be worse than SR. However, we believe that better ways of estimating the distribution (e.g., by using a limited time window) and bounding the threshold should yield better performance. We are currently in the process of further evaluating such alternative strategies.

4.5 Related Work

In this chapter, we discuss the adaptive replication of a partitioned data store for a dynamic workload. This work bears similarities with the file allocation problem (FAP) and the database allocation problem (DAP).

In the file allocation problem, a file and its copies must be allocated in a network of computers with the goal of either minimizing the cost or maximizing the performance for a workload of file reads and writes. This problem has been shown to be NP-complete by Eswaran [58]. Solutions of the FAP can be classified by their assumption of the workload. Most of the earlier work on the FAP assumed a static workload [56] or a dynamic workload, which is known in advance [69]. The first solution for a dynamic workload, which is unknown in advance, was presented by Wolfson et al. [158] in 1992. The problem was treated from the perspective of competitive analysis in [21]. Bartal et al. split the problem into separate allocation (placement of a fixed set of file copies), migration (movement of existing file copies), and replication (addition and deletion of additional file copies) problems [21]. Most research on the FAP assumed that each file could be allocated, migrated, and replicated independently. In our work, we instead use the correlation between access of different partitions to inform the replication decision.

The database allocation problem (DAP) defined the same problem for database partitions (or "table fragments"). Different to files, dependencies between database partitions have to be considered due to integrity and consistency constraints and for performance reasons (e.g., joins) [8, 120]. Therefore, solutions to the DAP often include mechanisms to re-partition the data [78] or to estimate the template of future queries [74]. In our work, we assume that partitions function as the smallest unit in the storage organization that can be replicated without incurring a prohibitive overhead. Further, we believe that partitions, which are co-accessed, have similar access statistics and are therefore replicated soon after each other.

Our solution is based on earlier work on the ski rental problem [92, 91, 65, 68, 93], i.e., the problem of deciding at what point in time one should switch from renting to buying a ski set. This and similar work on the FAP [25, 12, 39] are based on competitive analysis [141], a worst-case analysis for online algorithms. We have transferred the problem from independent objects, ski-sets, to partitions of a data store and from the worst-case to an average-case analysis.

To our knowledge, no commercial database includes an adaptive mechanism for data replication today. Even though modern commercial databases manage far more data than their predecessors, they use replication mostly to guarantee availability rather than improving performance and mostly rely on manual or static replication schemes. Often, database administrators must either manually specify the replication factor [53] or trigger the creation of new replicas [48, 79]. One exception is BigTable [40], which leaves the decision to the application. The strategies that we have proposed in this chapter can help to build a self-tuning database [42, 41, 94] that uses replication to improve its performance. We see a strong need for such databases for future Internet of Things deployments.

4.6 Summary

In this work, we study selective data replication as a mechanism to minimize the data transfer volume between pairs of front-ends and back-ends. We study the cost trade-off between shipping query results versus replicating data partitions. To this end, we introduce two online replication strategies that decide when to replicate partitions. The former is based on the competitive ski rental algorithm and replicates any partition that incurs a transfer cost in excess of the replication cost. The second strategy estimates a threshold over all data accesses. By applying our proposed strategies to two real-life datasets, we show that they can yield significant transfer cost reduction compared to the baseline strategies (exclusively relying on shipping or replication). The ski rental strategy proved to be superior and reduced the transfer cost by between 18% to 51%.

5 Online Replication Strategies for Storage-Constrained Environments

In the last chapter, our model assumed that partitions are, once created, indefinitely accessible. However, there are many scenarios where limited storage space and continuous data generation necessitate that old data partitions are removed or archived. In these scenarios, partitions are only accessible for a limited time window before they are replaced by data partitions that are more recently generated. Figure 5.1 depicts a model that covers these situations.

Naturally, the accesses occurring over each time window can be considered separately, and these separate problem instances would then be very similar to the model discussed in the last chapter. In contrast, there may be patterns occurring in the access of partitions in a one-time window that can be used to make a replication decision in the next time window. In the simplest case, all data produced by one industrial process may be highly popular or the opposite. This would simplify the replication strategy. The eventual removal of partitions from the system also means that the likelihood of replicating partitions should decrease as the end of the time window is getting closer. In this chapter, we study the performance of the previously introduced online replication strategies over this new model. We also add new strategies that exploit patterns from previous time windows, to make replication decisions. The success of these strategies depends largely on the generalizability of these patterns.

As part of our new online replication strategies, we propose a Machine Learning (ML) based strategy. Previous work has shown the potential of Machine Learning for efficiently replicating data, e.g., in the case of Content Delivery Network caching [24]. We compare this strategy against our previous competitive ski-rental strategy and a number of strategies based on heuristics. Finally, we investigate a hybrid strategy that combines both the ML and ski-rental strategies.

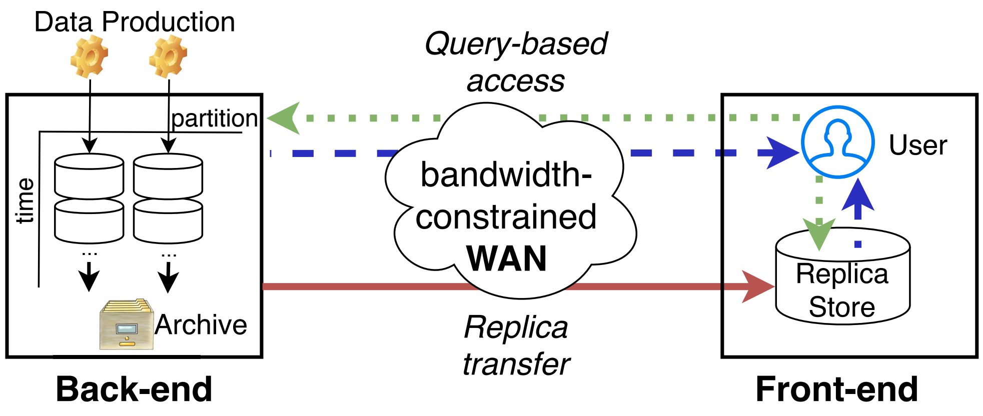


Figure 5.1: Transfers between back-ends and front-ends including a partition life-cycle.

Again, we use the term front-end to refer to the issuer of the query independent of whether it is the cloud or a different edge node. We use the term back-end to refer to the edge node functioning as a data store. We compare the performance of both strategies, namely the ski-rental and the ML-based ones, against naïve strategies, i.e., replicate all and replicate nothing, as well as the optimal offline strategy.

Our Contributions

- We extend our previous model to encompass storage-constrained environments.
- We develop online replication strategies that exploit patterns from previous time windows to decide the replication of partition.
- Our evaluation on real world database traces show that a competitive strategy based on the ski-rental algorithm can achieve a reduction in data transfer volume by 22% on average.
- We show that a hybrid strategy that combines both competitive and machine learning strategies can increase the reduction by an additional 3%.

5.1 Dataset

To evaluate our proposed online replication strategies, we use a set of Enterprise resource planning (ERP) database traces of a Global 2000 company [27]. After giving an overview of the traces, we detail how we cater it to our use case.

5.1.1 Raw Traces

The traces, see [27], record queries to various tables of an ERP database of a Global 2000 company. For each table and each query, the time of execution, as well as the accessed rows, are recorded. The traces span less than three days. We focus on production queries and, hence, exclude backup transfers taking place around 1 am each day. We focus on the two tables that were accessed most frequently and, hence, obtain two (sub-)traces. The first trace contains roughly 100 Million rows and records about 2.5 Million queries. The second trace has 24 Million rows and roughly 1.3 Million queries. Table 5.1 gives an overview of both traces.

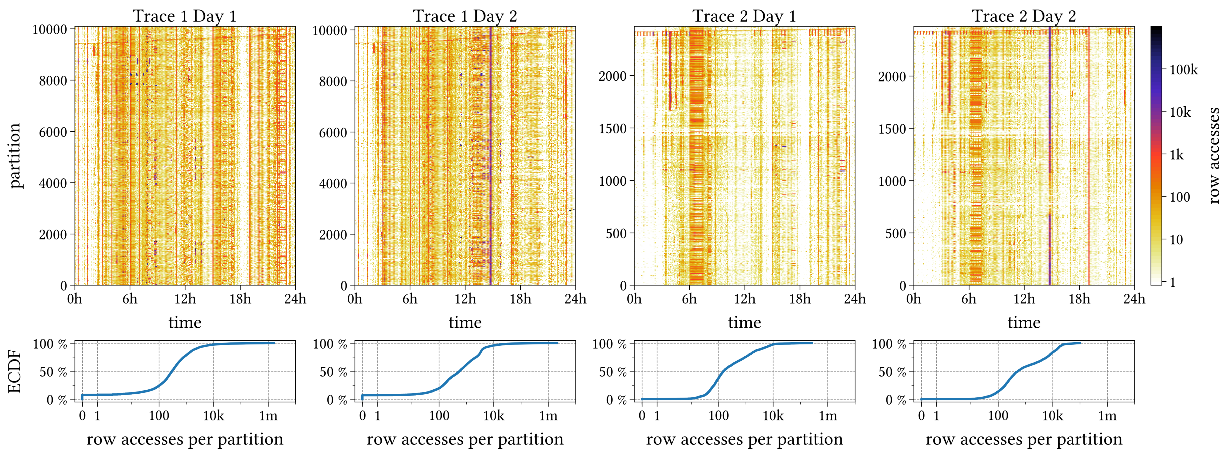


Figure 5.2: Trace Visualization. Top: row accesses aggregated over 500s intervals and 20 (trace 1) or 5 partitions (trace 2). Bottom: ECDF of cumulative row accesses per partition. Note the logarithmic x-axes.

5.1.2 Data Cleaning

In the following, we detail our modifications to the raw data to cater them to our use case.

- For performance reasons, the original trace only contains queries for the first 2 minutes for each 10 minute interval. To remove these gaps, we scale the captured queries to the full 10 minute interval.
- We introduce data partitions by aggregating 10k adjacent rows into a single partition[1], yielding roughly 10k and 2.4k partitions, respectively.
- We focus on two (full) consecutive days and aggregate accesses to partitions in 100-second intervals, yielding 864 data points per partition, trace, and day.

5.1.3 Time Windows

For our analysis, we focus on time windows of one day, which results in two time windows per trace. Figure 5.2 depicts the richness of our datasets. Clearly, the accesses recorded in both traces vary over time and are (temporally) correlated both within and across time windows (days): Periods with a large number of accesses are interleaved with periods of few to no accesses. Data locality correlations between adjacent partitions can also be observed: For example, in trace 2 for partitions 1720 through 2440 around the 5h mark. Additionally, we note that row accesses per partition are highly skewed. For roughly 75% of the partitions, less than 1k row accesses are recorded, while for more than 1% of the partitions, far more than 10k accesses are recorded. Notably, these heavy-hitters are observed at different times, and they can change from day to day: The common peaks at 15h on day 2 for trace 2 do not exist in the previous day. Thus, the training process of learning from previous windows may be challenging.

5.2 Replication Strategies: Challenges and Opportunities

Next, we introduce the general system model, introduce and discuss naïve replication schemes, and analyze potential cost reductions.

[1]Partition sizes between 1k [148] and 1,000k [98] are common.

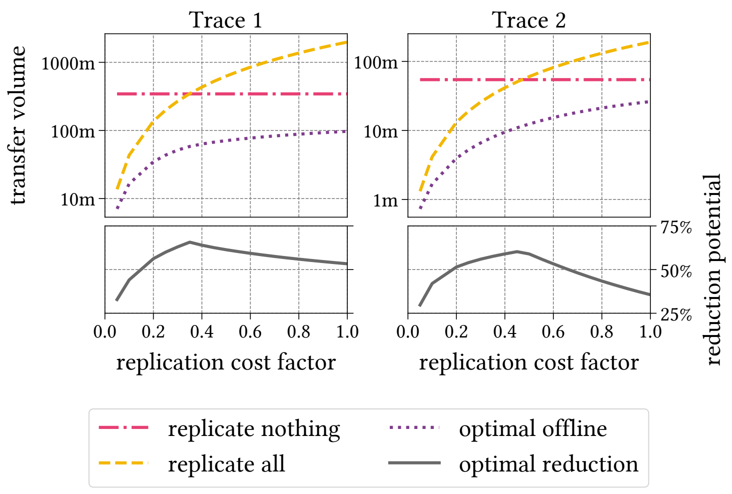


Figure 5.3: Costs of the naïve and optimal offline strategies and comparison of naïve baseline to the optimum.

5.2.1 Setting

As discussed in Section 5, we assume that data is stored locally at the back-end while queries are processed at the front-end. The main challenge is to decide which partitions to replicate to the front-end and which data partitions to keep at the back-end. In this chapter, our primary concern is the reduction of the data transfer volume. We leave the inclusion of secondary cost factors, e.g., storage and processing cost, to future work.

While replicating a partition incurs a transfer cost proportional to the size of the partition (in our case 10k rows, cf. Section 5.1), queries to non-replicated partitions are processed at the back-end and incur transfer costs proportional to the number of accessed rows. We refer to the former cost type as the *replication cost* and the latter type as the *transfer cost*. We furthermore denote by *partition cost* the cumulative costs for serving all queries of a partition from the back-end, i.e., the cumulative shipping cost. Clearly, replicating a partition to the front-end only yields a benefit if the *remaining partition cost* is at least as high as the replication cost itself.

5.2.2 Naïve Replication Strategies

A replication strategy is an algorithm that decides whether and when to replicate each partition. Its cost is the total transfer costs incurred within a time window across all partitions. The naïve replication strategies are either: *replicate nothing* or *replicate all*. These strategies do not differentiate between individual partitions. Hence, their performance depends on the ratio of partitions whose partition cost exceeds their replication cost. This may not only depend on the raw access volume but also system assumptions. For example, shipping individual rows is likely to

Table 5.1: Trace statistics for the two largest tables [27].

Name	Trace 1	Trace 2
Table size in rows [million]	100	24
Number queries [million]	2.49	1.28
Duration [days]	≈ 3	≈ 3
Accesses in rows [million]	137	34
Avg. rows per query	55	26

incur a higher overhead than replicating a partition as a whole. Furthermore, data partitions can often be compressed significantly [98] such that the replication cost may be a fraction of the cost of transferring all rows individually. We refer to this ratio of replication cost to transfer cost as the *replication cost factor*, whereby a factor of, e.g., 0.5 implies that replicating a partition only incurs half the costs of transferring all rows individually. To broaden the scope of our analysis, we study 10 replication cost factors, namely $\{0.1, 0.2, \ldots, 1.0\}$, albeit assuming the same replication cost factor for all partitions.

The optimal replication strategy is to replicate an individual partition only if its remaining partition cost is higher than its replication cost. Note, this optimal strategy can only be computed if all accesses are known in advance. Hence, we refer to it as the *optimal offline* strategy. While its performance is unattainable for all practical purposes, it serves as a lower bound for other replication strategies and quantifies the replication cost reduction potential.

Figure 5.3 (top) depicts the cost of both naïve strategies and the optimal offline strategy for both traces. For smaller replication cost factors, the replicate all strategy yields the best results, while for larger ones, the replicate nothing does. Note, both are far from optimal. Referring to the minimum cost over both strategies as the naïve baseline, the optimality gap of this naïve baseline ranges from 30% to 66% (cf. bottom of Figure 5.3). Moreover, a single strategy may not always yield the best results: For a replication cost factor of 0.4, neither the replicate all nor the replicate nothing strategy consistently yield the best performance.

5.3 Online Replication Strategies

Replication strategies can decide at any point in time to replicate a particular partition. Moreover, as future accesses and pattern shifts are often not known, these strategies are inherently online. In the following, we introduce several strategies, including some that offer *competitiveness* guarantees. A replication strategy is called c-competitive if for any sequence of time windows and any access pattern, the strategy's cost is at most c-times that of the cost of the optimal (offline) strategy. Next, we first discuss our competitive strategies and then propose several heuristics, including machine learning-based ones. See Table 5.2 for a summary of all strategies studied in this chapter.

5.3.1 Competitive Strategies

In Chapter 4, we introduce the *ski-rental* problem and its solution developed by Karlin et al. [92, 91]. Karlin et al. proved that the best 2-competitive strategy is to buy the skis once the cumulative daily rental fees would exceed the price of purchase.

This *ski-rental* strategy is applicable in the context of replication schemes (cf. [139]): once the cumulative transfer costs exceed the replication cost, the partition is replicated. Clearly, this strategy is also 2-competitive, and the result by Karlin et al. [92] also implies that no c-competitive strategy can exist for $c < 2$. However, empirically tuning the *threshold* which is used to decide when a partition is replicated might be beneficial in practice. In particular, one may replicate a partition once the transfer costs exceed t-times the replication cost. Notably, for any constant $t > 0$, the respective strategy is $\max\{1 + f/t, 1 + t/f\}$-competitive, where f denotes the replication cost factor.

Harnessing information on the previous time window, we propose the *last-threshold* strategy: (i) for the previous time window the *optimal* threshold t is computed, and (ii) for the current time window the t-threshold strategy is executed.

5.3.2 Heuristic Strategies

Next, we propose several heuristics, i.e., strategies not providing performance guarantees.

Last-Partition Strategy Given the accesses from the previous time window, one can compute a posteriori optimal replication decisions. Accordingly, a simple strategy is to immediately replicate partitions, which should have been replicated using the threshold from the previous time window. Interestingly, if access patterns are invariant over time, this strategy is *optimal*.

Machine Learning Strategies All of the above-discussed approaches handle partitions equally: upon meeting a certain common criterion, replication is performed. This is not necessary, and may give rise to improvements. To motivate *fine-granular* machine learning algorithms, consider the following (cf. [94]). First, access patterns are (highly) correlated across both: the temporal and the data-locality dimension (cf. Section 5.1), and the above approaches are agnostic to this *seasonality*. Second, the above approaches are agnostic to common (sub-)patterns shared by heavy hitters and which may hence be harnessed to perform replications early on. Third, learning-based approaches may be robust towards linear transformations of access patterns recorded in previous windows, e.g., due to increased demand.

To evaluate the potential of *learning* which partitions to replicate, we cast the problem as a *classification* problem. Specifically, we view partition accesses as time series and create for each point a feature vector together with the (a posteriori known) classification decision whether the replication *at this point* would have been beneficial. We generate features by using a set of well-studied aggregation methods (e.g., sum, mean, max, etc.) over varying numbers of preceding points.

To perform the actual classification, we propose to use random forest classifiers [30] because these scale well, even for large training sets, are implemented in several frameworks, and allow for human interpretation. Specifically, we propose to train the classifier over the preceding time window and, then, apply the obtained classification model to the current time window. The classifier then returns a *classification probability* in the range $[0,1]$. We (by default)

Table 5.2: Summary of Replication Strategies.

		Strategy	**Description**
naïve	**heuristic**	replicate all	replicates partitions immediately when a new time window starts
		replicate nothing	always answers queries directly
online		last-partition	replicates partitions that previously exceeded its replication cost
		classifier	uses random forest classifier trained on previous time window
	competitive	hybrid	replicates if ski-rental or the classifier strategy would do so
		ski-rental	replicates once replication cost has been exceeded (threshold=1)
		last-threshold	sets ski-rental threshold to optimal one of previous window
		optimal offline	uses knowledge about future to inform replication decisions

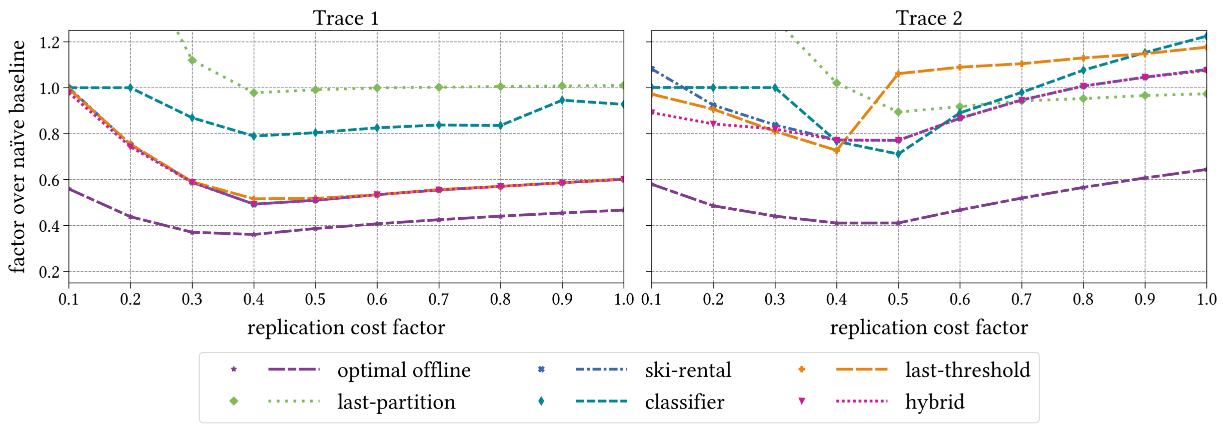


Figure 5.4: Normalized performance of strategies.

interpret it as a binary replication decision by checking whether the probability lies above 0.5. Besides deciding on a classification probability threshold, various classification model parameters need to be calibrated. We defer this discussion to the evaluation.

Besides the above classifier strategy, we also propose to bridge machine learning and competitive strategies to preserve performance guarantees (to some extent) while harnessing *fine-granular* trace histories. In particular, we propose the *hybrid* strategy that executes both the ski-rental and the classifier strategy: if either strategy decides to replicate a partition, the hybrid strategy also performs the replication. Intuitively, by using the disjunction of both, we aim at replicating heavy-hitters early on while preserving the 2-competitiveness for partitions whose transfer cost eventually exceeds the replication cost. We note that *calibrating* even such simple classification models requires substantial effort and defer implementation details to the evaluation section (cf. Section 5.4.1).

5.4 Evaluation

We now evaluate the performance of the diverse replication strategies (cf. Table 5.2) on the dataset introduced in Section 5.1. Before reporting on the results, we give some details on the calibration of the machine learning-based strategies.

5.4.1 Classifier Calibration

It is challenging to accurately calibrate classifiers, without over-training them. First, due to the skewed partition cost distribution, partitions that should not be replicated dominate. We adjust the model to this imbalance by weighting feature vectors according to their class frequency. Second, feature vectors differ in importance: correctly recognizing partitions whose partition cost is either very low or very high is of vital importance. Hence, we again adjust the weights accordingly. Third, to further increase the accuracy of the classifier, we combine our random forest classifiers with an isotonic regression model [113].

Being interested in the off-the-shelf performance, we employ the default classification probability threshold of 0.5 for the basic classifier strategy. For the hybrid strategy, we increase this threshold to 0.8 to minimize incorrect replication decisions but also discuss various other thresholds below.

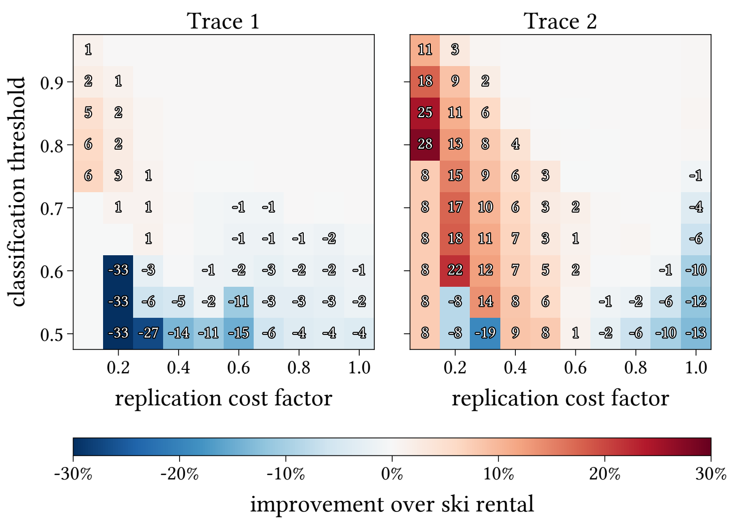


Figure 5.5: Sensitivity analysis of the hybrid strategy's performance relative to ski-rental's performance.

5.4.2 Results

Figure 5.4 shows the performance of the strategies relative to the naïve baseline for the 2nd day for both traces. A number lower/higher than one reflects a reduction/increase in the relative total cost. Notably, most strategies reduce the costs for a wide range of replication cost factors. The last-partition strategy is an exception: its potential reductions are outweighed by its additional costs for replication factors below 0.4. The competitive ski-rental strategy overall outperforms the basic classifier approach (especially on trace 1), while the last-partition's performance is less consistent for trace 2. From the "pure" strategies, ski-rental achieves the largest reduction with an average performance improvement of 22% and a maximum performance of 50%. Over all strategies, the hybrid strategy performs best with an average performance of 25% and a maximum performance of 51%. It significantly improves upon the standalone performance for both the ski-rental and the classifier strategies in trace 2. Specifically, the hybrid strategy improves upon ski-rental's cost by 28% for a replication factor of 0.1 while yielding the same performance as ski-rental for a replication factor of 1.0 even though the classifier alone performs 13% worse. Overall, the hybrid strategy improves the ski-rental approach by 3% on average.

5.4.3 Hybrid Strategy: In-Depth Analysis

The hybrid strategy yields the best performance when manually selecting a probability classification threshold of 0.8. While this *a posteriori* choice highlights the potential benefits of such hybrid strategies, it also highlights the challenges of robustly calibrating machine learning models. Figure 5.5 depicts the performance as a function of the classification threshold over both traces. While choosing a threshold greater than or equal to 0.75 improves performance, choosing a replication threshold of 0.6 yields improvements for trace 2 while consistently worsening performance for trace 1.

To gain insights into the effectiveness of the hybrid strategy, consider Figure 5.6. It depicts the precision[2], recall[3], and the mean replication time for the hybrid strategy and its sub-strategies. As ski-rental replicates any partition that exceeds the replication cost, its recall is 1.0. This carries over to the hybrid strategies, which, thus, also have

[2]The number of partitions being correctly identified to be replicated over the total number of replicated partitions.
[3]The number of partitions being correctly identified to be replicated over the total number of partitions that were to be replicated.

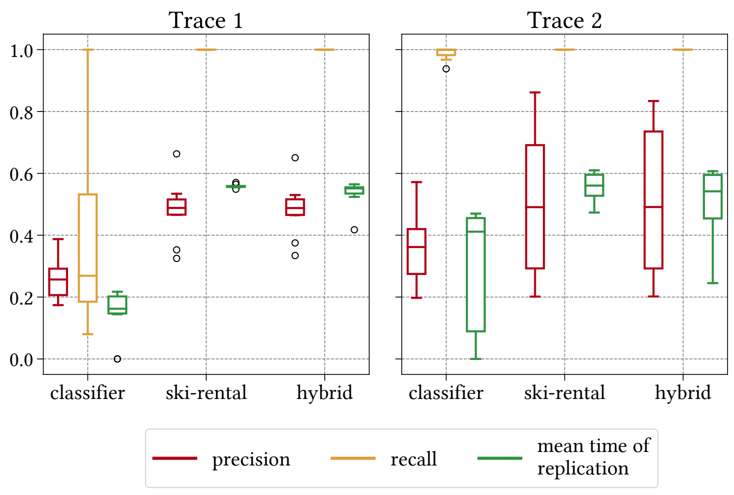


Figure 5.6: Analysis of hybrid strategy improvement for each sub-strategy.

a recall of 1.0. However, by using the classifier, the time at which partitions are replicated decreases significantly, thereby saving transfer costs. Furthermore, the classifier's slightly worse precision only marginally reduces the hybrid's precision. Therefore, it does not add excessive replication cost. This explains the improvement of hybrid over ski-rental.

5.5 Summary

As the traffic at the network edge continues to grow at an unprecedented pace, it is imperative to decide which data should be processed in-situ at the edge and which data should be forwarded to the cloud. In this work, we observe that such decisions have to be reactive to volatile accesses. We study two families of online algorithms, namely, competitive (ski-rental) and machine learning algorithms, to inform such decisions at the edge of the network. These algorithms proactively decide which data will be replicated to the remote cloud based on the recent access activity.

Our results show that ski-rental not only yields significant cost reductions (22% on average and up to 50% at most) compared to naïve strategies but is also easy to use at the edge even when resources may be limited. Moreover, the best online strategy may depend on the scenario. To address this, we introduce a hybrid strategy that combines the advantages of both families of strategies. It improves ski-rental by 3%.

6 Distributed Top-k

In this chapter, we develop a data-driven transfer optimization for distributed top-*k* operations. Distributed top-*k* is used to estimate the *k* most relevant objects from information distributed over multiple nodes. In the example depicted by Figure 6.1, an engineer uses distributed top-*k* to collect the most frequently occurring events among a set of autonomously driving vehicles, each an edge node in its regard. Each local dataset is structured in the form of top-lists. A central coordinator requests data from all nodes and aggregates the results into a single estimate. Other applications for distributed top-*k* include the use for server monitoring [33], for image processing [19], or in relational databases [31]. The result of distributed top-*k* can be used to inform other analytics, e.g., anomaly identification.

While distributed top-*k* is immensely useful for various use cases, it can produce large data transfers. If the number of objects is huge, then naïvely transferring all the data will create a large overhead. In contrast, if the coordinator transfers data iteratively to avoid the transfer of irrelevant data, then the number of round-trips drastically inflates the operation's run-time. Related work [33, 63] has recognized this problem and has proposed multi-phase algorithms. In every phase, the coordinator requests a subset of the data from all nodes. The coordinator aggregates the data and then decides how much data it will request in the next phase. Algorithms such as TPUT [33] guarantee a correct top-*k* estimate in three phases.

We use two insights to reduce the number of phases while keeping the data transfer volume low. First, previous empirical work has shown that many top lists follow the power-law distribution [112], meaning that the top objects (e.g., events) have drastically higher scores (e.g., frequency of occurrence in a time window) than objects lower in the list. This data skew indicates that collecting few items from each node can be enough to estimate the top-*k* correctly. Second, users of top-*k* operations often have a tolerance for a small error (i.e., that two objects are shown in reverse order) in their result, as top-*k* operations are often executed in highly volatile environments, where summaries are quickly outdated, and some risk of error is always present.

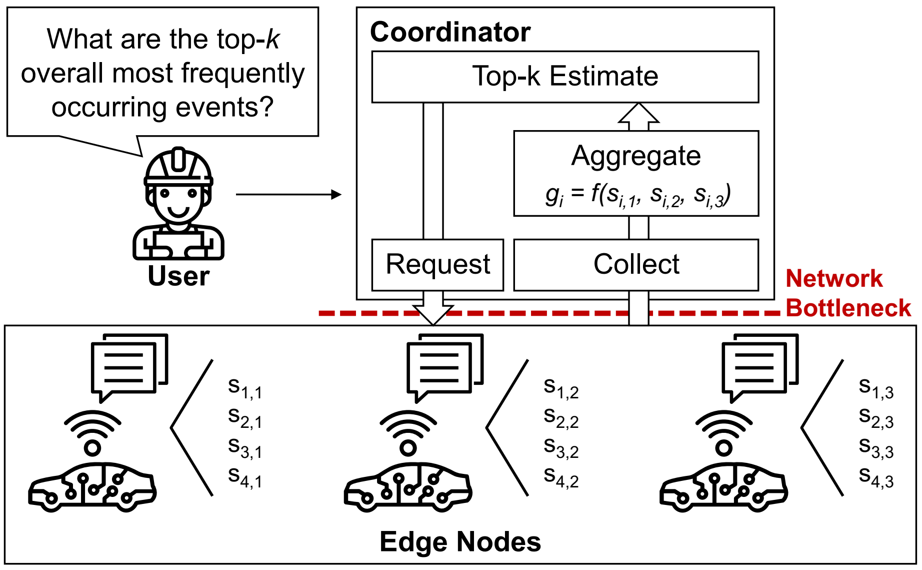

What are the top-k overall most frequently occurring events?
User
Coordinator
Top-k Estimate
Aggregate
$g_i = f(s_{i,1}, s_{i,2}, s_{i,3})$
Request
Collect
Network Bottleneck
$s_{1,1}$ $s_{2,1}$ $s_{3,1}$ $s_{4,1}$
$s_{1,2}$ $s_{2,2}$ $s_{3,2}$ $s_{4,2}$
$s_{1,3}$ $s_{2,3}$ $s_{3,3}$ $s_{4,3}$
Edge Nodes

Figure 6.1: The figure depicts a coordinator that creates a top-k estimate by requesting data from a set of nodes. We use the notation $s_{i,j}$ and g_i to refer to the node-local and the globally-aggregated scores, respectively.

To incorporate both insights, we created two parametrized algorithms of the existing, approximate, two-phase algorithm XPUT. The two algorithms 1PA and 2PA use a single phase and two phases, respectively. Then, we created a variant of each algorithm (1PA/wc & 2PA/wc) that is guaranteed to be correct in the worst-case as long as the aggregated scores follow a power-law distribution. Finally, we created another variant (1PA/avg & 2PA/avg) for the average-case and with the inclusion of a small error. For a real-world dataset, the 2PA reduced the transfer volume by 13% in comparison to the state-of-the-art algorithm TPUT.

Our Contributions

- We propose the one-phase algorithm 1PA and the two-phase algorithm 2PA that are parametrized variants of the existing XPUT distributed top-k algorithm.
- We identify the worst-case scenario for both algorithms in relation to the distribution parameter and number of objects. Based on the scenario, we define upper bound parameter choices that guarantee correctness.
- We discuss different average-case scenarios. Each scenario assumes that the global scores follow a power-law distribution and are split across the server using a probability distribution.
- We introduce the rank error term that represents the degree to which the order of objects in the estimate deviates from their true order.
- Based on the rank error, we define approximate variants of our algorithms. At the cost of a slight error, the approximate variant of 2PA reduces both the data transfer volume and the number of phases by 13% and 1 phase (compared to 3-phase algorithms) or 5% and 2 phases, respectively (compared to 4-phase algorithms).

6.1 Problem Definition

We consider a distributed environment with *nodes* (e.g., individual vehicles) holding data on *objects* (e.g., events). On each node, each object is associated with a *partial score* (e.g., the frequency of the encountered event) per node, and all object partial score pairs are stored in a list, which is sorted in descending order. We refer to the pair of an object and associated partial score as an *item*. The sum of all partial scores of an object constitutes its *global score* (e.g. the frequency by which an event appears among all vehicles). The k objects with the highest global scores are the *true top-k*. Items are *collected* or *observed* and aggregated by a central *coordinator* to construct a *top-k estimate*. In this chapter, we focus on *top-k algorithms* that transfer items in non-overlapping phases. Every phase starts with bulk transfers from all nodes to the coordinator and ends with a re-computation of the top-k estimate.

The performance of a top-k algorithm can be characterized by three factors, its transfer cost, its runtime and its accuracy. The transfer cost is determined by the number of items it collects. The runtime is heavily influenced by the number of round-trips between the coordinator and nodes, which can be reduced by bulk transfers. The accuracy is determined by the distance between the true top-k and the top-k estimate in terms of an error metric. We distinguish between three error metrics: *set error* – the fraction of objects in the estimate that are not in the true top-k, *rank error* – the degree to which the estimate contains either the wrong objects or the right objects at the wrong position, and the *score error* – the difference between the estimated and the true global scores on average. The set error has the disadvantage that it cannot distinguish between an estimate that has the true top-1 at the first position from an estimate that has it at the k-th position. The score error is useful when users need the scores of the top-k estimate to be correct. Yet, for many use cases, it is more important that the rank of the item is correct and that the score is only in the right ballpark. For these reasons, we focus on the rank error (see Subsection 6.6.2).

6.2 Approach

This section defines requests and mechanisms used by most multi-phased distributed top-k items. We then construct two algorithms using these. We use these algorithms in the remainder of this chapter.

6.2.1 Requests

Multi-phase top-k algorithms such as TPUT [33] and DTA [104] use three basic requests to collect data in bulk from nodes. We refer to them as *by-position*, *by-threshold*, and *by-object*. By-position requests collect data sequentially up to a position in the node's list. These are usually used in the first phase to get an initial estimate of the top-k. In the next round, data can then be collected by a computed threshold, and by-threshold requests or partial scores can be collected from nodes where they were previously not observed with by-object requests. By-threshold request By-threshold requests collect all partial scores with scores larger than a given threshold. Different to the by-position requests, these requests can transfer different numbers of items per node depending on the local score distribution. By-object requests collect partial scores for a set of objects from nodes where they were previously not observed. Apart from these requests, additional exchanges have been introduced that collect statistics on the score distribution to estimate the score of items (see [104]).

6.2.2 Mechanisms

Most top-k algorithms make use of the same core mechanisms to compute the top-k efficiently. We use the example given in Figure 6.2 to describe them. In this example, a coordinator collects a set of objects by-position in the first phase. For any object that has been observed with at least one partial score (at the central coordinator) a *lower bound global score* and an *upper bound global score* are computed. These bounds limit the value the estimated global score can take on when more data is collected. The lower bound global score for one object is the sum of all its collected partial scores. For B, this is its actual global score 80, for A and C this only reflects the score from one

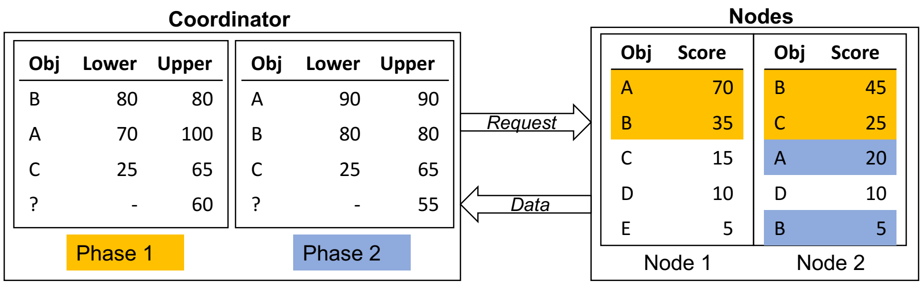


Figure 6.2: Example of a distributed Top-2 over two nodes in two phases. The coordinator collects data over two phases and computes both lower and upper bounds.

node. The top-k objects with the highest lower bound global scores represent the top-k estimate. In the example these are B, A. The upper bound global score is computed by summing the lower bound with the *high-value*. The high-value of a node is the smallest partial score for which all larger partial scores were also observed. In the example the high-values are 35 and 25 for each node, respectively, and so the upper bound global score becomes 100 and 65 for A and C. On top of the upper bound global score of all observed objects, the sum of the high-values gives us an upper bound global score of a possible unobserved object, in short *upper bound unobserved*. In the example, we use a question mark '?' to refer to this score, which is 60 in the first phase.

All of these values are re-computed after every phase and determine what data is collected in the next phase. The k-th lower bound global score has an important role for this purpose, as it separates objects that are potentially in the top-k from objects that definitely are not. It is referred to as the *bottom*. If the upper bound unobserved is higher than the bottom, then at least one object may exist that is in the true top-k and yet has not been observed so far. Hence, more items have to be collected with either a by-position or a by-threshold request. If this is not the case, then all objects in the true top-k have been observed from at least one node. The subset of objects with an upper bound greater-equal than the bottom are candidates for the top-k. So, for all of these objects, their missing scores need to be retrieved. As we in this chapter focus on the rank error, we can reduce this list by any object that has bounds that do not overlap with the bounds of any other object. In the example, the unobserved upper bound is lower than the bottom 70, the object C outside the estimate has an upper bound global score below the bottom, but the bounds of A and B overlap. Hence, we need to collect the items for only A and B, which is shown in the second phase. Using these mechanisms, we can be sure that no more items need to be collected. For more information on the use of the bounds discussed above, we refer to the work of Theobald et al. [153].

6.2.3 Algorithms

At this point, we define algorithms we use for the rest of the chapter. Our goal is to keep the run-time low by restricting the number of phases while restricting the data transfer volume to similar levels as observed in related work at the cost of a slightly reduced accuracy. For this purpose, we have constructed an one-phase algorithm (1PA) and a two-phase algorithm (2PA). Both are not radical new ideas. 1PA has been used as the first phase for most multi-phase algorithms, and 2PA is similar to the previously proposed algorithm XPUT [104]. It deviates from XPUT slightly in the second phase, where it only collects partial scores that could influence the rank error.

Algorithm 1 one-phase-algorithm(d, k)

```
1: data := collect(BY-POSITION, d)
2: items := sort-descending(aggregate(data))
3: return top(items, k)
```

1PA is shown in Algorithm 1. It collects from each node items up to a given depth *d*. The result is aggregated on the coordinator, and the k items with highest lower bound global scores are returned as estimate to the user.

2PA (see Algorithm 2) shares the first phase with 1PA, where it also transfers the top *d* items from each node. For the second phase, it defines candidates to be all objects with an upper bound greater-equal to the bottom (Line 5). It then refines the set of objects to only those objects which have a lower to upper bound interval that is overlapped with another object (Line 6). (This is the point where it deviates from XPUT.) For the refined candidates, it collects all missing scores, i.e., scores from nodes where the object was not observed before (Line 8). The additional partial scores are summed up with the existing lower bound global scores and used to produce a new estimate (Line 9).

The choice of the depth parameter *d* is for both algorithms the deciding factor for both the transfer cost and accuracy. For 1PA, a low value for *d* can translate into missing partial scores of objects in the true top-*k* and thus reduces the accuracy. A high value for *d*, on the other hand, produces a high transfer cost. For 2PA, the situation looks similar with the difference that 2PA can resolve all candidates in the second phase. Thus, as long as it observes all objects in the true top-*k* at least once, it is 100% accurate. However, a low value for *d* can lead to a large set of candidates and thus to a high transfer cost in the second round. A high value for *d*, in contrast, can mean that 2PA does no better than 1PA at the cost of an additional phase.

The remainder of this chapter studies the effect of the depth parameter and propose values to find a sweet spot between accuracy and transfer cost.

6.3 Related Work

The existing related work on the distributed top-*k* problem can be classified into four classes according to their method of data access: (1) Iterative algorithms, (2) Bulk iterative algorithms, (3) multi-phase algorithms, and (4) Algorithms that use statistical information.

The predominant work of the first class is work by Fagin et al. [61, 62], who developed algorithms that collect data one item from all nodes at a time. Fagin et al. designed multiple algorithms (Fagins' Algorithm – FA, Threshold Algorithm – TA, No Random Access – NRA, Combined Algorithm – CA) to account for different costs between sorted and random access on spinning disks. These algorithms were improved by Akbarinia et al. [2] in their work on the Best Position Algorithm (BPA) and Best Position Algorithm 2 (BPA2). These algorithms eliminated redundant transfers by keeping state on the node.

Algorithm 2 two-phase-algorithm(d, k)

```
 1: # Phase 1
 2: data := collect(BY-POSITION, d)
 3: items := sort-descending(aggregate(data))
 4: bottom := items[k]
 5: candidates := filter-by-upper-bound(items, bottom)
 6: candidates2 := filter-overlapping-bounds(candidates)
 7: # Phase 2
 8: data2 := collect(BY-OBJECT, candidates2)
 9: items2 := sort-descending(aggregate(data ∪ data2))
10: return top(items2, k)
```

The second class of algorithms includes the Distributed Threshold Algorithm (DTA) by Michel et al. [104] and the Distributed Bulk Best Position Algorithm (DBBPA) by Fang [64]. Both reduced the number of round-trips by transferring items in bulk.

The third class of algorithms limits the transfer of data to a small number of phases. The work by Cao et al. [33] is the primary example of this class. Their algorithm Three-Phase Uniform Threshold (TPUT) is shown to perform close to optimal when the node score distributions follow a log–log slope. Two close relatives are XPUT [104] by Michel et al., which is an approximate variant of TPUT with two instead of three phases, and 4RUT [63], which introduced an additional phase between the second and the first.

The fourth class of algorithms used summaries of the score distribution to estimate global scores at a reduced transfer cost. One prominent example of this class is work by Michel et al. [104], who used bloom filter histogram data structures to summarize the items. Based on this data structure, Michel et al. proposed the two and the three-phase algorithms Klee-3 and Klee-4, respectively.

We have summarized the discussed algorithms in Table 6.1. Our work is closest to Three-Phase Uniform Threshold algorithm developed by Cao et al. [33]. Our work is different in that we focus on approximate algorithms with one and two phases. We share with Cao et al. the focus on power-law distributions, but in contrast to their work we assume that the global score distribution not the node score distribution follows a power-law. Most of the algorithms listed here produce exact solutions. The only exception is work by Michel et al. [104]. Their approach with Klee-3 and Klee-4 trades lower transfer cost against more computation and a larger set of operations on the node-side. We work with the simple three requests outlined above and thus reduce computation on the node-side to a minimum.

In our work we focus on top-*k* queries on static data. We expect that the scores for each object do not change while a top-*k* operation is executed. This occurs frequently in practice when data is collected in time intervals. For a discussion of distributed top-*k* queries over streams please see work by Babcock et al. [17].

6.4 Power-Law and Distributed Top-k

Power-law distributions are well-known for being able to describe a wide range of phenomena both from the natural and the social world. Examples include city sizes, the number of telephone calls and the magnitude of earthquakes [112]. In visualizations, power-law distributions are easily identified for their appearance as straight line on log-log plots. See Figure 6.3 for examples of power-law distributions with varying parameters. Here we use the

Table 6.1: Algorithms from related work.

Name	Execution	Transfer	Correctness
FA [61]	iterative	row-wise	correct
TA, NRA, CA [62]	iterative	row-wise	correct
BPA, BPA2 [2]	iterative	bulk	correct
DBBPA [64]	iterative	bulk	correct
DTA [104]	iterative	bulk	correct
TPUT [33]	3 phases	bulk	correct
4RUT [63]	4 phases	bulk	correct
XPUT [104]	2 phases	bulk	heuristic
Klee-3 [104]	2 phases	bulk	heuristic
Klee-4 [104]	3 phases	bulk	heuristic

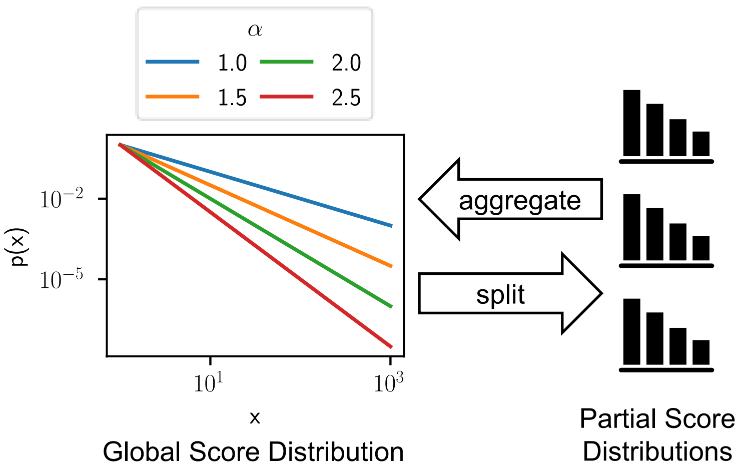


Figure 6.3: Global scores, scores aggregated over multiple nodes, often follow a power-law distribution. For our simulation we artificially split the global scores over multiple nodes.

power-law to describe the *global score distribution*. We follow the mathematical convention used by Newman et al. in [112]:

$$p(x) = Cx^{-\alpha}, \alpha \geq 1$$

The constant C is used to normalize the distribution to sum up to 1, while α is a parameter of the distribution that influences the skew of the distribution. As you can see in Figure 6.3, an increase in α increases the skew of the line leading to a quicker decrease in score for objects at lower positions.

While we generally assume that the global score distribution follows a power-law, there are different choices for the *node score distribution*, i.e., what distribution the scores at individual nodes follow. We describe the relationship between the global score distribution and the node score distribution as the *split*. The split defines how the global score of each object is distributed over all nodes in terms of one fraction per node. We refer to the fractions as *split values*. The split is the reverse operation of an aggregation, which is used in a real environment (see Figure 6.3).

We distinguish two types of splits: *random splits* and *adversarial splits*. Random splits are sampled from a random distribution. The split values for each object are then normalized to sum up to one. We use uniform, exponential and Pareto random distributions to produce increasingly skewed node score distributions. Adversarial splits are splits that are constructed with intent of maximizing the transfer cost of a given algorithm. We discuss these splits in more detail in the next section.

6.5 Worst-Case Parameter Choices

In this section we determine parameter choices for 1PA and 2PA that guarantee correctness even in the worst-case. To this end, we investigate the worst-case transfer cost of 1PA and 2PA for global score distributions that follow the power-law. The worst-case transfer cost for an algorithm is the highest number of items it has to transfer to create an estimate that is correct for any possible splits of global scores. We focus here on zero rank error as the correctness condition. Any algorithm violates this condition when its estimate excludes an object that is in the top-k (similar to the set error) or when an object at some position in the true top-k is placed at a different position in the estimate. We refer to these violations as *missed-object* and *displaced-object*, respectively.

For 1PA, the transfer cost is the depth parameter d multiplied with the number of nodes m: dm. Therefore, the worst-case transfer cost is the scenario where the depth parameter is maximized. To put it in another way, given the depth parameter d and a depth D_1, where the estimate is incorrect if $d < D_1$, then the split that produces the largest D_1 is the worst-case split for 1PA.

For 2PA, the transfer cost is the sum of dm with the number of scores that are collected in the second phase. Due to the second phase, the depth D_2 for which holds that 2PA produces an incorrect estimate if $d < D_2$ and a correct estimate if $d \geq D_2$ is lower-equal than the same factor for 1PA: $D_2 \leq D_1 + 1$ and is often substantially lower than D_1, as it can collect additional scores in the second phase. (The additional 1 occurs in scenarios, where the estimate is correct after the first phase, but some upper bounds of objects outside of the estimate are larger than the bottom. See Subsection 6.2.2 for details.) Similarly, the worst-case transfer cost of 2PA can be at most $(D_1 + 1)m$ for any split. To see this let us assume that the cost is higher than $(D_1 + 1)m$. As the depth chosen in the first phase is at most $D_1 + 1$ and so the cost is $(D_1 + 1)m$, then the additional cost must be incurred in the second phase. Yet, in that case 2PA could increase its depth d to a value higher than $D_1 + 1$ in the first phase to transfer these items. This contradicts our assumption and therefore proves that the transfer cost of 2PA is at most $(D_1 + 1)m$.

Coming back to the two correctness violations *missed-object* and *displaced-object*. Both violations require some partial scores to be "hidden" at depth D, where D is maximized in the worst-case. This means, that the global scores of objects outside of the true top-k are split in a way that their partial scores are at least as high as the to-be-hidden partial scores. To maximize the number of objects that can be used in this fashion, the to-be-hidden partial scores must be as small as possible. We describe the split that constructs the missed-object violation with maximized D as *hide-object* and respectively the split for the displaced-object violation as *hide-difference* split.

In the *hide-object* split, an object inside the true top-k has not been observed at any node up to a depth D. To produce the smallest partial scores, the global scores of the hide-object is split evenly among all nodes and all partial scores are located at the same depth D (at most $D+1$). If the object were hidden at some higher depth at one node, then a redistribution of scores would lead to an increase of D. To maximize the factor depth D, this split always hides the k-th object in the true top-k as it has the smallest global score in the true top-k.

To produce a displaced-object violation at least the difference between the object that is in the true top-k but displaced in the estimate and an object at a lower position is hidden at depth D. If less than the difference is hidden, then the first object is not displaced in the estimate. If more than the difference is hidden, then the partial score is higher than necessary. For this reason, we call the corresponding split the hide-difference split. The difference can be hidden at at most $m-1$ nodes, as the remaining score must be located at least on one node.

Of the two splits, the hide-difference split produces only a minimal transfer cost for 2PA, as any hidden difference can be collected in the second phase. Therefore, we only consider the hide-object split for this algorithm. For 1PA, both splits produce high transfer costs.

Table 6.2: Variables used in the description of the distributed top-k problem.

Name	Description
n	number of items
m	number of nodes
k	parameter of the top-k request
g_i	i-th largest global score
D	depth parameter that achieves a correct result

The rest of this section describes how both the hide-object and the hide-difference split are constructed and how depth can be computed when the number of items n, the number of nodes m and the distribution parameter α of the global score distribution are known.

6.5.1 Hide-Object

The *hide-object* split hides the top-k object in the true top-k by creating filler values from other objects' scores that are then placed at a lower position at the individual nodes. For example, if for a top-1 the global scores of the are $(a,10)$, $(b,7)$ and $(c,5)$, then for two nodes the items are distributed so that object a ends up on the bottom. For node one, that becomes $(b,5),(a,5),(c,0)$ and for node two $(c,5),(a,5),(b,2)$. In this way, the local top-2 items from each node have to be collected to construct the top-1. We now describe the procedure for constructing the hide-object split and then show the depth D as a function of the number of items n, nodes m and distribution parameter α.

Procedure. The hide-object split is constructed in four steps. (1) The scores of all true top-k objects are split evenly among all nodes. The k-th global score takes the value v at all nodes. (2) From the global scores of all other objects as many partial scores equal to the value v are created as possible. We call these values *fillers*. (3) The partial scores are then distributed in a round-robin fashion among all nodes. So that all nodes receive the same number of fillers (at most one more). (4) The order of scores is changed, so that the partial score of the k-th object are placed as far down as possible. All partial scores end up at the depth D or $D+1$.

Depth. To compute the value of depth we use the formalism defined in Table 6.2. The global score of the k-th object in the true top-k is: g_k. The partial score at each node is g_k/m. We can use objects from $k+1$ up to n to construct fillers. The depth D is then defined by the numbers of fillers divided by the number of nodes m summed up with k:

$$D = k + \lceil \frac{1}{m} \sum_{i=k+1}^{n} \lfloor m \frac{g_i}{g_k} \rfloor \rceil \tag{6.1}$$

Using the power law as global score distribution, we can replace occurrences of g_i:

$$D = k + \lceil \frac{1}{m} \sum_{i=k+1}^{n} \lfloor m \frac{k^\alpha}{i^\alpha} \rfloor \rceil \tag{6.2}$$

From here, we can compute the upper bound for D as:

Table 6.3: Maximum upper bound depth D (as factor of k) for the hide-object split and different choices of the distribution parameter α and number of items n.

n	$\alpha=1$	$\alpha=1.5$	$\alpha=2$	$\alpha=2.5$	$\alpha=3$
1000	8.5	3.5	2.6	2.3	2.2
10000	10.8	3.6	2.6	2.3	2.2
100000	13.1	3.6	2.6	2.3	2.2
1000000	15.4	3.6	2.6	2.3	2.2

$$D \leq k+1+k^{\alpha} \sum_{i=k+1}^{n} \frac{1}{i^{\alpha}} \tag{6.3}$$

We find that α has the biggest impact on the depth D. An increase of n has only a very limited influence on D. The maximum depth D for different choices of n and α is shown in Table 6.3. As can be seen in the table the maximum depth for α values larger than 1 stabilizes largely independent of the number of items n.

6.5.2 Hide-Difference

The *hide-difference* split hides the order of the k-th and $(k+1)$-th object, so that their order appear reversed in the estimate (i.e., the k-th object appears at the $(k+1)$-th position and the other way round) if less than D values are collected. This is done by hiding only the difference of the two objects, by again constructing filler values. For example, if for a top-1 the global scores are $(a,7),(b,5),(c,2),(d,2)$, then for two nodes the difference between the two objects a and b is hidden. For node one, that becomes $(b,5),(a,5),\dots$ and for node two $(c,2),(d,2),(a,2),(b,0)$. Here, the top-3 scores from the second node have to be collected to establish that object a comes before object b.

Procedure. The procedure to procedure that generates the split can be broken up into the following steps: (1) The global score of the k-th object is initially broken up into two parts, a score v_1 equal to the global score of the $k+1$-th object and the difference value v_2. (2) The global scores of all objects above the k-th object are distributed evenly. (3) The global score of the $k+1$-th object is placed as a single partial score at one node. (4) The value v_1 of the k-th object is placed on the same node above the previous value. The difference value v_2 is distributed evenly among the remaining nodes. (5) From the global scores of all other objects as many fillers as possible are generated that are equal to the divided difference value. (6) The order of the partial difference value and the fillers is reversed, so that the partial difference scores are located as deep as possible.

Depth. For the hide-difference split, the computation of the depth is more complicated. (See Table 6.2 for a definition of the variables.) Instead of the full global score of the k-th object, this adversarial split hides the difference between the k-th and the $(k+1)$-th global score $g_k - g_{k+1}$ at $m-1$ nodes. The scores to displace this difference are constructed from objects starting with the object at the $k+2$-th position downwards:

$$D = k-1+\left\lfloor \sum_{i=k+2}^{n} \min(1, \frac{1}{m-1}\left\lfloor \frac{g_i(m-1)}{g_k - g_{k+1}} \right\rfloor \right\rfloor) \tag{6.4}$$

We apply our formulation of the power-law:

$$D = k-1+\sum_{i=k+2}^{n} \min(1, \left\lfloor \frac{1}{m-1}\left\lfloor \frac{k^{\alpha}(k+1)^{\alpha}(m-1)}{i^{\alpha}((k+1)^{\alpha}-k^{\alpha})} \right\rfloor \right\rfloor) \tag{6.5}$$

Table 6.4: Maximum upper bound depth D (as factor of k) for the hide-difference split and different choices of the distribution parameter α and number of items.

n	$\alpha=1$	$\alpha=1.5$	$\alpha=2$	$\alpha=2.5$	$\alpha=3$
1000	39.3	16.4	9.5	6.5	5.0
10000	122.3	40.9	20.4	12.6	8.8
100000	384.6	102.5	43.9	24.3	11.7

We compute the upper bound of depth D as:

$$D \leq k - 1 + \sum_{i=k+2}^{n} \min(1, \frac{k^{\alpha}(k+1)^{\alpha}}{i^{\alpha}((k+1)^{\alpha} - k^{\alpha})}) \tag{6.6}$$

We find that the distribution parameter α has the largest influence, but for this split the factor n also has a large influence on D. The maximum depth D for different choices of n and α is shown in Table 6.4. As can be seen in the table, the depth D does not stabilize at any point in our sample, however the increase of D with increasing n is smaller for smaller values of α.

6.5.3 Summary

The parameter choices for the worst-case can be constructed from the two adversarial splits *hide-object* (for 2PA) and *hide-difference* (for 2PA). For 2PA, a depth parameter of ≈ 3.6 times the factor k guarantees correctness when the distribution parameter of the global score distribution is larger than 1.5. For both 1PA and 2PA, the upper bound of the depth D can be used to compute the worst-case depth parameter, when the number of items and the distribution parameter α are known. These parameter choices are conservative estimates and any score distribution observed in the real world is likely to allow smaller values. For this purpose, we turn next to random splits.

6.6 Average-Case Parameter Choices

In this section we investigate parameter choices for random splits of the global score distribution with the purpose of identifying parameter choices that also work well for the average-case. At the same time we show how tolerance towards limited error can help reduce the transfer cost further. As a first step, we introduce the random split and the error tolerance and then turn to simulation results.

6.6.1 Random Splits

With random splits we refer to the method of splitting each global score according to a different distribution. We construct the split by creating m (number of nodes) positive fraction for each object. We sampled these split values from a distribution and normalized them to ensure that the original score is conserved, by dividing them by their sum. The first split value for an object then represents the fraction of its global score that is placed on the node.

We use three different distributions to emulate differently skewed node score distributions. We choose an uniform distribution for no skew, the exponential distribution for more skew, and the Pareto distribution for very high skew. To get a representative overview, we executed each experiment 20 times. The alpha parameter of the Pareto and the scale ($\frac{1}{\lambda}$) parameter of the exponential distribution iterations were set to 1 for this experiment. The global power-law distribution was initialized to an α value of 1, a choice that produces relatively little skew in the global score distribution and thus helps our evaluation to err on the conservative side.

6.6.2 Rank Error

The rank error measures how far an object in the true top-k is displaced in the estimate from its true position. Our error formulation follows:

$$\text{error} := \sum_{i=1}^{k} t_i w_i$$

Here t_i represents the distance between the true position of an object and its estimated position and w_i is a weight that reflects the importance of the rank to the user. The weights from 1 to k sum up to 1. So, if all true objects are contained within the estimate, but each object is displaced by one position (e.g. every two items are switched), the error becomes exactly 1. If all items are displaced by more than one position, then the error can become more than 1.

We assume that the weights decrease with growing index $w_i > w_{i+1}$ and for the rest of the chapter we use weights that follow a power-law with an alpha of 1. This assumption and choice reflect the intuition that that lower positioned objects are usually much more important to the user than higher positioned objects (e.g., the top-1-st object is more important than the top-10-th object).

This error formulation faces one challenges: When a member of the true top-k is not contained in the estimate, the distance cannot be computed. In this case we need to replace the distance by a constant, the difference between the true rank and the true rank. In the following we refer to the replacement value of the object at position i as u_i. In our work we use a **constant of 2k as replacement value**, but a user may want to use a function of i instead to penalize the absence of the top-1-th item in the estimate stronger than the absence of the top-100-th item.

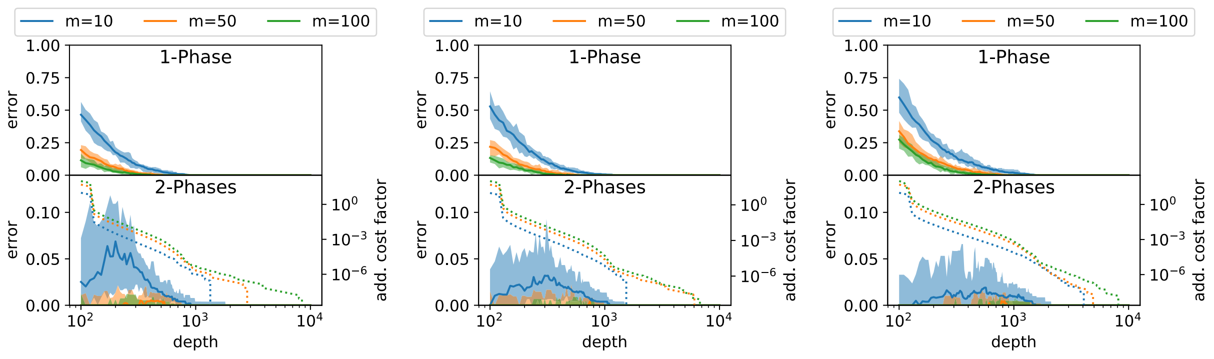
$$t_i := \begin{cases} |i - r_i| & i \in E \\ u_i & \text{otherwise} \end{cases}$$

Using this error formulation, we choose an error budget. We assume in this work that an **error budget of 0.1** should not be exceeded. For uniform error weights, this error value correspond to 10% of items to be displaced by exactly one position, one item to be displaced by $0.1k$ positions or something in between. With the power-law-based error weights, a slight displacement of few items higher up in the top-k has a larger impact than the larger displacement of many items lower down in the top-k.

6.6.3 Simulation Results

In Figure 6.4 we show the results of our simulation. We use three different node score distribution according to our choice of random splits: uniform, exponential and Pareto. In every plot we compare the choice of the depth parameter with the error that this parameter produces. The areas represent the region between minimum and maximum error and the line represent the average. Each color is associated with a different number of nodes. The top plots represent the performance of 1PA, while the bottom plot represents the performance of 2PA. As 2PA includes a second phase, we add the additional cost on a separate y-axis on the left with a dotted line in the bottom plot. If the depth is equal to, e.g., 100 and the additional cost factor is equal to 1.2, then the algorithm transfers 220 items per node.

In general, in this experiment we observe that smaller numbers of nodes increase the error. The reason behind this is, that the higher number of partial scores for an object ensure that any extreme value on any node, i.e., a partial



(a) Uniform. **(b)** Exponential. **(c)** Pareto.

Figure 6.4: Effect of the depth parameter on the rank error for random splits. Top plots represent 1PA's, bottom plot 2PA's performance. Regions mark min and max and lines the average error. The dotted line in the bottom plot is associated with the left y-axis. Each experiment was executed 20 times. 10,000 items are used, k is set to 100 and the replacement value u_i to 2k.

score that is higher or lower than its average and thus ends up at a higher/lower position, is influencing the overall result less.

A surprising result is that the skew of the random split has different impact on 1PA and 2PA. For 1PA, the error increases with increasing skew of the split. The higher skew leads to a situation where partial scores stray more from their original position. For 1PA, this means it has to go deeper, as it needs to observe all relevant partial scores by means of sequential access in a single phase. For 2PA, the same situation means that there is a higher chance for items to appear higher up and so it can collect them in the second phase, albeit at an increased cost, as becomes visible from the steep development of the additional cost curve.

For the error budget of 0.1, the depth parameter of the one-phase parameter has to be set to 4 times k items and for the two-phase algorithm a depth parameter of about 1.5 has to be chosen. We use these parameters continuing with our evaluation. These values are conservative estimates, as we made sure that they would work even for just 10 nodes and because we focus on a global score distribution with an alpha of 1. The same evaluation with higher α values should lead to a faster decreasing error due to the increase in skew.

6.7 Evaluation

In this Section we compare the performance of the parametrized one-phase and two-phase algorithms 1PA and 2PA against multi-phase algorithms from related work. We differentiate between worst-case and average-case versions of each algorithm. The former ensures that no error can be introduced, the latter has been chosen to limit the

Table 6.5: Parameter of the experiments.

dataset	no. items	no. nodes	k	no. iterations
Power-law & adversarial split	10000	25	100	1
Power-law & random split	10000	25	100	30
FIFA workload	(mean) 11226	26	100	25

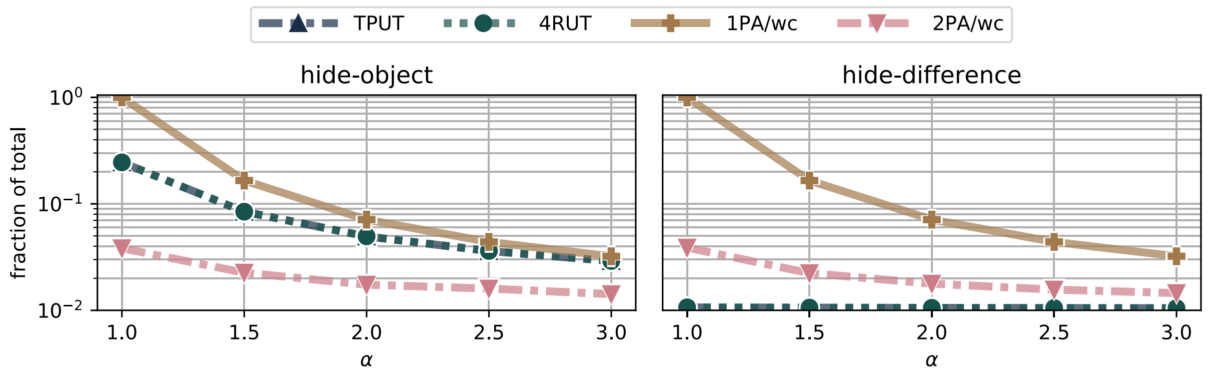


Figure 6.5: Data transfer volume for the adversarial splits.

error to an error budget of 0.1. We call these variants 1PA/wc, 1PA/avg and 2PA/wc, 2PA/avg for 1PA and 2PA, respectively.

We compare the two algorithms and their variants to three algorithms from related work: Two Phase Uniform Threshold (XPUT), Three Phase Uniform Threshold (TPUT) and Four Round Uniform Threshold (4RUT). Of the three, TPUT and 4RUT guarantee a correct result and XPUT produces an approximate result with an unbounded error.

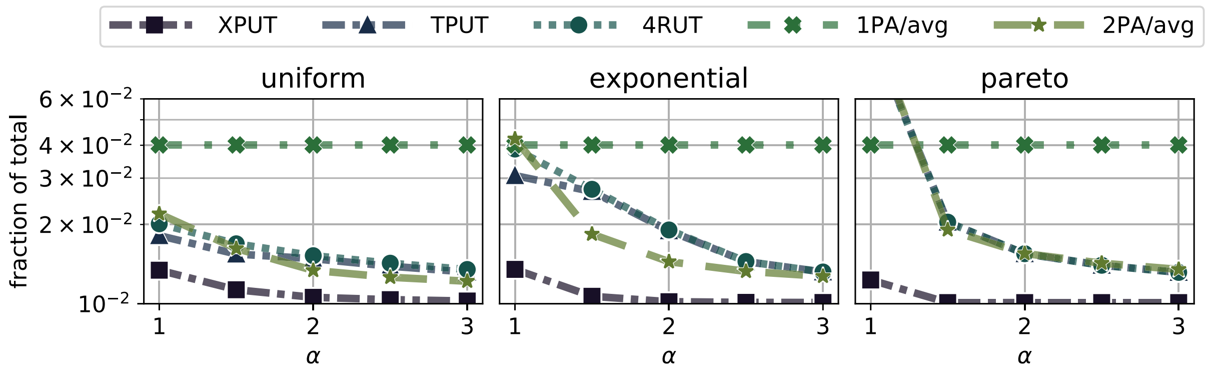
For our evaluation, we use three types of datasets. We compare the algorithms over adversarial splits and random splits that we have discussed above. Additionally we compare the algorithms on the FIFA dataset that has been used by related work. The parameters of each experiment are documented in Table 6.5. Our main evaluation criteria is the respective data transfer volume and we further require each algorithm to not exceed the chosen error budget of 0.1.

6.7.1 Adversarial Splits

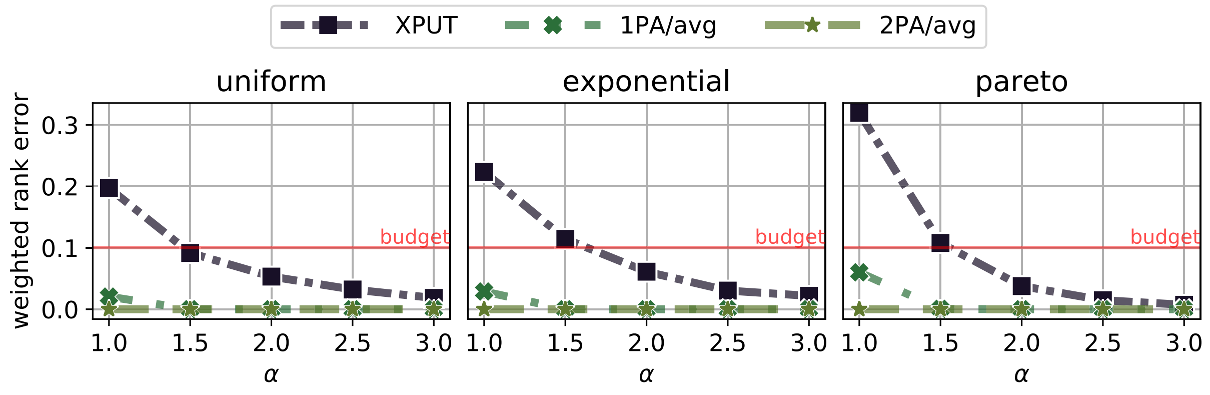
We compare the performance of the worst-case variants of 1PA and 2PA that we developed in Section 6.5 for the adversarial splits to other multi-phase algorithms. For a fair comparison, we exclude the approximate algorithm XPUT and instead chose TPUT and 4RUT.

In the line plot in Figure 6.5, we have created the adversarial splits from power-law distributions with varying alpha value (x-axis) and evaluate the algorithms in term of the data transfer volume they exchange (y-axis). We normalize the data transfer volume relative to the full dataset (i.e., a fraction of 0.1 means that 10% of the full data was transferred). The lines for TPUT and 4RUT are indistinguishable, as they perform the same for both splits.

A general observation is that the transfer cost decreases with an increase of the alpha value. This is expected, as the alpha value has a direct effect on the skewness of the global distribution and ensures that the difference between the global top-*k* scores and the respective partial scores are increased. For 1PA/wc, we find that it performs worse than the remaining algorithms and particularly so for the hide-difference split. This can be explained by its restriction to sorted access and a single phase. The 2PA/wc on the other hand algorithm outperforms 4RUT and TPUT for the hide-object split and is relatively close for the hide-difference split. Our main takeaway is that the two-phase algorithm outperforms the three- and four-phase algorithms for one adversarial split.



(a) Data transfer volume over baseline by split and alpha.



(b) Weighted rank error for each algorithm.

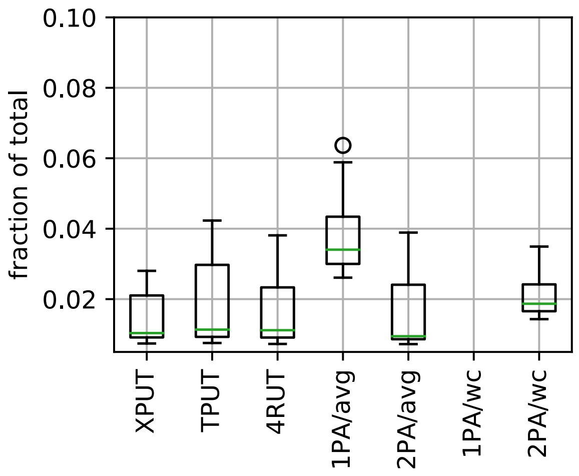
Figure 6.6: Multi-phase strategies performance on random splits.

6.7.2 Random Splits

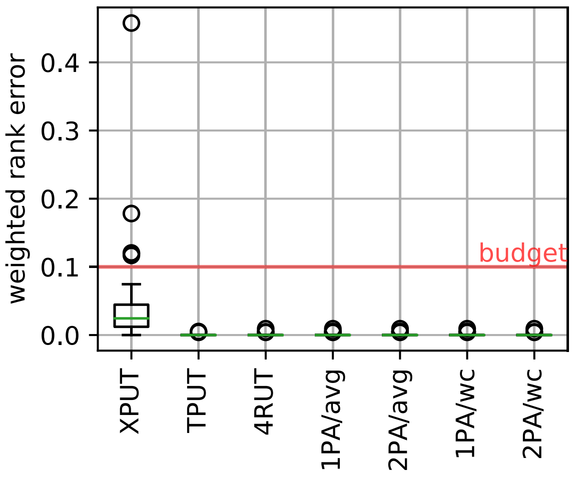
In Figure 6.6, we evaluate the algorithms over different random splits. Again, we use the global power-law distribution to generate global scores (x-axis). Then, we use random distributions of increasing skew to split the global scores across the nodes (individual subplots).

Figure 6.6a shows the normalized data transfer volume of the algorithms. For better visibility, we chose to use a log-scaled y-axis and to cut the y-dimension off at 0.06. Apart from one data point (Pareto-subplot with $\alpha = 1$), all algorithms achieve a data transfer volume between 1% and 6% of the data volume. Of the four algorithms, 1PA/avg performs consistently the worst with a data transfer volume of 4% and XPUT performs the best with a data transfer volume of 1$ for cases where $\alpha > 1$. The remaining algorithms perform similar and 2PA/avg outperforms TPUT and 4RUT on average even though it requires one less phase.

Considering the error produced by all algorithms (see Figure 6.6), we find that the XPUT violates our error budget limit of 0.1, while the other two approximate algorithms stay consistently below the limit. Our main takeaway here is that 2PA/avg can outperform the other algorithms while introducing only limited error.



(a) Data transfer volume over baseline per alpha.



(b) Weighted rank error for each algorithm.

Figure 6.7: Multi-phase strategies performance on FIFA Workload.

6.7.3 Real World Workload

Finally, we compare the algorithms over real-world data. We focus on the world cup data from 1998 which has been used by related work particularly work on TPUT [33]. The dataset contains access records for web objects stored across several nodes. The dataset is spread out over several days. We use every day as an individual experiment and average the results across experiments.

The results are depicted in Figure 6.7. Figure 6.7a shows the transfer cost incurred by each algorithm. In our plot we show the result for values in the range 2% to 10% of the total data transfer volume. The algorithm 1PA performed the worst both in its worst-case and its average-case variant. The worst-case variant exceeded the limit of 10%. The 1PA/wc places third-worst, but is far closer to the remaining algorithms. To better compare the remaining algorithms, we provide Table 6.6. Here we can see that 2PA/avg improves on TPUT and 4RUT by 13% and 5% on average. It only falls short of XPUT by 10%. Still 2PA/avg is a better choice than XPUT to its lower weighted rank error, as we can see from Figure 6.7b. Where both 1PA/avg and 2PA/avg produce a minimal error, XPUT exceeds the error budget of 0.1.

6.8 Summary

In this chapter we present a distributed top-k for the distributed top-k. We focus on multi-phase algorithms that estimate the distributed top-k by synchronously exchanging messages over a fixed number of rounds. Our investigation shows that the number of phases can be reduced, when the distribution of the data is known. Furthermore

Table 6.6: Mean improvement of 1PA/avg (left) and 2PA/avg (right) respectively versus other algorithms for FIFA, workload.

	XPUT	TPUT	4RUT
FIFA	-168.31% / -9.86%	-111.67% / 13.33%	-132.32% / 4.88%

we consider the tolerance towards error as means to reduce the data transfer volume further. We study the XPUT algorithm and propose two parametrized variant, the one-phase algorithm 1PA and the two-phase algorithm 2PA. For both algorithms, we investigate their performance in the worst-case and the average-case given that the objects aggregated scores follow a power-law distribution. We introduce a weighted rank error that reflects typical intuitions of top-*k* users: (1) That items higher up in the list are more important than lower and (2) That the actual score is of less importance than the rank of the object in the list.

Our evaluation shows that the use of 1PA generally leads to a drastic increase of the data transfer volume and so limits the use of this algorithm to cases where the number of phases has to be limited to one. 2PA in contrast transfer less volume than state of the art algorithms TPUT and 4RUT, while also saving between one and two phase(s). For a real-world dataset, the average-case variant of 2PA saves 13% compared to TPUT and 5% compared to 4RUT, while keeping the error far below 0.1. We conclude that 2PA/avg is a better alternative for scenarios where the global data follows a power-law and where the user has a slight tolerance for error.

Given that the error of the 2PA/avg variant is often far below 0.1 even though we designed it for an error of at most 0.1 and that the XPUT algorithm saves even more data than 2PA/avg, we see potential for future work on the mechanism of choosing the algorithm's parameter.

7 Load Shedding for Highly Parallel Stream Processing Systems

So far, we discussed optimizations for IIoT application accessing datasets. However, many IIoT applications rely on real-time event processing to implement fast responses. For example, an industrial robot arm may emit an event on each movement. Applications can use these events to track, recalibrate and, optimize the arm's behavior. Online event processing requires the use of specialized stream processing systems. Many modern stream processing systems are designed for high-volume data streams. They use a combination of the dataflow model and the MapReduce programming model. They deploy programs as highly parallelized operator graphs over a shared commodity infrastructure.

Unfortunately, systems based on commodity hardware are prone to suffer from failures of components. At the edge, bottlenecks can arise from multiple sources: (1) The shared use of local and uplink networks, (2) the limited resources of local server clusters, and (3) the sudden spikes of data generation due to environmental changes. Unmitigated, these bottlenecks can lead to the saturation of the queues of the system. Saturated queues increase the time records have to wait before being processed and thus the end-to-end latency of records passing through the system. Simultaneously, fully saturated queues of one operator can slow down preceding operators. This process is referred to as *backpressure*. In highly parallel stream processing systems, the slow-down of a single operator instance due to back-pressure can also decrease the output rate of parallel operator instances. A common solution for highly parallelized stream processing systems is the reactive scaling of the operator graph [137, 76, 36, 123].

Unfortunately, IIoT applications are often at least partially deployed at edge nodes, where computing resources are limited. Hence, reactive scaling has only limited potential for IIoT applications. An alternative solution is to drop data records selectively to improve performance. Related work refers to this solution as *load shedding* [14, 150]. Existing work on load shedding fits into two categories. The first category of work focuses on control mechanisms to decide the intensity and location of load shedding across a stream processing graph [14, 150]. The second category focuses on the capacity of individual load shedders to drop the "right" records, e.g., records that do not distort the data distribution [127] or have less influence on an operator's state [161]. This work is related to adaptive sampling (selecting interesting records) and adaptive filtering (selecting unpredictable records) [72]. The fields differ less by their chosen methods and more by the motivation. Load shedding is primarily focused on performance, while adaptive sampling and filtering are primarily focused on result quality. This work is part of the first category. We investigate where and how much load shedding needs to be applied to the highly parallel stream processing graph. We leave the selection of records that are dropped to future work. Our work addresses load shedding for the processing graphs of highly parallel stream processors. We show that the placement of load shedding has to strike a balance between wasted work, i.e., processor cycles spent on records that are later dropped, and data loss, i.e., data records that are dropped even though they do not pass the bottlenecks. We propose a solution that limits wasted work while avoiding data loss altogether.

Our Contributions

- We describe the backpressure patterns appearing in highly parallel stream processing systems and how they can exacerbate the effect of a performance bottleneck.
- We introduce the load shedding controller MERA. MERA decides the intensity and location of load shedding to react to performance bottlenecks.
- MERA reduces the end-to-end latency by more than one magnitude compared to vanilla Apache Flink. In some cases, it also increases the throughput by 23%
- MERA reduces wasted work and data loss compared to approaches that either drop at the location of the bottleneck or the source, respectively.

7.1 Highly Parallel Stream Processing Systems

We focus on highly parallel stream processing systems in general and Apache Flink [34] in particular. These stream processing systems follow the dataflow model first described by Sutherland [149]. Its main idea is that data streams are processed as they are consumed and emitted by an operator graph. Highly parallel stream processing systems parallelize each operator by deploying the operator logic in multiple operator instances, a *task set*, across a server cluster. The degree of parallelism per operator, i.e., the number of tasks per operator, can vary. As a result, highly parallel stream processing systems deploy directed acyclic task graphs. Figure 7.1 shows a task graph of Map and Reduce tasks connected across multiple hosts (gray stripes). We simplify our investigation by excluding dataflow programs that include joins between two or more operators [90] and cyclic iterative task graphs [60]. We also exclusively focus on continuous operators (e.g., counting packets over time) in contrast to windowed operators (e.g., counting packets over 5-minute intervals).

7.1.1 Execution Model

Records are ingested into the task graph at source tasks and emitted at sink tasks. Tasks in-between process data records and push them forward through the graph. We refer to tasks relatively closer to the source and the sink as *upstream* and *downstream* tasks, respectively. Apache Flink and other highly parallel stream processing systems assign each task in-memory input and output queues that absorb bursty traffic. A task processor collects records

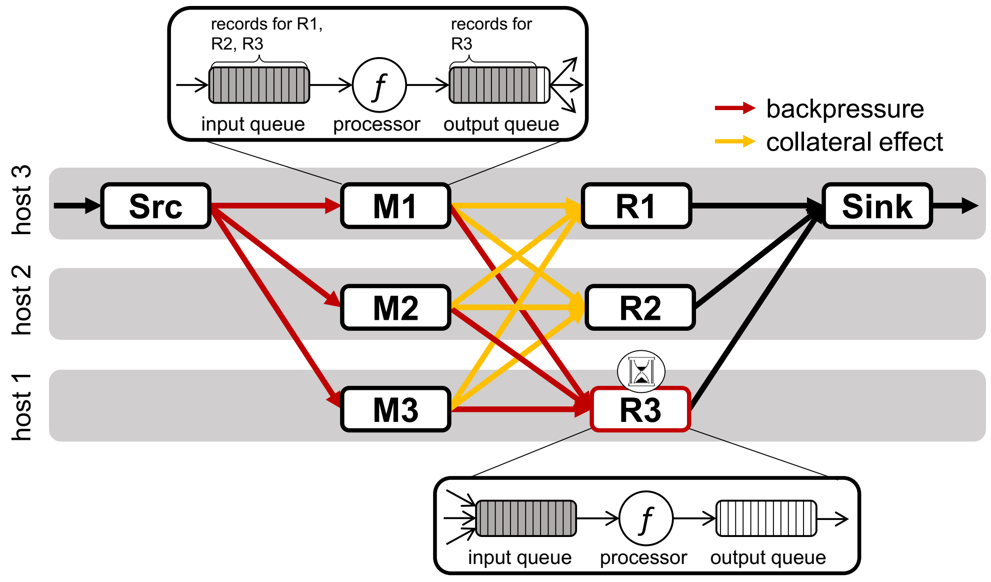


Figure 7.1: Simple task graph of Map (M1, M2, M3) and Reduce (R1, R2, R3) tasks. A single source (Src) ingests data, and two sinks (Si1, Si2) emit data.

from the input queues and pushes them into the output queue. (To improve throughput at the cost of latency, records are stored and transferred in groups [34].) An independent data transfer module takes records from the output queue to the input queue of the downstream task. Depending on the location of the downstream task, the records are either transmitted directly via the memory or transferred over the network.

Similar to MapReduce, operators can be of two fundamental types: *ungrouped* or *grouped*. The ungrouped type includes Map operators and the grouped type includes Reduce operators. An ungrouped task consumes any records without preference. A grouped task only consumes records that are members of groups that are assigned to the task. The data exchange between the task sets of two operators depends on the type of the downstream operator. If two operators are connected, and the downstream operator is a grouped operator, then the tasks of both operators are connected in an all-to-all fashion. (See the Map to Reduce connectivity in Figure 7.1.) In this case, the exchange between the two task sets is controlled by a partitioning scheme. The partitioning scheme assigns record groups to individual tasks. All records belonging to that group are forwarded to the respective task. If the downstream operator is ungrouped, then the task sets are connected in a 1-to-1 fashion if both operators use the same degree of parallelism. Otherwise, they are also connected in an all-to-all fashion. In this case, a load balancing mechanism (usually round-robin) decides the distribution of records.

7.1.2 Load Imbalances

How evenly or unevenly data flows through the task graph, i.e., to what degree one operator's task set receives the same number of records or *load*, depends on multiple factors. First, operators have a varying output-to-input ratio of records (also referred to as *selectivity*). Some operators may generate a high number of records from a single record or the other way round. For complex operators, the selectivity can depend on ingested data. Second, each grouped operator assigns groups to individual tasks according to some partitioning scheme. Depending on the data distribution, this can lead to imbalanced load distribution, with a small number of tasks receiving the majority of data records. Performance variabilities between different tasks amplify load imbalances. Possible Performance variabilities include hardware or network faults, the influence of 3rd party applications, and the influence of complex operator logic. Load imbalances and performance variabilities can remain undetected when the system is under little load and appear only when the input rate of the system increases due to changes in the environment. Similarly, a sudden change in the data distribution can introduce an uneven load distribution among the task set of a grouped operator. Load imbalances and performance variabilities can ultimately lead to situations where tasks become bottlenecks as they process records slower than they receive them. Predicting these events in advance is difficult and scaling the deployment of a job for a worst-case peak load can be costly. Particularly in resource-constrained environments, the altogether avoidance of bottlenecks is very challenging.

7.1.3 The Direct and Collateral Effects of Backpressure

Figure 7.1 shows an example of a bottleneck and its effect. There, task $R3$ has become a bottleneck. Its processing rate is slowed down. On the one side, this reduces its output rate and the system throughput. On the other side, it leads to the saturation of its input queue. A saturated input queue can only receive records at the processing rate. As a consequence, records meant for *R3* can only be delivered at the processing rate of $R3$. This slow-down leads to the saturation of the output queues and later input queues of $M1$, $M2$, and $M3$. This effect is called *backpressure*. Unmitigated, backpressure continues towards the source *Src*. The effects of backpressure are two-fold. First, the saturated queues increase the end-to-end latency of all records passing *Src*, $M1$, $M2$, $M3$, and $R3$. Three factors influence the magnitude of this increase: (1) the length of the operator graph upstream of the bottleneck, (2) the size of the queues, and (3) the size of individual records. Second, as the queues of $M1$, $M2$, $M3$ become saturated with records for $R3$, the records meant for $R1$ and $R2$ can also only move forward once space is freed up. Therefore, both $R1$ and $R2$ receive records at reduced speed. This collateral effect further reduces the throughput of the system. Different from the first effect, the second collateral effect depends on the task graph. It does not appear when the bottleneck is the only task of the respective operator. To our knowledge, we are the first to consider the collateral effect of backpressure for highly parallel stream processors. The increased end-to-end latency and reduced throughput can decrease the utility of applications relying on the output of the stream processor.

7.2 Approach

Our goal is to ensure low end-to-end latency and high throughput by shedding records. Backpressure affects both end-to-end latency and throughput negatively. To avoid backpressure, we need to ensure that no task receives records faster than it can process. In other words, we need to use load shedding to remove the difference between the input rate of a task and its processing rate. This leaves the choice for the location where the load is to be shed (i.e., where records are dropped). Two opposing concerns influence this choice: *wasted work* and *wasted data*. Wasted work commonly refers to the effort the system expended upon a record that is ultimately dropped. With data loss we refer to dropped records that would not have passed the bottleneck.

For single-CPU stream processors, wasted work describes the number of cycles that the CPU spent on a record that is later dropped. Shedding the record at an earlier position produces more idle time, which the processor can use to speed up other parts of the pipeline, including the bottleneck [15]. Eliminating wasted work is essential for single-CPU stream processors. Distributed stream processors can only use the additional idle time of a single CPU on the bottleneck if it is located on the same host and shares the CPU. However, for distributed stream processors, wasted work includes the time the records spend in the network. The elimination of wasted work can speed up the bottleneck if the bottleneck is caused by the network. Additionally, wasted work can be an important factor when different applications share the same hardware or computing power is priced for usage time. A simple load

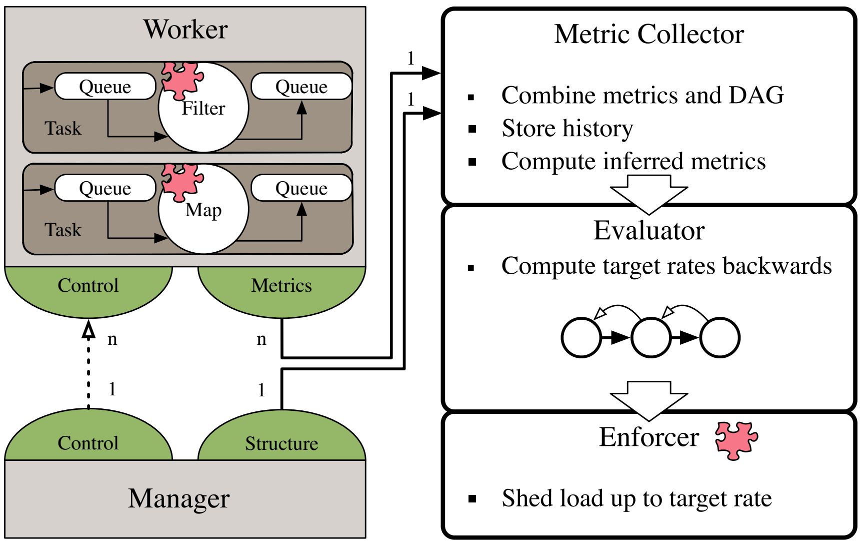


Figure 7.2: MERA Controller.

shedding strategy to eliminate wasted work is to drop records exclusively at the source. We call this strategy *source-drop*. data loss can occur when data is dropped at a task that is connected to several tasks downstream of which only a subset of them are or are connected to bottlenecks. The source-drop strategy is likely to incur data loss. A load shedding strategy that avoids data loss completely is to drop records directly at the bottleneck. We call this strategy *local-drop*. However, the local-drop strategy will incur data loss (unless the source is simultaneously the bottleneck).

For distributed stream processors, data loss directly reduces precision, while wasted work only indirectly reduces performance and increases cost. Therefore, we argue that the avoidance of data loss takes priority over the avoidance of wasted work. To eliminate data loss and reduce wasted work, records need to be dropped as early as possible, but only at tasks that push data exclusively towards bottlenecks. The disadvantage of load shedding is that it introduces an error into the result of the stream processor. Furthermore, unless the computation is naïvely simple, the introduced error cannot be accurately measured [16]. We need to ensure that only a minimal number of records are dropped.

7.3 MERA

We introduce the load shedder controller MERA. It controls load shedding throughout the task graph. Its purpose is to identify sustainable input rates at which no task becomes a bottleneck and to enforce these rates while minimizing wasted work. Our solution is an integration of multiple steps: (1) We collect metrics from all tasks, (2) we identify bottlenecks and their sustainable input rates, (3) we identify the best locations to drop records, (4) we communicate drop rates to the load shedders.

MERA is designed as a service on top of the highly parallelized stream processor Apache Flink [34] and interfaces with Flink through Flink's metric exchange interface and Flink's REST API. Figure 7.2 depicts the architecture. We assume that all tasks deployed on the same machine are managed by the same *Worker*, and a single *Manager* administers the deployment as a whole. In this setup, the *Worker* coordinates the setup, tear-down, network interaction, and metric collection of the tasks. Our solution consists of three modules. The *MetricCollector* collects metrics

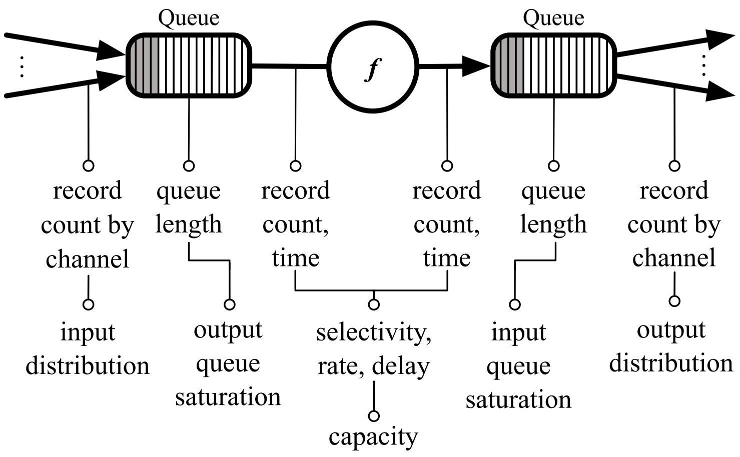


Figure 7.3: Structure of the tasks and metric collection.

from all *Workers*. It stores them on a directed acyclic graph and infers metrics. The *Evaluator* module performs the optimization. It propagates the drop rates backward and identifies the best location to drop records. Each optimization step is run in intervals of 5 seconds. The *Enforcer* communicates the drop rates to the load shedders. Our load shedders are tightly integrated within the tasks. The concepts used here are similar to other highly parallel stream processors [108, 3, 97, 73]. We modified Apache Flink slightly to track the number of records exchanged over each edge and the processing delay of all tasks. In the following, we discuss the function of each component, including the load shedders, in detail.

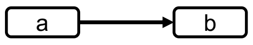
7.3.1 Metrics

Figure 7.3 shows the metrics that we collect from each task. We store all metrics on the nodes and edges of a graph data structure. We collect several basic metrics, including the queue lengths, the processing delay, the number of ingested and produced records, and the number of records passing over each channel. From these, we can infer a set of new metrics. First, we establish the selectivity (output-to-input ratio) and the input and output distribution (fraction of records received and sent over each channel) for each task. These metrics give us insight into how data flows through the graph. Second, we measure the input rate for each source task and infer the processing capacity for each task from the respective processing delay. The processing capacity details the record number a task can process without creating backpressure. Third, we infer the saturation of all queues. If more records are enqueued between any two tasks than a task can handle in 1 second, we penalize this task's capacity. Fourth, using the input rates and the graph, we can compute each task's expected input rate. All tasks that have an expected input rate higher than their processing capacities are bottlenecks. For each bottleneck, we define its drop rate as the difference between its expected input rate and its penalized capacity.

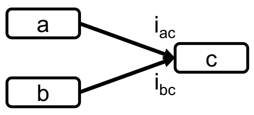
The metrics described here form the input of our system. To respect changes over time, we collect Flink's metrics every 5 seconds and construct our metrics as moving averages over 5 time intervals. For dataflow programs that change quicker, exponentially weighted averages can be used.

7.3.2 Drop Rate Propagation

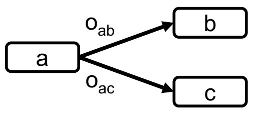
The optimizer ensures that the drop rates of all bottlenecks are installed in the task graph. It propagates the drop rates backward through the graph. The propagation has to deal with four connectivity patterns 1-to-1, all-to-1, 1-to-



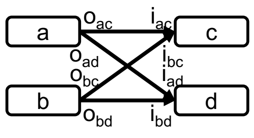
(a) One-to-one pattern.



(b) All-to-one pattern.



(c) One-to-all pattern.



(d) All-to-all pattern.

Figure 7.4: Drop Rate Propagation Patterns.

all, and all-to-all. The patterns are visualized in Figure 7.4. We discuss each pattern in turn and use the variables described in Table 7.1.

Backward propagation of the drop rate for 1-to-1 patterns (see Figure 7.4a) is straightforward. If task a has a higher drop rate d_a than task b, then task a's remains d'_a drop rate stays the same. If task b's drop rate d_b is higher, task a's revised drop rate is replaced. In this case, the propagation of task b's drop rate has to account for the selectivity (output-to-input ratio) s_a of task a. In both cases, task b's drop rate is set to 0. We can express the changes as an equation:

$$\begin{aligned} d'_a &:= \max\{d_a, 1/s_a d_b\} \\ d'_b &:= 0 \end{aligned} \tag{7.1}$$

The all-to-1 pattern in Figure 7.4b is very similar. The only difference is that the drop rate d_c of task c has to align according to the input distribution i_{*c}. The input distribution (i_{ac} and i_{bc}) reflects the fraction of records arriving from tasks a and b.

$$\begin{aligned} d'_a &:= \max\{d_a, 1/s_a i_{ac} d_c\} \\ d'_b &:= \max\{d_b, 1/s_b i_{bc} d_c\} \\ d'_c &:= 0 \end{aligned} \tag{7.2}$$

The 1-to-all pattern in Figure 7.4c works slightly differently. For this pattern, drop rates are only propagated backward if both task c and d have a non-zero drop rate. Then, the minimum drop rate is propagated, and the rest stays local. The propagation has to account for the output distribution o_{a*}, i.e., the fraction of records sent to each subsequent task. In the case that the drop rate of task a or b is higher than the propagated drop rate, we need to ensure that the drop rate never becomes negative.

Table 7.1: Variables for Drop Rate Propagation.

Term	Description
o_{ab}	Fraction of all records emitted by task a going to task b
i_{ab}	Fraction of all records received by task b coming from task a
d_a	Drop rate of task a
d'_a	Revised drop rate of task a

$$\begin{aligned} d'_a &:= \max\{d_a, 1/s_a \min\{1/o_{ab}d_b, 1/o_{ac}d_c\}\} \\ d'_b &:= \max\{0, d_b - s_a d'_a\} \\ d'_c &:= \max\{0, d_c - s_a d'_a\} \end{aligned} \tag{7.3}$$

Finally, the all-to-all pattern in Figure 7.4d is a combination of the other patterns. It has to account for the input and output distributions and the respective selectivities.

$$\begin{aligned} d'_a &:= \max\{d_a, 1/s_a \min\{1/o_{ac}i_{ac}d_c, 1/o_{ad}i_{ad}d_d\}\} \\ d'_b &:= \max\{d_b, 1/s_a \min\{1/o_{bc}i_{bc}d_c, 1/o_{bd}i_{bd}d_d\}\} \\ d'_c &:= \max\{0, d_c - o_{ac}s_a d'_a - o_{bc}s_b d'_b\} \\ d'_d &:= \max\{0, d_d - o_{ad}s_a d'_a - o_{bd}s_b d'_b\} \end{aligned} \tag{7.4}$$

Based on the four patterns, we can define a single propagation rule. For each task x_k of operator x and task y_l of operator y, with y following immediately on x, the drop rates can be propagated as follows:

$$\begin{aligned} d'_{x_k} &:= \max\{d_{x_k}, 1/s_{x_k} \min_{y_l} 1/o_{x_k y_l} i_{x_k y_l} d_{y_l}\} \\ d'_{y_l} &:= \max\{0, d_{y_l} - \sum_{x_k} o_{x_k y_l} s_{x_k} d'_{x_k}\} \end{aligned} \tag{7.5}$$

Additional care has to be taken if only a subset of the tasks are load shedders. In that case, drop rates may only be propagated when a load shedder is located upstream. We compare this solution against local-drop and source-drop. Local-drop drops records at the bottleneck. Source-drop only drops record at the source, but as soon as a bottleneck emerges. The following rules summarize Source-drop can be summarized by the following rule:

$$\begin{aligned} d'_{x_k} &:= \max\{d_{x_k}, \max_{y_l} 1/s_{x_k} 1/o_{x_k y_l} i_{x_k y_l} d_{y_l}\} \\ d'_{y_l} &:= 0 \end{aligned} \tag{7.6}$$

Source-drop propagates the full drop rate backward, which can lead to unnecessary data loss (or data loss).

7.3.3 Load Shedder

We use a straightforward approach to enforce the drop rates. For a drop rate of d_x, we drop the first floor(x) records that arrive at task x in each second. We drop an additional record with the probability $x - \text{floor}(x)$. The advantage of this simple approach compared to other solutions such as uniform sampling is that it does not rely on estimates of previous input rates. As the drop rates are propagated and split over the graph, input rates of one task may vary greatly from one second to the next. The disadvantage of this approach is that it can skew the data distribution. More sophisticated solutions can rely on the controller to estimate and communicate input rates. A survey on possible sampling methods by Giouroukis et al can provide a starting for future work [72].

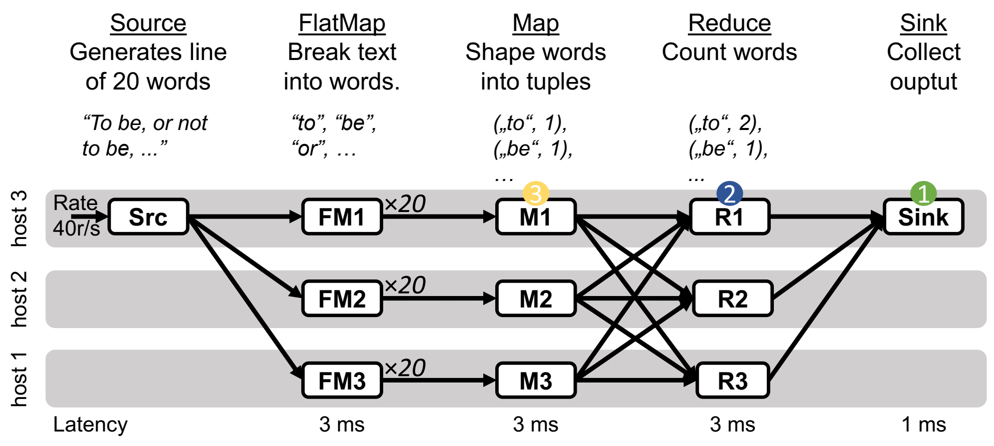


Figure 7.5: Dataflow program. The figure depicts a simple three-phase word count deployed over three hosts (gray stripes). We introduce bottlenecks at three numbered locations.

7.3.4 Limitation

In its current state, MERA is limited to dataflow programs that are relatively stable over time. It does not apply to dataflow programs, where either selectivities or output distributions change rapidly. An example for this case is an iterative dataflow graph with complex operator logic. Similarly, dataflow programs with windowed operators may encounter difficulties if window sizes change rapidly. An example for this case is an operator that defines windows based on event time (i.e., the time that events where produced) instead of processing time [3] (i.e., the time events arrive at the operator). An additional prediction component is necessary to cover these cases. We leave this extension to future work.

7.4 Evaluation

We use the simple task graph shown in Figure 7.5 to evaluate the performance of MERA. The graph has been inspired by the simple WordCount program. WordCount has previously been used as a "hello world" program for both MapReduce and Apache Flink. With this approach we follow similar related work [66, 89]. The source *Src* generates text lines. Each line is split by the FlatMap tasks $FM1$, $FM2$, and $FM3$ into individual words. (FlatMap tasks differ from Map tasks in that they can emit zero, one, or multiple records for each consumed record. Map tasks consume and emit precisely one record.) Each word is then formatted into a tuple by the subsequent Map tasks $M1$, $M2$, and $M3$. The Reduce tasks $R1$, $R2$, and $R3$ consume the tuple and emit a new tuple containing the word and the number of times it has appeared so far. The sink *Si* consumes the tuples. We ensured that the Map tasks distribute words evenly among the Reduce tasks. The program is very simple compared to common dataflow programs, but its strength is that it is easily understandable.

To show the effect of load shedding, we created an artificial bottleneck. We realize the bottleneck by suspending the thread running a task. This method is limited to the suspension of at least a single millisecond. We choose the input rate of the program accordingly and also slow down non-bottleneck tasks. At the source, we introduce a input rate of 40 lines per second at the source. Due to the selectivity of the FlatMap, the Sink receives 800, and the Map and Reduce tasks each receives about $\frac{800}{3}$ records a second. At the FlatMap, Map, and Reduce, we introduce a delay of $3ms$, and at the sink a delay of $1ms$. This leads to processing rates of $\frac{1000}{3}$ and 1000 records per second, respectively.

We evaluate four conditions: First, *vanilla* Apache Flink does not shed load at any task beyond the source. We only shed load once the source output queue runs full and the source cannot produce new records. All new records that would have been generated are dropped. Second, source-drop also drops records only at the source, but does so with the goal of removing backpressure from the task graph. Source-drop is based on Equation 7.6. Third, local-drop sheds load at the bottleneck. Finally, MERA drops records at locations computed according to the rules introduced in Equation 7.5.

We present three scenarios. In each scenario, we produce bottlenecks by doubling the delays of a single task. The scenarios cover tasks with different connectivity patterns and thus favor different conditions. The first scenario produces a bottleneck at the single sink task Si. As all records produced by the source end up on this task, they may as well be dropped by the source. This favors the source-drop condition. The second scenario creates a bottleneck at the Reduce task $R1$. As this task receives records from all Map tasks, dropping records at an earlier position leads to data loss. This scenario favors the local-drop condition. In the final scenario, the bottleneck is the Map task $M1$. This scenario favors neither source-drop nor local-drop.

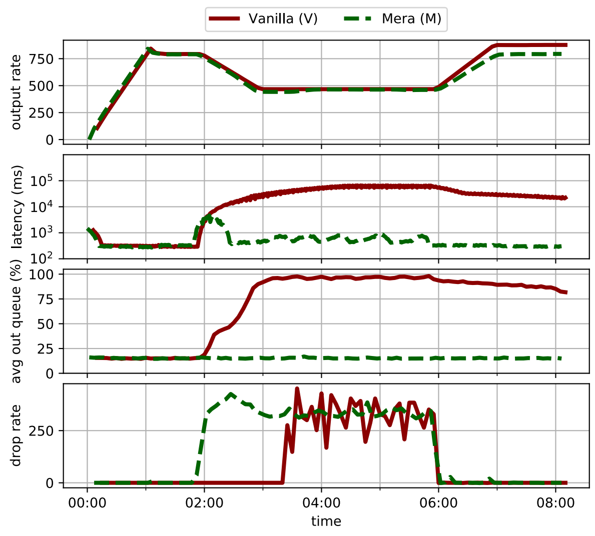
We designed a set of metrics to compare our conditions across the scenarios. Of major concern, for a stream processing system, are the record *end-to-end latency* and the system's *throughput*. To precisely measure latency, we synchronized the clocks in our test cluster and included the start time in every record. We also track the *average queue saturation* of the system to explain changes in the latency. Additionally, we track the *effective drop count* of all conditions. The effective drop accounts for varying selectivity between tasks. It is the sum of all drops at any operator normalized to indicate the number of records that do not appear in the output of the sink due to the drop. (E,g., a drop at $FM1$ is equal to a drop of 20 records.) To show the difference between the local-drop and MERA, we compute a measure of *wasted work*. Our measure only accounts for the processing time spent on the record at each task. We normalize this time according to the selectivity of each task. Finally, we track for both local-drop and MERA the *drop distribution* over all operators. This metric can be used to explain differences in the wasted work.

7.5 Results

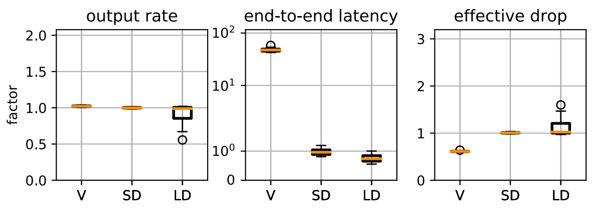
We executed the experiments over three machines in a cluster. The hardware was hardly utilized due to the limited input rate of the program. Each scenario was executed 30 times, and each experiment ran for about 8 minutes. The bottleneck was activated at around 2 minutes and deactivated at 6 minutes. The timings varied slightly between each experiment. We discuss each scenario in turn.

Table 7.2: Summary of conditions.

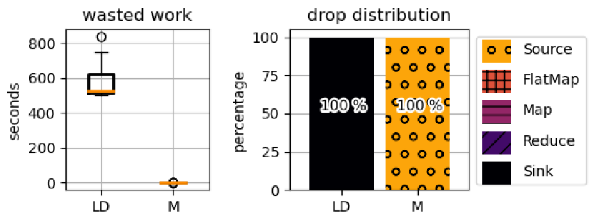
Condition	Description
Vanilla (V)	Vanilla Apache Flink.
Source-drop (SD)	Load shedding at the source (see Equation 7.6).
Local-drop (LD)	Load shedding at the location of the bottleneck.
MERA (M)	Shedding location is determined by propagation rules (see Equation 7.5).



(a) Comparison of vanilla to MERA condition.



(b) Summary over multiple iterations.



(c) Comparison between local-drop and MERA.

Figure 7.6: Results for Scenario 1.

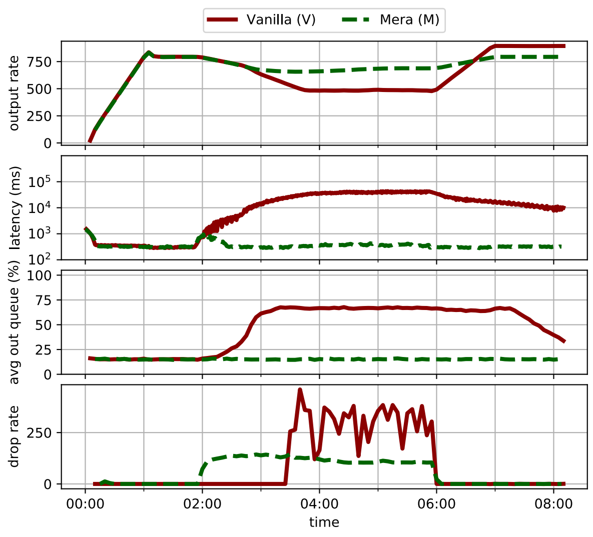
7.5.1 Scenario 1: Slow Sink

In this scenario, we double the latency on the sink node to $2ms$ to create a bottleneck. Results are shown in Figure 7.6. Figure 7.6a compares the vanilla condition to the optimizer for the first iteration of the scenario. The top panel shows the output rate. The top panel shows a clear dip in the output rate as soon as the bottleneck is activated. Both the vanilla and MERA perform about equally for the largest part of the execution. Only at the end does vanilla have an increased speed over MERA. The reason can be gleaned from the third panel. The queues are almost fully saturated, and the excess records can be processed. The second panel shows the latency increase for both conditions. Both show an initial spike of the end-to-end latency up to several seconds. Under the vanilla condition, the end-to-end latency increases from there up to tens of seconds. In MERA, the end-to-end latency quickly decreases below a second and remains there. The fourth panel explains this behavior. The drop rate of MERA reacts to the increase in processing delay and remains steady at about 250 to 400 records. The drop rate of vanilla, in contrast, only starts when the queues are fully saturated, and the source is reached.

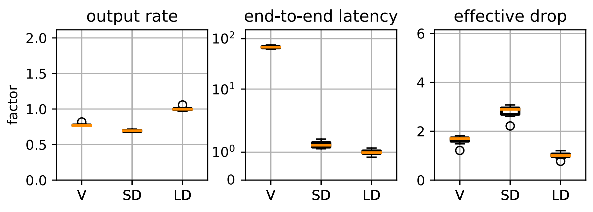
Figure 7.6b visualizes the results aggregated over all iterations. We have normalized all results to the results achieved by MERA. We see that apart from vanilla, all conditions perform very similarly. The gain compared to vanilla is restricted to the end-to-end latency, where the improvement is more than one magnitude. More insight can be gained from Figure 7.6c. In the left panel, we see a drastic difference between the wasted work of the local-drop condition and MERA. The reason can be learned from the panel on the right. MERA drops at the source, while local-drop drops at the sink. We here assume that the source has no processing delay on its own.

Naturally, one should be skeptical with respect to the concrete numbers in this evaluation. The program is a toy example that at the same time uses processing delays that are far too high for and record sizes that are too small for real-world programs. Simultaneously, the input rate is too low compared to real-world programs, and the length of the program is too short. Significant is the relative difference between the conditions.

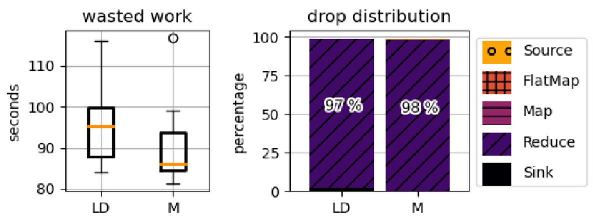
We can gain a number of insights from this scenario. First, using load shedding can drastically lower the end-to-end latency of individual records. Second, in this scenario MERA and source-drop behave the same. Third, particularly



(a) Comparison of vanilla to MERA condition.



(b) Summary over multiple iterations.



(c) Comparison between local-drop and MERA.

Figure 7.7: Results for Scenario 2.

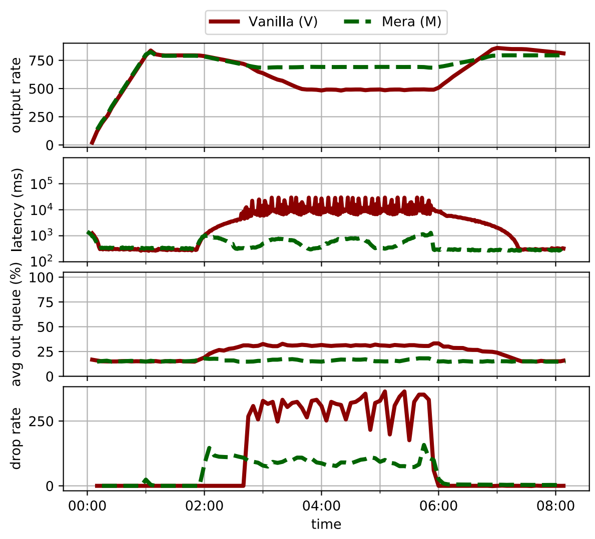
for bottlenecks at the sink, the difference in wasted work between local-drop and the other load shedding conditions is large.

7.5.2 Scenario 2: Slow Reduce Task

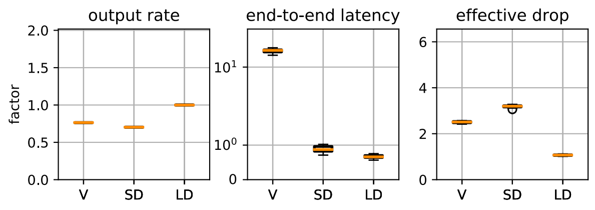
In the second scenario, the bottleneck is introduced at a single Reduce task. Otherwise, the experiments and the plots are the same. We can observe a number of important differences from Figure 7.7. In Figure 7.7a, the average queue saturation shown in the third panel is smaller. This is because the queues at the sink and the other Reduce tasks are running empty. More interesting is that the output rate in the second panel is actually higher for MERA during the time the bottleneck is active. This shows what we have previously called the *collateral effect*. The saturated queues at the Map tasks slow down the transfer of records to Reduce tasks that are not bottlenecks themselves. We see this effect reflected in the drop rate. Once vanilla starts to drop records, it drops around 3 times as many records. The latency and are similar to scenario 1.

Figure 7.7b shows that not only vanilla but also source drop are negatively affected by the bottleneck. Dropping records at the source has the negative consequence that more records than necessary have to be dropped. This shows both in the first and the last panel of the plot. The latency is only slightly affected. Different than in scenario 1, Figure 7.7c shows little difference between local-drop and MERA. In contrast, for this bottleneck, both conditions behave almost the same. MERA cannot propagate the drop rate further backward because only a single Reduce task is affected by the bottleneck.

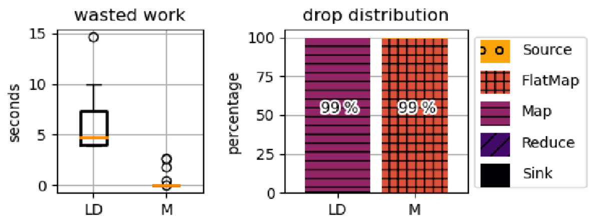
The central insight in this experiment is the observation of the *collateral effect* in action. The difference is about 23% of the average output rate while the bottleneck is active.



(a) Comparison of vanilla to MERA condition.



(b) Summary over multiple iterations.



(c) Comparison between local-drop and MERA.

Figure 7.8: Results for Scenario 3.

7.5.3 Scenario 3: Slow Map Task

In our third scenario, we create a bottleneck by doubling the processing delay of the top Map task. Figure 7.8a shows results very similar to before. Again the average queue saturation decreases as fewer queues are affected. The output rate is also better for MERA compared to vanilla. We notice a slightly different behavior in the end-to-end latency. For vanilla, the end-to-end latency increases up to 10^4 milliseconds and oscillates on that level. The explanation is that only a third of all records pass the Map task that is a bottleneck and its saturated input queue. Two third of the tasks are slowed down by the saturated output queue of the Source task, but then pass the remainder of the task graph unhindered. Hence, for every three records arriving at the sink, one has a considerably larger end-to-end latency.

Figure 7.8b also draws a very similar picture to the one seen in scenario 2. The difference only becomes apparent in Figure 7.8c. MERA almost completely avoids wasted work by dropping at an earlier location than local-drop. Additionally, similar to local-drop, it achieves an output rate that is again about 23% higher than in both the vanilla and source-drop conditions. In this scenario, MERA has the opportunity to identify a middle path that improves on both alternatives. The difference in wasted work may appear less than in other scenarios, but this is caused by the bottleneck's location and the task graph. Similar gains can be achieved for different topologies and particularly once multiple bottlenecks become involved.

Scenario 3 differs from scenario 2 in one important aspect. The bottleneck in scenario 2 is a grouped task, and the distribution of records across grouped tasks cannot be changed during runtime without state migration. In contrast, the bottleneck in scenario 3 is an ungrouped operator that receives records through a round-robin scheme. In principle, the round-robin scheme can be replaced by a different scheme that distributes records only to tasks that do not exhibit backpressure. A load-sensitive round-robin scheme can decrease the impact of the bottleneck considerably.

7.6 Related Work

Bert Sutherland introduced the dataflow model in his 1966's Ph.D. thesis [149]. Its basic tenet is that data is processed as it "flows" across a set of static operators. The model has guided both the development of stream processing systems and highly parallel data processors such as MapReduce. Çetintemel et al. [38] defined the requirements for stream processors (SPs) or stream processing engine (SPEs) to be (1) time series operations (particular windowed operations), (2) real-time stream processing, and (3) the management of bursty data streams. The system Aurora [1] was developed as a single-host stream processor to meet these three requirements. The system Medusa extended Aurora to a distributed stream processor (DSP) [18]. Later, Borealis [38] used Medusa and Aurora as a starting point for the development of a distributed stream processing engine with extended application semantics (e.g., retraction of earlier results).

Simultaneously, highly parallel data processors rose to prominence with the introduction of private and public clouds [5], which allowed for quick access to large clusters of commodity machines. Google's MapReduce [52] has been one of the most influential systems. Initially, these systems were built with a focus on batch workloads. Later generations extended the programming model to stream processing. Notable examples include Apache Flink [34], which we use as host system in this chapter, Heron [97] and its predecessor Storm that is used by Twitter, Timely Dataflow [108] that has been developed by Murray et al. at Microsoft and the Dataflow service by Google [3, 73]. These systems use MapReduce's basic operator model and parallelization scheme and extend it to the streaming case with additional semantics, state management mechanism and fault tolerance. Unlike distributed stream processors that are meant to execute multiple queries on the same system, highly parallel stream processors have been developed to execute a single program continuously.

Similar to MapReduce, highly parallel stream processing systems that rely on commodity hardware and shared infrastructures are prone to failure and the emergence of bottlenecks. Approaches to mitigate bottlenecks in stream processing systems include scaling operators up [137], splitting and deploying streaming queries separately [76], scaling stateful operators out [36], and limited-overhead migration protocols [123]. The disadvantage of these approaches is that they require the presence of additional resources. An alternative is the use of load shedding that reduces the precision of the result but can be constrained to limited resources.

Babcock et al. [14] and Tatbul et al. [150] introduced load shedding to stream processing systems. Babcock et al. [14] assumed that multiple queries share the same input sources and intermediate results, and each query is a single node. The resulting topology is a union of multiple query trees. Babcock et al. [15] optimized their load shedding algorithm based on prior known data input distributions to ensure that each query does not exceed an acceptable error. This work is different from ours in multiple key aspects. First, the assumption of a single CPU means that load shed anywhere speeds up the system. This does not hold for distributed stream processors. Second, the assumption of a known data input distribution does not work well with general-purpose stream processing. Finally, the lack of parallelization leads to different topologies. Tatbul et al. [150] introduced load shedding to Aurora. Their work shares most assumptions with the work by Babcock et al. Unlike Babcock et al., they assume that the user provides piecewise functions that relate the drop of records at one operator with a similar loss in importance.

Various studies investigated the optimization of a single load shedder for specific scenarios. Chi [45] describe a single load shedder that is optimized for data mining over multiple input streams. Its algorithm increases the likelihood that events in the input streams are sampled successfully. Srivastava and Widow[143] develop a load shedder specific to joining data streams. Tatbul and Zdonik [151] present a load shedder for sampling data over time windows. Nehme and Rundensteiner [111] developed a load shedder for spatio temporal data. Complex operator logic can lead to a complex relationship between processing time and data records. Rivetti et al. [131]

developed a solution that collects information on processing times with the aid of sketches, and then samples the incoming stream accordingly.

Few studies covered load shedding techniques in light of highly parallel stream processors. Quoc et al. [127] developed a load shedder that uses stratified sampling to ensure that infrequent appearing records appearing in the stream are not dropped by the load shedder. Zhao et al. [161] developed an approach that decides what records are shed based on a tasks' internal state. These studies discuss how load should be shed at a given location, while our approach studies where and by how much records should be dropped.

The variety of mitigation methods inspired the use of general-purpose optimizers. Dhalion [66] uses a pattern recognition system, where each mitigation strategy maps to a signal, and successful mitigation reinforces its use. DS2 [89] introduced instrumentation to measure the true processing time of tasks and scale tasks accordingly. The work is similar to ours in that we, too, estimate the rate at which tasks avoid the creation of backpressure. Unlike Dhalion and DS2, we focus on the influence on the task graph's connectivity patterns and use the insight to guide the load shedding decision.

7.7 Summary

Event processing and stream processing, in general, are essential methods for the IIoT to implement applications relying on fast feedback cycles. IIoT server clusters are located at the edge of the internet, where resources are limited. Sudden bursts of generated data, software, and hardware failures can lead to bottlenecks that can significantly slow downstream processors. This slowdown can put IIoT applications at risk. In this chapter, we have developed MERA, a load shedding system that applies to highly parallel stream processing systems. At the cost of a loss in precision, it can limit the increase of end-to-end latency and the reduction of throughput. Our experiments show that MERA can lower end-to-end latency by more than a magnitude and increase the output rate by 23% compared to vanilla Apache Flink. It strikes a balance between wasted work and data loss and improves on methods that exclusively focus on either. Our load shedding system is the first to our knowledge that applies to the topologies of highly parallel stream processing systems and takes the collateral effect of backpressure into account.

Future work can extend our work along at least two dimensions. First, it can add a prediction component to estimate the behavior of complex operator logic. We described this challenge in Subsection 7.3.4. Second, it can adapt the controller to enable semantic load shedding on a program level. We briefly discussed this topic in Subsection 7.3.3.

8 Conclusion

The Industrial Internet of Things (IIoT) promises increased automation across industries, leading to faster, cheaper, more energy-efficient, and more customizable industrial processes. This advancement relies on the analysis of low-level large-volume sensor data and the quick adaptation of industrial processes using IIoT applications. Analyses are often processing-heavy and require data from multiple industrial sites for machine-learning methods to work efficiently. Unfortunately, IIoT big data is stored at industrial facilities with limited processing power that are connected by low-bandwidth access networks. This creates an asymmetry between data generated and largely stored at the edge and abundant processing power readily available in the cloud. IIoT big data requires a new architecture to store and process data and new transfer optimizations that reduce the data before it is transferred. In this work, we propose such an architecture and introduce three case studies of data-driven transfer optimizations.

8.1 Summary

IIoT big data is often of simultaneous interest to multiple user groups (facility owners, states, end consumers, etc.) and different applications built on different frameworks (e.g., stream processing systems and databases). These different uses of the data require different forms and degrees of aggregation. Chapter 3 introduces an architecture that simultaneously accommodates different forms of aggregations. This architecture introduced novel computing primitives to flexibly handle aggregation across data hierarchies and data-driven transfer optimizations to reduce the transfer cost between individual components. We expand upon the data-driven transfer optimizations in the remainder of the thesis in the form of three case studies.

The first case study discusses the use of proactive data replication for database-like systems that provide access to their data both by query and by replication of data partitions. We introduce online replication strategies that exploit past access behavior to decide whether a data partition should be proactively replicated. We compare various strategies based on traces obtained from a real-world database. Chapter 4 introduces a setting in which data partitions are accessed over a single time window. We introduce a set of strategies that continuously re-evaluate their replication decision based on the current access records. Among others, we present a strategy based on the competitive ski-rental algorithm [92]. It outperforms the strategy of shipping query results by 18% and 51% and the strategy of replicating everything by 37% and 43% for both traces. Chapter 5 changes this setting to include multiple consecutive time windows. This change allows for new strategies that estimate a pattern from access records of previous time windows and applies it to the current time window. We compare heuristic, competitive, machine learning-based, and hybrid strategies. Our results show that the competitive strategy reduces the data transfer volume by 22% on average and up to 50% at most. The hybrid strategy improves this result by another 3% on average.

The second case study in Chapter 6 studies multi-phase Distributed Top-K algorithms as an example of a synchronous messaging algorithm. We assume a common data distribution, the power-law distribution, and investigate how we can use the property of the data to reduce the data transfer volume and the number of phases. To reduce the data transfer volume further, we exploit the tolerance of distributed top-k users towards slight errors in the result. Our investigation produces an algorithm that reduces the number of phases to 2 and simultaneously achieves a reduction in data transfer volume of about 13% compared to a state-of-the-art 3-phase algorithm and about 5% compared to a state-of-the-art 4-phase algorithm for a real-world workload.

The final case study in Chapter 7 focuses on stream processing systems in resource-constrained environments. Dataflow systems process data streams by moving them through a chain of operators. The advantage of this approach is that it combines a simple programming model with an easily scalable execution model. A common challenge to dataflow systems is the performance deterioration of individual operators. When these operators process records slower than they receive them, they cause backpressure that slows down earlier operators. Unmitigated, this can lead to a chain reaction that slows down the whole program and drastically increases end-to-end latency. The common solution is to scale operators in response to backpressure. Unfortunately, this reaction is not available for resource-constrained environments such as clusters located at industrial facilities. In these circumstances, the only general-purpose solution is load shedding. We show that load shedding can reduce end-to-end latency by almost two magnitudes and the stream processing system's output rate by 23% for some cases. Naïve load shedding approaches have the disadvantage that they shed either too much load (data loss) or drop records that are the product of costly computation (wasted work). We propose an approach that exploits patterns in the dataflow graph to balances data loss and wasted work.

8.2 Future Work

As the IIoT continues to take shape, a small number of data processing frameworks will emerge that satisfy the demands of most use cases. These frameworks will have a deciding influence on the data storage and processing architecture of the IIoT. Our architecture can serve as an inspiration and guideline for this development, but it will have to be adjusted according to the communication patterns of the dominant data processing frameworks. Similarly, future frameworks may use different communication patterns or use combinations of the communication patterns that we discussed (e.g., a mix of data stream processing and data batch processing in response to outside events). Still, we believe that future work may find valuable insights in our case studies that can be extended to other frameworks. Here, we discuss how future work can build on each individual case study.

Our investigation on the trade-off between querying and replicating data offers multiple promising angles for extensions. For reasons of simplicity, we did not account for queries such as joins that require multiple data partitions to be present at the same location. This would add another level of complexity. Alternatively, one can study how more topologies of multiple front-ends and back-ends influence the performance of replication strategies. An important consideration for this question is whether it is more useful to join data access traces across locations or if each front-end to back-end pair is better optimized in isolation. A third interesting perspective is to investigate the size of partitions and the relationship between partitions. Performance can be improved by using more complex partitioning mechanisms.

In our work on distributed top-k, we assumed that the data distribution follows the power-law. This distribution frequently occurs in nature and human behavior. Still, there may be cases where the data deviates from this distribution. Future work can expand on our work by learning the data distribution. For long-running systems, this learning process can take place over multiple iterations of the distributed top-k. Tracking concept drift can be an additional valuable extension. Alternatively, future work may extend our methods to related algorithms such as heavy-hitter algorithms [160].

Orthogonal to our work on load shedding for dataflow systems, related work has designed various semantic load shedding methods that reduce the loss of data quality. Zhao et al. [161], for example, drop records based on their relation to the internal state, and Quoc et al. [127] drop records based on their group membership. While our record-agnostic solution operates over the full dataflow program, these methods usually operate at the operator-level. Joining both approaches can lead to a load shedding solution that minimizes data quality loss simultaneously with wasted work and data loss. Chapter 7 has outlined a starting point for this endeavor.

Apart from the extensions of the case studies, future work can investigate how the insights gained in our architecture can be transferred to existing IIoT architectures such as the OPC Unified Architecture [100]. Additionally, while we have limited our discussions to IIoT scenarios, insights gained in this area may be transferable to other areas. One area that may profit from our results can be augmented reality applications that are enabled by micro data centers. Similar to IIoT, this area requires careful optimization of data exchanges between micro data centers and the cloud.

8.3 Concluding Remarks

In this thesis, we have pointed out that access networks connecting industrial facilities to the cloud can become bottlenecks that hinder the deployment of IIoT applications. This bottleneck and the resource limitations at industrial facilities limit the extent to which big data approaches can be transferred to IIoT big data. We designed a software solution that reduces both effects. Chapter 3 introduces an architecture that processes inherently hierarchical and distributed data for diverse IIoT applications. It uses novel computing primitives to exchanges and combines data across hierarchies and data-driven transfer optimizations to reduce the data that is exchanged. This thesis primarily focuses on data-driven transfer optimizations. We investigate three case studies with different data transfer patterns: database-like query-results exchanges & data replication, synchronous message exchanges, and dataflow stream processing. In each case study, we replace static worst-case optimized mechanisms with data-driven mechanisms. Our solutions for these case studies exploit patterns in the data distribution, data access, and resource utilization to reduce the data transfer volume, the number of round-trips, and the effect of bottlenecks, respectively. This work provides methods to limit the impact of the network bottleneck on IIoT applications.

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, et al. “Aurora: a data stream management system”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 666–666.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. “Best position algorithms for top-k queries”. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 495–506.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803.

[4] S. Albers. “Online Algorithms: A Survey”. In: *Mathematical Programming* 97.1-2 (2003), pp. 3–26.

[5] Amazon Web Services. *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta*. 2006. URL: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/ (visited on Sept. 9, 2020).

[6] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[7] AMFG. *40+ 3D Printing Industry Stats You Should Know [2020]*. 2020. URL: https://amfg.ai/2020/01/14/40-3d-printing-industry-stats-you-should-know-2020/ (visited on Oct. 29, 2020).

[8] P. M. Apers. “Data Allocation in Distributed Database Systems”. In: *ACM Transactions on Database Systems (TODS)* 13.3 (1988), pp. 263–304.

[9] K. Arabi. “Mobile computing opportunities challenges and technology drivers”. In: *Proc. Design Autom. Conf.* 2014.

[10] K. Ashton. *That ‘Internet of Things’ Thing | RFID JOURNAL*. [Online; accessed 24. Jan. 2021]. June 2009. URL: https://www.rfidjournal.com/that-internet-of-things-thing.

[11] M. Aslett. *What we talk about when we talk about NewSQL*. 2011. URL: https://blogs.451research.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/ (visited on Nov. 10, 2020).

[12] B. Awerbuchy, Y. Bartalz, and A. Fiatx. “Competitive Distributed File Allocation”. In: *Theory of Computing* (1993), pp. 164–174.

[13] A. Aziz, O. Schelén, and U. Bodin. “A Study on Industrial IoT for the Mining Industry: Synthesized Architecture and Open Research Directions”. In: *IoT* 1.2 (2020). Publisher: Multidisciplinary Digital Publishing Institute, pp. 529–550.

[14] B. Babcock, M. Datar, and R. Motwani. “Load shedding techniques for data stream systems”. In: *Proceedings of the 2003 Workshop on Management and Processing of Data Streams*. Vol. 577. Citeseer, 2003.

[15] B. Babcock, M. Datar, and R. Motwani. “Load shedding for aggregation queries over data streams”. In: *Proceedings. 20th international conference on data engineering*. IEEE, 2004, pp. 350–361.

[16] B. Babcock, M. Datar, and R. Motwani. “Load shedding in data stream systems”. In: *Data Streams*. Springer, 2007, pp. 127–147.

[17] B. Babcock and C. Olston. “Distributed top-k monitoring”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 28–39.

[18] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, et al. “Retrospective on aurora”. In: *The VLDB Journal* 13.4 (2004), pp. 370–383.

[19] W.-T. Balke and W. Kießling. “Optimizing multi-feature queries for image databases”. In: *Proc. of the Intern. Conf. on Very Large Databases*. 2000.

[20] J. Barr. *Amazon introduces Lambda, Containers at AWS re:Invent*. 2006. URL: `https://aws.amazon.com/blogs/aws/amazon_ec2_beta/` (visited on Nov. 10, 2020).

[21] Y. Bartal, A. Fiat, and Y. Rabani. “Competitive Algorithms for Distributed Data Management”. In: *Journal of Computer and System Sciences* 51.3 (1995), pp. 341–358.

[22] C. Baur and D. Wee. “Manufacturing’s next act”. In: *McKinsey & Company* (Feb. 2016). [Online; accessed 24. Jan. 2021]. URL: `https://www.mckinsey.com/business-functions/operations/our-insights/manufacturings-next-act#`.

[23] C. Beeley and S. R. Sukhdeve. *Web Application Development with R Using Shiny: Build stunning graphics and interactive data visualizations to deliver cutting-edge analytics*. Packt Publishing Ltd, 2018.

[24] D. S. Berger. “Towards Lightweight and Robust Machine Learningfor CDN Caching”. In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 2018. DOI: `10.1145/3286062.3286082`.

[25] D. L. Black and D. D. Sleator. *Competitive Algorithms for Replication and Migration Problems*. Carnegie-Mellon University. Department of Computer Science, 1989.

[26] M. Boissier and K. Daniel. “Workload-Driven Horizontal Partitioning and Pruning for Large HTAP Systems”. In: *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. Apr. 2018, pp. 116–121. DOI: `10.1109/ICDEW.2018.00026`.

[27] M. Boissier, C. A. Meyer, T. Djürken, J. Lindemann, K. Mao, P. Reinhardt, T. Specht, T. Zimmermann, and M. Uflacker. “Analyzing Data Relevance and Access Patterns of Live Production Database Systems”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 2473–2475. DOI: `10.1145/2983323.2983336`.

[28] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson. “The industrial internet of things (IIoT): An analysis framework”. In: *Computers in Industry* 101 (2018), pp. 1–12.

[29] T. Bradicich. *What the intelligent edge is and why it’s useful*. en. Nov. 2018. URL: `https://www.hpe.com/us/en/insights/articles/the-intelligent-edge-what-it-is-what-its-not-and-why-its-useful-1711.html` (visited on Apr. 10, 2022).

[30] L. Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32. DOI: `10.1023/A:1010933404324`.

[31] N. Bruno, L. Gravano, and A. Marian. “Evaluating top-k queries over web-accessible databases”. In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, pp. 369–380.

[32] Bundesministerium für Bildung und Forschung. *Industrie 4.0.* [Online; accessed 24. Jan. 2021]. Dec. 2018. URL: `https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html`.

[33] P. Cao and Z. Wang. "Efficient top-k query calculation in distributed networks". In: *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM. 2004, pp. 206–215.

[34] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[35] M. Cardullo and W. Parks. *Transponder apparatus and system.* US Patent 3,713,148. Jan. 1973.

[36] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. "Integrating scale out and fault tolerance in stream processing using operator state management". In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 2013, pp. 725–736.

[37] S. Cavalieri and S. Mulè. "Interoperability between OPC UA and oneM2M". In: *Journal of Internet Services and Applications* 12.1 (Dec. 2021), p. 13. ISSN: 1869-0238. DOI: `10.1186/s13174-021-00144-9`. (Visited on Apr. 18, 2022).

[38] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, and A. Rasin. "The aurora and borealis stream processing engines". In: *Data Stream Management*. Springer, 2016, pp. 337–359.

[39] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. "Approximate query processing using wavelets". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 10.2-3 (2001), pp. 199–223.

[40] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[41] S. Chaudhuri and V. Narasayya. "Self-Tuning Database Systems: A Decade of Progress". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 3–14.

[42] S. Chaudhuri and V. R. Narasayya. "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server". In: *VLDB*. Vol. 97. 1997, pp. 146–155.

[43] P. P.-S. Chen. "The Entity-Relationship Model—toward a Unified View of Data". In: *ACM Transactions on Database Systems* 1.1 (Mar. 1976), pp. 9–36. ISSN: 0362-5915. DOI: `10.1145/320434.320440`.

[44] S. Cherry. "Edholm's law of bandwidth". In: *IEEE spectrum* 41.7 (2004), pp. 58–60.

[45] Y. Chi, H. Wang, and P. S. Yu. "Loadstar: Load shedding in data stream mining". In: *Proceedings of the 31st international conference on Very large data bases*. Citeseer, 2005, pp. 1302–1305.

[46] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: `10.1145/362384.362685`.

[47] E. F. Codd, S. B. Codd, and C. T. Salley. "Providing OLAP to user-analysts: An IT Mandate". In: *White Paper of Arbor Software Corporation* (1993).

[48] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[49] A. Correa, M. R. Walter, L. Fletcher, J. Glass, S. Teller, and R. Davis. "Multimodal Interaction with an Autonomous Forklift". In: *Proceedings of the 5th ACM/IEEE International Conference on Human-robot Interaction*. HRI '10. event-place: Osaka, Japan. Piscataway, NJ, USA: IEEE Press, 2010, pp. 243–250. ISBN: 978-1-4244-4893-7. URL: `http://dl.acm.org/citation.cfm?id=1734454.1734550` (visited on Feb. 4, 2019).

[50] Y. Cui and J. Widom. “Lineage tracing for general data warehouse transformations”. In: *The VLDB JournalThe International Journal on Very Large Data Bases* 12.1 (2003), pp. 41–58.

[51] M. J. da Cunha, M. B. de Almeira, R. F. Fernandes, and R. S. Carrijo. “Proposal for an IoT architecture in industrial processes”. In: *2016 12th IEEE International Conference on Industry Applications (INDUSCON)*. IEEE, 2016, pp. 1–7.

[52] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[53] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *ACM SIGOPS operating systems review*. Vol. 41. 6. 2007, pp. 205–220.

[54] Deloitte University Press. *Inside the Internet of Things (IoT)*. 2015. URL: `https://www2.deloitte.com/content/dam/insights/us/articles/iot-primer-iot-technologies-applications/DUP_1102_InsideTheInternetOfThings.pdf` (visited on Oct. 29, 2020).

[55] B. A. Devlin and P. T. Murphy. “An architecture for a business and information system”. In: *IBM systems Journal* 27.1 (1988), pp. 60–80.

[56] L. W. Dowdy and D. V. Foster. “Comparative Models of the File Assignment Problem”. In: *ACM Computing Surveys (CSUR)* 14.2 (1982), pp. 287–313.

[57] G. F. Estabrook and R. C. Brill. “The theory of the TAXIR accessioner”. In: *Mathematical Biosciences* 5.3-4 (1969), pp. 327–340.

[58] K. P. Eswaran. “Placement of Records in a File and Fileallocation in a Computer Network”. In: *Proc. IFIPS Conference*. 1974, pp. 304–307.

[59] E. Evans. *NoSQL: What’s in a name?* 2009. URL: `http://blog.sym-link.com/posts/2009/30/nosql_whats_in_a_name/` (visited on Nov. 10, 2020).

[60] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. “Spinning Fast Iterative Data Flows”. In: *Proceedings of the VLDB Endowment* 5.11 (2012).

[61] R. Fagin. “Combining fuzzy information from multiple systems”. In: *Journal of Computer and System Sciences* 58.1 (1999), pp. 83–99.

[62] R. Fagin, A. Lotem, and M. Naor. “Optimal aggregation algorithms for middleware”. In: *Journal of Computer and System Sciences* 66.4 (2003), pp. 614–656.

[63] Q. Fang and G. Yang. “Efficient Top-k Query Processing Algorithms in Highly Distributed Environments”. In: *Journal of Computers* 9.9 (2014), pp. 2000–2006.

[64] Q. Fang, Y. Zhao, G. Yang, B. Wang, and W. Zheng. “Best Position Algorithms for Top-k Query Processing in Highly Distributed Environments”. In: *Networking and Distributed Computing (ICNDC), 2010 First International Conference on*. IEEE. 2010, pp. 397–401.

[65] A. Fiat. *Online Algorithms: The State of the Art (Lecture Notes in Computer Science)*. Springer, 1998.

[66] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. “Dhalion: self-regulating stream processing in heron”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1825–1836.

[67] M. J. Flynn. “A prospectus on integrated electronics and computer architecture”. In: *Proceedings of the November 7-10, 1966, fall joint computer conference*. 1966, pp. 97–103.

[68] H. Fujiwara and K. Iwama. “Average-Case Competitive Analyses for Ski-Rental Problems”. In: *Algorithmica* 42.1 (2005), pp. 95–107.

[69] B. Gavish and O. R. Liu Sheng. "Dynamic File Migration in Distributed Computer Systems". In: *Communications of the ACM* 33.2 (1990), pp. 177–189.

[70] GE Digital. *Everything you need to know about IIoT | GE Digital.* [Online; accessed 24. Jan. 2021]. May 2017. URL: `https://www.ge.com/digital/blog/everything-you-need-know-about-industrial-internet-things`.

[71] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43.

[72] D. Giouroukis, A. Dadiani, J. Traub, S. Zeuch, and V. Markl. "A survey of adaptive sampling and filtering algorithms for the internet of things". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 27–38.

[73] Google Cloud. *Dataflow | Google Cloud.* [Online; accessed 5. Feb. 2021]. Nov. 2020. URL: `https://cloud.google.com/dataflow#section-5`.

[74] T. Groothuyse, S. Sivasubramanian, and G. Pierre. "Globetp: Template-Based Database Replication for Scalable Web Applications". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 301–310.

[75] GSMA. *Data demand explained.* 2018. URL: `https://www.gsma.com/spectrum/wp-content/uploads/2015/06/GSMA-Data-Demand-Explained-June-2015.pdf` (visited on Oct. 29, 2020).

[76] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. "Streamcloud: An elastic and scalable data streaming system". In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012). Publisher: IEEE, pp. 2351–2365.

[77] A. Handy. *Amazon introduces Lambda, Containers at AWS re:Invent.* 2014. URL: `https://sdtimes.com/amazon/amazon-introduces-lambda-containers/` (visited on Nov. 10, 2020).

[78] J. O. Hauglid, N. H. Ryeng, and K. Nørvåg. "DYFRAM: Dynamic Fragmentation and Replica Management in Distributed Database Systems". In: *Distributed and Parallel Databases* 28.2-3 (2010), pp. 157–185.

[79] Hazelcast. *Hazelcast IMDG Reference Manual.* published: 2019-04-25, accessed: 2019-05-05. Apr. 25, 2019. URL: `https://docs.hazelcast.org/docs/latest-dev/manual/html-single/#consistency-and-replication-model` (visited on May 5, 2019).

[80] C. Hewitt, P. Bishop, and R. Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[81] J. Holusha. "GENERAL MOTORS: A GIANT IN TRANSITION". In: *New York Times* (Nov. 1982). ISSN: 0362-4331. URL: `https://www.nytimes.com/1982/11/14/magazine/general-motors-a-giant-in-transition.html`.

[82] W. W. Hsu, A. J. Smith, and H. C. Young. "I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks—an Analysis at the Logical Level". In: *ACM Transactions on Database Systems* 26.1 (Mar. 2001), pp. 96–143. ISSN: 0362-5915. (Visited on May 5, 2019).

[83] R. Ihaka and R. Gentleman. "R: a language for data analysis and graphics". In: *Journal of computational and graphical statistics* 5.3 (1996), pp. 299–314.

[84] Industrial Internet Consortium. *The Industrial Internet of Things Volume G1: Reference Architecture Version 1.9.* June 2019.

[85] Industry Internet Consortium. *About Us | Industry IoT Consortium.* URL: `https://www.iiconsortium.org/about-us.htm` (visited on Apr. 17, 2022).

[86] Industry IoT Consortium. *The Industrial Internet Reference Architecture v 1.9 | Industry IoT Consortium.* URL: `https://www.iiconsortium.org/IIRA.htm` (visited on Nov. 14, 2021).

[87] I. ISO. “9075: 1987, Database languages–SQL”. In: *International Organization for Standardization* (1987).

[88] Jonny Williamson. *Industrial robots: Global sales hit record $16.5bn.* 2019. URL: `https://www.themanufacturer.com/articles/industrial-robots-global-sales-hit-record-16-5bn/#:~:text=The%20latest%20International%20Federation%20of,compared%20to%20the%20previous%20year.` (visited on Oct. 29, 2020).

[89] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. “Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows”. In: *13th ${$USENIX$}$ Symposium on Operating Systems Design and Implementation (${$OSDI$}$ 18).* 2018, pp. 783–798.

[90] J. Karimov, T. Rabl, and V. Markl. “AJoin: ad-hoc stream joins at scale”. In: *Proceedings of the VLDB Endowment* 13.4 (2019), pp. 435–448.

[91] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. “Empirical Studies of Competitve Spinning for a Shared-Memory Multiprocessor”. In: *ACM SIGOPS Operating Systems Review.* Vol. 25. 5. 1991, pp. 41–55. DOI: `10.1145/121133.286599`.

[92] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. “Competitive Snoopy Caching”. In: *Algorithmica* 3.1-4 (1988), pp. 79–119. DOI: `doi.org/10.1007/BF01762111`.

[93] A. Khanafer, M. Kodialam, and K. P. Puttaswamy. “The Constrained Ski-Rental Problem and its Application to Online Cloud Cost Optimization”. In: *Proceedings of IEEE INFOCOM.* 2013, pp. 1492–1500.

[94] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. “SageDB: A Learned Database System”. In: *CIDR.* 2019.

[95] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. “The case for learned index structures”. In: *Proceedings of the 2018 International Conference on Management of Data.* 2018, pp. 489–504.

[96] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn. “Digital Twin in manufacturing: A categorical literature review and classification”. In: *IFAC-PapersOnLine* 51.11 (2018), pp. 1016–1022.

[97] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. “Twitter heron: Stream processing at scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM. 2015, pp. 239–250.

[98] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. “SQL Server Column Store Indexes”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’11. Athens, Greece: Association for Computing Machinery, 2011, pp. 1177–1184. ISBN: 9781450306614. DOI: `10.1145/1989323.1989448`.

[99] J. Lebers. *Industrial Internet.* Nov. 2012. URL: `https://www.technologyreview.com/2012/11/28/114725/general-electric-pitches-an-industrial-internet`.

[100] S.-H. Leitner and W. Mahnke. “OPC UA–service-oriented architecture for industrial applications”. In: *ABB Corporate Research Center* 48 (2006), pp. 61–66.

[101] M. Zaharia and M. Chowdhury and M. J. Franklin and S. Shenker and I. Stoica. “Spark: Cluster Computing with Working Sets”. In: *USENIX conference on Hot topics in cloud computing.* 2010.

[102] D. Mc Hugh. *Volkswagen to network factories in the cloud with Amazon.* published: 2019-03-27, accessed: 2019-04-17. Mar. 27, 2019. URL: `https://www.reuters.com/article/us-volkswagen-amazon-cloud/vw-to-improve-production-with-amazon-cloud-to-network-its-factories-idUSKCN1R80RZ` (visited on Apr. 17, 2019).

[103] B. Melzer. *Reference Architectural Model Industrie 4.0 (RAMI 4.0)*. en.

[104] S. Michel, P. Triantafillou, and G. Weikum. “Klee: A framework for distributed top-k query algorithms”. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 637–648.

[105] Microsoft Dynamics 365. *2019 Manufacturing Trends Report*. 2019. URL: `https://info.microsoft.com/rs/157-GQE-382/images/EN-US-CNTNT-Report-2019-Manufacturing-Trends.pdf` (visited on Oct. 29, 2020).

[106] G. Moore. “Moore’s law”. In: *Electronics Magazine* 38.8 (1965), p. 114.

[107] H. Munz and E. Barnstedt. *Architecture Alignment and Interoperability: An Industrial Internet Consortium and Plattform Industrie 4.0 Joint Whitepaper*. Dec. 2017.

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

[109] N. Duffield, C. Lund, and M. Thorup. “Estimating Flow Distributions from Sampled Flow Statistics”. In: *ACM SIGCOMM*. 2003.

[110] Navstar GPS Operations. *USNO NAVSTAR Global Positioning System*. 1997. URL: `https://web.archive.org/web/20110126200746/http://tycho.usno.navy.mil/gpsinfo.html` (visited on Nov. 10, 2020).

[111] R. V. Nehme and E. A. Rundensteiner. “ClusterSheddy: Load shedding using moving clusters over spatio-temporal data streams”. In: *International Conference on Database Systems for Advanced Applications*. Springer, 2007, pp. 637–651.

[112] M. E. Newman. “Power laws, Pareto distributions and Zipf’s law”. In: *Contemporary physics* 46.5 (2005), pp. 323–351.

[113] A. Niculescu-Mizil and R. Caruana. “Predicting good probabilities with supervised learning”. In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 625–632. DOI: `10.1145/1102351.1102430`.

[114] Nidhi Singh. *Here’s the World’s First Country to Launch 5G Services*. 2019. URL: `https://www.entrepreneur.com/article/331801` (visited on Nov. 10, 2020).

[115] Nielsen Norman Group. *Nielsen’s Law of Internet Bandwidth*. 2019. URL: `https://www.nngroup.com/articles/law-of-bandwidth/` (visited on Oct. 29, 2020).

[116] C. Null and B. Caulfield. *Fade To Black The 1980s vision of "lights-out" manufacturing, where robots do all the work, is a dream no more. - June 1, 2003*. [Online; accessed 18. Jan. 2021]. Jan. 2021. URL: `https://web.archive.org/web/20091123102010/https://money.cnn.com/magazines/business2/business2_archive/2003/06/01/343371/index.htm`.

[117] oneM2M. *oneM2M The IoT Standard – Using oneM2M*. URL: `https://onem2m.org/using-onem2m/developers/basics` (visited on Apr. 17, 2022).

[118] OPC Foundation. *OPC 10000-13, OPC Unified Architecture, Part 13: Aggregates, Release 1.04*. Nov. 2017.

[119] OPC Foundation. *Unified Architecture*. en-US. URL: `https://opcfoundation.org/about/opc-technologies/opc-ua/` (visited on Apr. 17, 2022).

[120] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer Science & Business Media, 2011.

[121] P. Carbone and A. Katsifodimos and S. Ewen and V. Markl and S. Haridi, and K. Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38.4 (2015), pp. 28–38.

[122] M. Peshkin, J. Colgate, W. Wannasuphoprasit, C. Moore, R. Gillespie, and P. Akella. "Cobot architecture". In: *IEEE Transactions on Robotics and Automation* 17.4 (Aug. 2001), pp. 377–390. ISSN: 1042296X. DOI: `10.1109/70.954751`. (Visited on Feb. 4, 2019).

[123] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis. "Uninterruptible migration of continuous queries without operator state migration". In: *ACM SIGMOD Record* 46.3 (2017). Publisher: ACM New York, NY, USA, pp. 17–22.

[124] Platform Industrie 4.0. *Hintergrund zur Plattform Industrie 4.0*. de. URL: `https://www.plattform-i40.de/IP/Navigation/DE/Plattform/Hintergrund/hintergrund.html` (visited on Apr. 17, 2022).

[125] Pressebox. *Lufthansa Technik präsentiert "AVIATAR"*. 2017. URL: `https://www.pressebox.de/inaktiv/lufthansa-technik-ag/Lufthansa-Technik-praesentiert-AVIATAR/boxid/849380` (visited on Sept. 27, 2020).

[126] A. H. project. *Hadoop*. `https://hadoop.apache.org`. 2022. (Visited on Apr. 17, 2022).

[127] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. "StreamApprox: approximate computing for stream analytics". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM. 2017, pp. 185–197.

[128] A. Radziwon, A. Bilberg, M. Bogers, and E. S. Madsen. "The smart factory: exploring adaptive and flexible manufacturing solutions". In: *Procedia engineering* 69 (2014), pp. 1184–1190.

[129] D. Raposo, A. Rodrigues, S. Sinche, J. Sá Silva, and F. Boavida. "Industrial IoT monitoring: Technologies and architecture proposal". In: *Sensors* 18.10 (2018). Publisher: Multidisciplinary Digital Publishing Institute, p. 3568.

[130] J. Rifkin. *The Third Industrial Revolution: How Lateral Power is Transforming Energy, the Economy, and the World*. Macmillan, 2011.

[131] N. Rivetti, Y. Busnel, and L. Querzoni. "Load-aware shedding in stream processing systems". In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLVI*. Springer, 2020, pp. 121–153.

[132] D. P. Rodgers. "Improvements in multiprocessor system design". In: *ACM SIGARCH Computer Architecture News* 13.3 (1985), pp. 225–231.

[133] T. Roughgarden. "Beyond worst-case analysis". In: *Communications of the ACM* 62.3 (2019), pp. 88–96.

[134] S. J. Saidi, A. Maghsoudlou, D. Foucard, G. Smaragdakis, I. Poese, and A. Feldmann. "Exploring Network-Wide Flow Data With Flowyager". In: *IEEE Transactions on Network and Service Management* 17.4 (2020), pp. 1988–2006. DOI: `10.1109/TNSM.2020.3034278`.

[135] V. D. Scheinman. *Design of a computer controlled manipulator*. Tech. rep. Stanford Univ Calif Dept of Computer Science, 1969.

[136] D. Schel, C. Henkel, D. Stock, O. Meyer, G. Rauhöft, P. Einberger, M. Stöhr, M. A. Daxer, and J. Seidelmann. "Manufacturing service bus: an implementation". In: *Procedia CIRP* 67 (2018). Publisher: Elsevier, pp. 179–184.

[137] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. "Elastic scaling of data parallel operators in stream processing". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[138] K. Schwab. *The Fourth Industrial Revolution.* [Online; accessed 24. Jan. 2021]. Dec. 2015. URL: `https://www.foreignaffairs.com/articles/2015-12-12/fourth-industrial-revolution`.

[139] N. Semmler, G. Smaragdakis, and A. Feldmann. "Online Replication Strategies for Distributed Data Stores". In: *Open Journal of Internet Of Things (OJIOT)* 5.1 (2019), pp. 47–57. ISSN: 2364-7108.

[140] R. Serling. *The official corporate history of American Airlines, Eagle.* New York, NY: St. Martin's/Marek, 1985.

[141] D. D. Sleator and R. E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Communications of the ACM* 28.2 (1985), pp. 202–208.

[142] Smart2Zero. *Smartphones to cede CMOS image sensor market growth to cars.* 2018. URL: `https://www.smart2zero.com/news/smartphones-cede-cmos-image-sensor-market-growth-cars` (visited on Oct. 29, 2020).

[143] U. Srivastava and J. Widom. "Memory-limited execution of windowed stream joins". In: *VLDB*. Vol. 4. 2004, pp. 324–335.

[144] Statista. 2019. URL: `https://www.stl.tech/blog/bandwidth-demands-growing-incessantly-3-things-well-need-to-do-to-meet-them/#:~:text=Well,%20yes!,Traffic%20Forecast,%202017-2022` (visited on Oct. 29, 2020).

[145] Statista. *Average costs of industrial Internet of Things (IoT) sensors from 2004 to 2020.* 2020. URL: `https://www.statista.com/statistics/682846/vr-tethered-hmd-average-selling-price/` (visited on Oct. 29, 2020).

[146] Statista. *Volume of data/information created worldwide from 2010 to 2024.* 2020. URL: `https://www.statista.com/statistics/871513/worldwide-data-created/` (visited on Oct. 29, 2020).

[147] STL25. *Bandwidth Demands Growing Incessantly: 3 Things We'll Need to Do to Meet Them.* 2019. URL: `https://www.stl.tech/blog/bandwidth-demands-growing-incessantly-3-things-well-need-to-do-to-meet-them/#:~:text=Well,%20yes!,Traffic%20Forecast,%202017-2022` (visited on Oct. 29, 2020).

[148] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. "Fine-Grained Partitioning for Aggressive Data Skipping". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 1115–1126. ISBN: 9781450323765. DOI: `10.1145/2588555.2610515`.

[149] W. Sutherland. "On-line graphical specification of procedures". In: *SJCC, Boston, Mass* (1966).

[150] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. "Load shedding in a data stream manager". In: *Proceedings 2003 vldb conference*. Elsevier, 2003, pp. 309–320.

[151] N. Tatbul and S. Zdonik. "Window-aware load shedding for aggregation queries over data streams". In: *VLDB*. Vol. 6. 2006, pp. 799–810.

[152] Telegeography. *International Internet Capacity Growth Just Accelerated for the First Time Since 2015.* 2018. URL: `https://blog.telegeography.com/international-internet-capacity-growth-just-accelerated-for-the-first-time-since-2015` (visited on Oct. 29, 2020).

[153] M. Theobald, G. Weikum, and R. Schenkel. "Top-k query evaluation with probabilistic guarantees". In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30.* VLDB Endowment. 2004, pp. 648–659.

[154] C. Thompson and L. Shure. *Image Processing Toolbox: For Use with MATLAB;[user's Guide].* MathWorks, 1995.

[155] L. G. Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[156] VDI nachrichten. *Industrie 4.0: Mit dem Internet der Dinge auf dem Weg zur 4. industriellen Revolution*. [Online; accessed 24. Jan. 2021]. Apr. 2011. URL: `https://web.archive.org/web/20130304101009/http://www.vdi-nachrichten.com/artikel/Industrie-4-0-Mit-dem-Internet-der-Dinge-auf-dem-Weg-zur-4-industriellen-Revolution/52570/1`.

[157] D. W. Walker and J. J. Dongarra. “MPI: a standard message passing interface”. In: *Supercomputer* 12 (1996), pp. 56–68.

[158] O. Wolfson and S. Jajodia. “Distributed Algorithms for Dynamic Replication of Data”. In: *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1992, pp. 149–163.

[159] A. Woodruff and M. Stonebraker. “Supporting fine-grained data lineage in a database visualization environment”. In: *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 1997, pp. 91–102.

[160] K. Yi and Q. Zhang. “Optimal tracking of distributed heavy hitters and quantiles”. In: *Algorithmica* 65.1 (2013), pp. 206–223.

[161] B. Zhao, N. Q. V. Hung, and M. Weidlich. “Load Shedding for Complex Event Processing: Input-based and State-based Techniques”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1093–1104.

[162] D. Zuehlke. “SmartFactory Towards a factory-of-things”. In: *Annual Reviews in Control* 34.1 (2010), pp. 129–138.

[163] ZVEI: Die Elektroindustrie. *Industrie 4.0: Das Referenzarchitekturmodell Industrie 4.0 (RAMI 4.0)*.