Hierarchical Data Storage And Processing on the Edge of the Network

by

Seyed Hossein Mortazavi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Hierarchical Data Storage And Processing on the Edge of the Network

Seyed Hossein Mortazavi

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2020

Current wireless mobile networks are not able to support next generation applications that require low latency or produce large volumes of data that can overwhelm the network. Examples include video analysis applications, wearable devices, safety critical applications and intelligent smart city systems. The use of servers on the wide-area cloud, however, is also not an option as these applications require low response times, or involve processing of large volumes of data from many devices. To address these challenges, edge computing proposes the addition of computation and storage capabilities to the edge of the network [30, 19]. This thesis generalizes edge computing into a hierarchical cloud architecture deployed over the geographic span of a network. The vision supports scalable processing by providing storage and computation along a succession of datacenters positioned between the end device and the traditional wide area cloud datacenter. I develop a new deployment and execution platform called CloudPath based on the Function as a Service (FaaS) model that supports code and data mobility and distribution by enforcing a clear separation between computation and state. In CloudPath applications will be composed of a collection of light-weight stateless event handlers that can be implemented using high level languages, such as Java. In this thesis, I also develop a shared database abstraction called PathStore that enables transparent data access to the hierarchy of cloud and edge datacenters. PathStore supports concurrent object reads and writes on all nodes of the database hierarchy and its extension called SessionStore adds session consistency (read your own writes, monotonic reads/writes) for mobile applications. Finally, I implement a geo-distributed query engine that exploits the hierarchical structure of our eventually-consistent geo-distributed database to trade temporal accuracy (freshness) for improved latency and reduced bandwidth.

I dedicate this work to my parents, my wife, and my brother for their endless love, support and encouragement. And, to my dear friend Mohammad Salehe, who is resting in a better place ...

هَٰذَا مِن فَضْلِ رَبِّي

Acknowledgements

First, I would like to express my sincerest gratitude and most profound appreciation to my advisor, professor Eyal de Lara who continuously supported and directed me during my Ph.D. Eyal's exceptional coherence, intuition, and understanding as well as his immense patience and expert advise, are what any graduate student could wish for. Without his guidance and persistent help, this dissertation would not have been possible.

I would also like to thank prof. Yashar Ganjali and prof. Angela Demke Brown, for serving as my committee members and for their thoughtful and constructive comments and suggestions. In addition, I express my gratitude to my collaborators in various projects, Mickey Gabel, Iqbal Mohomed, Bharath Balasubramanian, and Shankaranarayanan Puzhavakath Narayanan, for their expert knowledge and constant feedback. I would also like to thank professor Jorg Liebeherr for his support while I was at the network research lab.

I thank my fellow labmates and friends at the University of Toronto: Daniyal Liaqat, James Gleeson, Alexey Khrabrov, Pegah Abed, Caleb Phillips, Carolina Simões Gomes, Ali Ramezani, Parsa Mirdehghan, Alireza Shekaramiz, Alborz Rezazadeh, Mehdi Zamani, Sadegh Davoudi, Kaveh Aasaraai, Milad Eftekhar, Kianoosh Mokhtarian for stimulating discussions, for the sleepless nights of work before deadlines, and for all the fun we have had in the past few years.

I would like to thank my family, my parents, my brother, and my lovely wife, Sana, who have patiently supported me with their everlasting love and kindness as well as their invaluable encouragement and trust. They have been my main primary for finishing this thesis, and without them, I would have quit the Ph.D. program years ago. It is impossible to thank them adequately for everything they have done for me. I am indebted to them forever.

Finally, I would like to express my profound appreciation to my life long friend, Mohammad Salehe, who tragically died in the PS752 plane crash in January 2020. Mohammad was humble, hard-working, and exceptionally intelligent. He was the most brilliant computer scientist I ever knew and was a collaborator on all of the projects in this thesis. He is missed every single day.

Contents

Chapter 1

Introduction

For the past 15 years, cloud computing has transformed the means of computing and data storage by providing scalable, efficient, flexible, and agile computing and storage services to users, applications, and businesses. However, with the number of distributed devices and sensors growing exponentially [41], centralized processing is becoming a bottleneck, as it begins to run against the limits of network bandwidth and latency [162]. Requirements and restrictions on user data privacy, as well as the cost of forwarding and processing data [137], are also limiting the use of the cloud for some applications. Current cloud computing architectures are not optimized for next-generation mobile or Internet of Things (IoT) applications that require low latency, or that produce large volumes of data in a carrier network.

Data is ever-increasingly generated at the edge of the network [1]. By Cisco's estimates[41], there would be 28.5 billion networked devices by 2022 and humans and machines will produce a total of 850 zettabytes of data by 2021 [1]. To handle this amount of data, in recent years Edge (also known as Fog) [1] computing has emerged as a new paradigm to process data and deliver services. In edge computing, computation and storage resources are geo-distributed and placed near the data source typically one-hop away, or at intermediate local datacenters [138]. These local datacenters are limited in resources, but their proximity to the data source allows more bandwidth and less latency, which can be beneficial to a whole range of applications and overcomes cloud computing limitations.

Topology

In this thesis, by *edge networks* we mean hierarchical networks comprised of a collection of datacenters, as shown in Figure 1.1. Each node in the network is a datacenter where part of the application is potentially deployed.

At the top of the network is the *cloud* datacenter, with high-performance computational and storage resources that are easy to scale. As we go down the network hierarchy, datacenters become increasingly resource-constrained, but also closer to the users and sensors that are the source of the data. At the very edge of the network are *edge nodes*: these are small datacenters, often comprised of a limited number of computationally-limited machines [133, 18]. We refer to datacenters on the path from the cloud to an edge as *core nodes*. Note we do not consider user

[1] In this thesis, we use the terms edge and fog computing interchangeably.

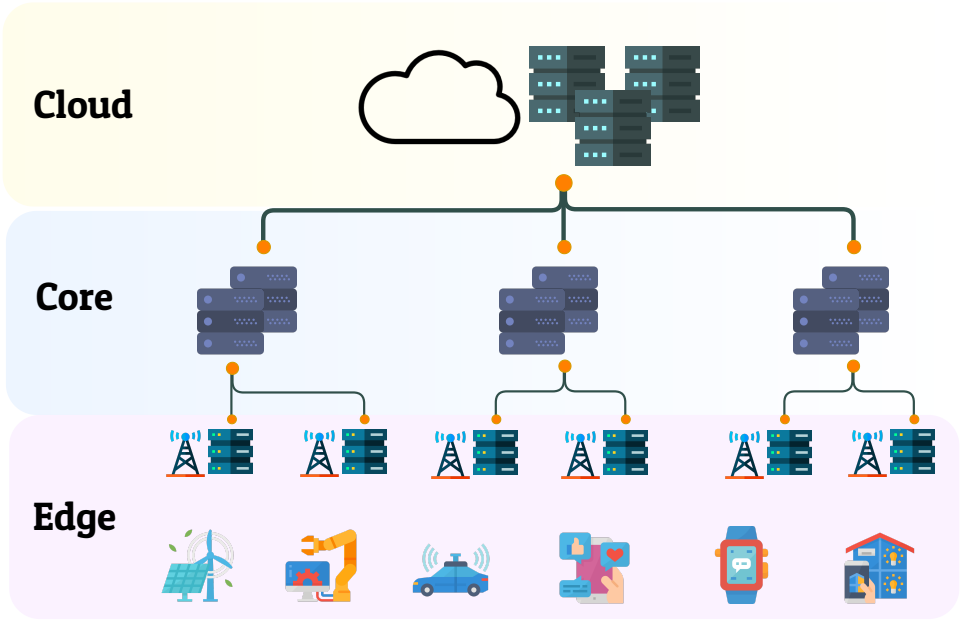


Figure 1.1: Edge computing architecture: a succession of datacenters of increasing sizes, positioned between the client device and the traditional wide-area cloud datacenter.[2]

devices or sensors as part of the network itself. Edge computing applications are applications deployed over edge networks that divide computation and storage tasks between edge, cloud, and core nodes.

Figure 1.1 illustrates how edge computing extends the traditional cloud architecture. At the top and bottom of the figure are the traditional wide-area cloud datacenter and the end-user devices, respectively. The figure also shows the path that traffic between these two endpoints follows over a collection of intermediate network links and routers.

Edge computing provides an opportunity to re-envision the way applications are created and deployed. Whereas existing applications and services may be *replicated* across wide-area cloud datacenters to improve scalability and fault tolerance, edge computing encourages *partitioning* service functionality by placing components or functions at the datacenter layer that best meets performance and security requirements. For example, a wearable smart assistant could execute latency-sensitive and bandwidth-intensive functions, such as face recognition on a nearby edge datacenter, while running infrequent and latency tolerant functions, such as user authentication and preference editing, on a traditional wide-area cloud datacenter.

[2]Icons in the figure and this dissertation are by monkik, pause08 from flaticon.com

Applications

With more bandwidth, versatility, scalability and privacy as well as less latency, some applications that benefit from novel edge computing architecture include:

Video processing and analysis [76, 94, 156]: With more bandwidth and less latency available through edge computing, video frames are processed nearby and only analytical data is sent to the cloud resulting in faster applications. For example, counting the numbers of cars that pass a certain inter-section does not require sending all video frames to the distant cloud.

IoT applications [132, 127]: With the ever-increasing use of IoT devices and sensors that are geographically distributed and produce enormous amounts of data, edge computing has emerged as a scalable middle layer between the cloud and the IoT sensors that aggregates data and provides a platform to serve real-time applications. This middle layer can store the data produced by these applications, and process queries on the data.

Healthcare and Wearable devices [35, 155, 22, 66]: As edge computing enables higher quality interactive applications, healthcare applications and assistive devices can use local secure datacenters to provide services that process data quickly and maintain the user privacy. Applications that need fast real time data processing such as in cardiac arrest prediction [147] can benefit most from this model.

Smart cities and transportation [109, 57, 163]: Due to the important issue of safety and security, roads, railways and other infrastructure in smart cities are already operating on edge computing platforms. Smart city applications require fast, reliable and secure access to data processing units and the traditional cloud is not a scalable approach for the increasing number of devices and sensors in a smart city.

Virtual and augmented reality (VR/AR) [165, 140]: Both these interactive real-time applications require massive amount of graphical rendering processing that can only be achieved in scale if there is a fast, cheap network between the rendering units and the devices. This network can be achieved in edge computing.

1.1 Challenges

While Cloud computing generally centralizes computation and storage services at certain datacenters while scaling horizontally within a datacenter, edge computing decentralizes resources to get processing units closer to data sources. This decentralization brings about challenges on how to effectively manage resources and how to provide reliable services to end-users. These challenges include:

- Partitioning and Offloading: Now that the computing and storage resources are divided onto different geo-distributed locations, how the application logic is split and provisioned onto these distributed computing resources is an open question. Some applications may benefit from aggregating data on core nodes, while some may need to split the logic/data based on the geographical location.

- Application Deployment and Execution: Unlike most applications on the cloud, different parts of edge applications can run on different datacenters. Partitioning the application and moving the application code and data to datacenters that will execute the application is a challenge the execution and deployment service has to handle. In addition, requests have to be routed to the application appropriately.

- Data Storage: Most datacenters on the edge of the network can only store and process a fraction of the data saved on the wide-area cloud nodes. At the same time, most reads and writes should be executed locally; running code close to the edge of the network has little benefit if most data accesses have to go to the cloud. In addition, data consistency for mobile applications should also be considered.

- Scheduling and Management: Resources on edge datacenters are limited, so strategies and policies on scheduling and managing these resources between competing applications have to be developed.

- Reliability and Quality of Service: Applications only choose to use edge computing if they can be offered improved services compared to the cloud. The challenge for edge computing is to ensure that applications can reliably achieve high throughput and low latency in different environments and situations.

This thesis focuses on providing solutions for the challenges above, specifically the first three challenges by providing a new model for structuring and deploying applications and managing their data. While prior studies [134, 30] only suggest frameworks to offload computation to nearby edge nodes, they do not exploit the full potential of tiered geo-distributed datacenters and generally do not manage how application data is stored, queried and processed. Our goal in this thesis is to enable edge computing more comprehensively by proposing a framework for application code deployment and execution as well as application data management. We summarize our contributions as follows:

1.2 Contributions

In this thesis we suggest a new architecture for application deployment and execution on the edge based on the serverless model. Through **Path Computing**, we generalize the edge computing model into a multi-tier cloud architecture that supports processing and storage on a progression of datacenters deployed over the geographic span of a network. This approach differentiates from the legacy edge computing models that only provide limited process offloading services to edge applications. By partitioning application data and functionality, Path Computing envisions application deployment at locations that best meet quality of service and cost requirements based on resource availability and geographic coverage. We implement the Path Computing vision by presenting **CloudPath** [116, 49]. CloudPath leverages the Function as a Service (FaaS) model to break applications into functions that are placed over distributed datacenters. CloudPath supports code and data mobility and scalability by enforcing a clear separation between computation and state. In CloudPath we develop a wide range of modules required in edge computing, including execution, deployment, storage, monitoring, and routing modules.

An essential aspect of the edge computing platform is data management and how it transforms, aggregates, and consumes data. Edge applications benefit minimally if the edge datacenters need to access a remote database to process a local request. In chapter 3, we also present **PathStore**, which is an eventually consistent data storage layer for a multi-tier cloud architecture. PathStore supports data storage on a progression of datacenters deployed from

the edge to the cloud. In PathStore, we develop a shared database abstraction that enables transparent data access across the hierarchy of cloud and edge datacenters. This storage layer supports concurrent object reads and writes on all nodes of the database hierarchy. PathStore performs all read and write operations against the local database node with data replicated dynamically between datacenters as needed. Pathstore makes possible different classes of applications, including workloads that aggregate data (such as IoT applications), or services that cache data and process information at different layers. We explain Path Computing, CloudPath, and PathStore in Chapter 3.

Many applications and devices that produce data on the edge are mobile. Their movement between datacenters raises the question of what data consistency model should be delivered to these applications. Web services and applications typically deployed on highly available storage layers such as PathStore, commonly relax consistency between replicas. A common approach for web services and applications in these scalable systems is to rely on *eventual consistency* where the storage system guarantees that if no new updates are made to an object, eventually all reads will return the last updated value [82]. To addresses issues with consistency and to enhance our storage with stronger consistency models, we present **SessionStore** [114, 111, 112, 113] that ensures session consistency on a top of otherwise eventually consistent replicas. SessionStore groups related data accesses of the database into a session and uses a session-aware re-conciliation algorithm to reconcile only the data relevant to the session when switching between replicas. We discuss SessionStore in more detail in Chapter 4.

In many edge computing scenarios, data is generated over a wide geographic area and is stored near the edges, before being pushed upstream to a hierarchy of datacenters. A key question is how to query this data and extract information. Querying such geo-distributed data traditionally falls into two general approaches: push incoming queries down to the edge where the data is, or run them locally in the cloud. **Feather** [115] is a hybrid querying scheme that exploits the hierarchical structure of such geo-distributed systems to trade temporal accuracy (freshness) for improved latency and reduced bandwidth. Rather than pushing queries to the edge or executing them in the cloud, Feather selectively pushes queries towards the edge while guaranteeing a user-supplied per-query freshness limit. Partial results are then aggregated along the path to the cloud until a final result is provided with guaranteed freshness. Feather is

designed for ad-hoc queries and supports a broad set of queries, including grouping, aggregation, and raw row retrieval.

In Chapter 5, we present Feather, and in Chapter 6, we conclude the thesis and suggest avenues for future work.

Chapter 2

Related Work

In this Chapter we study the recent state of the art research in the areas of mobile edge computing and data management and survey the evolution of these technologies.

2.0.1 Edge Computing Architectures

Early edge computing systems have relied on virtual machines (VM) as the unit of application deployment [133, 61]. These systems rely on optimizations, such as VM synthesis [67] and unikernels [105], to reduce the network traffic and deployment time. Satyanarayanan et al. [133] where among the first who emphasized on the importance of having cloud resources close to the mobile user. They argue that empirically, cloud resources can only be accessed through the network with significant delay(currently close to hundreds of milliseconds [75]) and this latency is unlikely to be reduced because of restrictions on the Wider Area Network (WAN) and propagation delay. This restricts offloading computation on Cloud nodes in a timely manner and a new model where computation is close to the mobile users is required. They present the idea of using nearby (generally one hop away) resource-rich computers or clusters in **Cloudlets** to increase the computational power of neighboring mobile devices. In their paper they use the term Cloudlet to refer to a layer of trusted computers between mobile devices and cloud servers. The devices used in Cloudlets have similar capabilities to a datacenter but on a lower scale and are in the vicinity of the mobile user. On these workstations, a virtual machine (VM) will run customized service software and some of the processing requirements of the mobile devices is migrated to these virtual machines through Wi-Fi (using LTE is later suggested). The proposed applications that could benefit from this framework include: Augmented reality, Optical Character Recognition (OCR), face recognition systems, Vehicle to Vehicle (V2V) communications and crowd sourcing applications.

The drawbacks on developing the VM based Cloudlet model includes relying on carriers and internet service providers to deploy this architecture in LAN networks over Wi-Fi. Furthermore, VM's are not versatile units of processing that can be deployed quickly and efficiently on smaller cloudlets units as they can be resource intensive and slow. Studies such as [66], [40], [68], [134], [150], [91] present implemented frameworks and applications based on the Cloudlet architecture, and companies such as Akamai, Huawei and Dell have built micro-datacenters that are similar to the cloudlets [134] model.

More recently, several research platforms have switched to operating system containers as the unit of deployment [95, 25]. While operating system containers are smaller than VMs, they can still require the transfer of hundreds of megabytes to instantiate a container.

In contrast, ClodPath leverages a new cloud computing model known as Function as a Service (FaaS) with Serverless Computing. In Serverless computing another layer of abstraction is added to the virtualization hierarchy. Rather than sharing operating systems (containers) or hardware (virtual machines), users share the language run-time environment. This model reduces the resource footprint and response time compared to containers and virtual machines. In addition, this model has the ability to automatically and quickly scale and increase the number of workers when load increases using a orchestration service that is responsible for resource allocation, fault-tolerance, monitoring, maintenance and scalability. However, the user applications are more limited and can only operate in a limited set of pre-configured environments.

Examples of FaaS systems include Amazon Lambda [151], OpenLambda [72], IBM OpenWisk [21], Microsoft Azure functions, Google Cloud functions and AppScale [39]. In OpenLambda [72] an open source serverless system is suggested where the goal is to implement an elastic and scalable platform. OpenLambda consists of a series of subsystems that execute and maintain applications handler. Handelers are sandboxed in containers which are controlled by schedulers and load-balancers forward requests to containers.

All these systems target the wide-area cloud environment, and assume a flat replicated environment with a relatively small number of large datacenters accessible over the Internet. Our work differs in that it is the first application of FaaS to be running code on a hierarchy of datacenters stretched from the network edge to the wide-area cloud.

Path computing has similarities to previous approaches that have infused networking nodes with processing, such as active networks [32] and the intentional naming system [12]. These previous efforts, however, focused on low-level network processing (e.g., encryption, routing, load balancing), whereas CloudPath targets full server workloads.

Previous work has explored automatic application partitioning and migration [46, 40, 60]. In comparison, our approach requires application developers to explicitly partition their applications into clearly-defined functions. We argue that this approach is consistent with existing

best practices for web back-end design, which mandate the use of stateless REST functions for scalability and fault tolerance.

A complete decentralized computing platform that distributes compute, storage, and networking services closer to the mobile devices called **Fog Computing** is proposed by Bonomi et al. from Cisco systems in [30]. Fog servers are geographically distributed and close to the mobile and ubiquitous devices. Fog computing distinguishes itself from cloudlets by putting more emphasis on requirements for Internet of Things(IoT) devices. Bonomi et al. argue that clouds have difficulty achieving IoT requirements such as low latency, mobility and location awareness. Similar to Cloudlets, fog computing extends the paradigm of the Cloud to the edge networks for the Internet of Things (IoT) devices. Similar to how the fog is closer the earth than the clouds, fog servers are also closer to edge devices than cloud servers. Their solution is the Fog architecture which is composed of a number of geo-distributed heterogeneous fog nodes placed between the end devices and the cloud in different tiers. Each fog node can be composed of different devices such as routers, IoT gateways, access points as well as any computing devices which makes the edge, part of the fog system. A service orchestration layer is proposed for dynamic, policy-based life-cycle management and an abstraction layer hides the complexity of each heterogeneous node and *"provides generic APIs for monitoring, provisioning and controlling physical resources"* [29]. Applications of such an architecture include smart traffic light systems, and sensors controlling a wind fire. Alternative types of fog nodes have also been proposed including using networking devices (such as home routers) [96] and moving or parked vehicles[74] for computational processes.

2.0.2 Distributed Storage and Session Consistency

A large body of research exists about replicated databases for geographically distributed datacenters both in industry and academia [157, 136, 63, 99]. These systems offer stronger consistency models, but assume a flat overlay structure. In this project, we use Cassandra as an existing widely used system and use it as the basis for our hierarchical storage system.

Session consistency can in principle be provided by solutions that provide stronger consistency such as in [92] where all transactions are always executed at the local replica through snapshot isolation, or by geo-distributed transactional databases like Spanner [44], CockRoachDB [2]

or by systems that use mixed consistency [89] and workload management [158] to provide strong consistency only where required. Unfortunately, these solutions use variants of distributed consensus that are very expensive across the wide-area-network [13, 88] and makes them impractical for the edge. As we show in our experiments in Section 4.4, enforcing strong consistency even for a moderate number of replicas incurs large latency costs.

A better approach is to use causally consistent systems like Bayou [143], COPS [98]. In practice, however, these approaches are also not applicable to edge deployments for three reasons: (i) these approaches assume a low and fixed number of replicas, whereas popular edge services may have hundreds or thousands of replicas. Many of these systems make use of vector time stamps where the overhead grows linearly with as the replication factor increases. While there are methods to trim the vector [123, 122], a compact vector clock that implicitly assigns vector positions to nodes requires centralized arbitration (or some other method of distributed consensus). Alternatively, the vector may include a unique node identifier like an IP address. In the latter case, however, significant additional storage is required, which makes using these systems on the edge unfeasible; (ii), these approaches assume full data replication (i.e., a complete copy of the database is stored at each site). Unfortunately, the resource-limited nature of edge datacenters dictates that they are only able to store a small fraction of the total state of a service or application. These limitations require the use of on-demand partial replication where only the state that is relevant to the current users of the edge datacenter is presently replicated on the edge datacenter; and (iii), data reconciliation is not fine-grained based on client or function data, rather reconciliation is done on table granularity. This approach results in high reconciliation latency and high bandwidth consumption for the transfer. This is particularly the case when only a fraction of the data is relevant to a given client.

Providing causal consistency on top of eventual consistency has been studied in [24] and [20]. In these studies a layer between the client and the data storage layer provides causal consistency for the client using vector clocks. In addition to the discussion on causally consistent systems presented above, we note that these works do not focus on session-aware reconciliation, which is crucial for our edge scenarios.

Other solutions to session consistency in the literature [152, 7] use a combination of the following basic techniques: (i) sticky sessions can ensure that all reads and writes within a

session maintained by a client always communicate with a single replica, (ii) maintain state at the client so that when a session does change replicas, the client can service requests from its cache until the new replica is up to date, and (iii) use vector time stamps for the requests and ensure that each read or write is served or accepted at a replica in such a manner as to satisfy the session guarantees. The first approach is not applicable to edge computing scenarios where the clients switch replicas over time due to client mobility or to access functionality deployed on different datacenters. The second approach is only applicable when the client is fully trusted and has enough resources to store data. It is not practical for multi-user applications where raw data is kept at the server and is made available to clients in mediated form in response to explicit application requests. Finally, approaches that require vector time stamp break down when the number of replicas is large and dynamic as the overhead grows linearly with the replication factor. While there are methods to trim the vector [123, 122], they come at the cost of significant complexity and require strong coordination between replicas, which makes using these systems on the edge unfeasible.

In addition, various approaches have been proposed for application and service migration on the edge [118, 141, 102], however these approaches commonly depend on VM/Container migration methods or full application state synchronization. Our approach provides applications with flexible, fine grained data reconciliation through sessions. Our approach is especially advantageous for multi-user services that use the same replica to handle requests on behalf of multiple clients, where only a fraction of the replicated state is relevant to a given client.

There have been several works on data/state reconciliation in transactional databases [47, 53] and most key-value stores support some form of data reconciliation across replicas [34, 43]. However, in the former case, these solutions are heavy-weight, as necessary to guarantee ACID transactionality. The reconciliation function in common key-value stores are typically implemented as generic “stop-and-migrate” techniques that do not carefully track subsets of client data. This results in significant down times for the client. SessionStore, on the other hand, is much more light-weight since it guarantees session consistency as opposed to ACID transactionality.

2.0.3 Distributed Querying

There exist several general approaches for querying in geo-distributed settings: querying the edge nodes directly, distributed engines for query planning and execution, and stream processing. We also review existing approaches to providing and characterizing freshness guarantees.

Querying Edge Nodes

Respawn [31] is a distributed time series database that provides low latency range queries on a multi-resolution time series from edge devices. In Respawn, sensors send data to nearby edge nodes which store the data and compute aggregates at different time resolutions. Lower resolution aggregated data are periodically sent to a cloud node, and a query dispatcher on the cloud node decides on whether to send the query to the edge nodes or process it on the cloud node based on the requested resolution. Similarly, EdgeDB [159] is a time-series database for edge computing that proposes a multi-stream merging mechanism to aggregate correlated streams together at runtime.

Other approaches aggregate data closer to the devices [138, 80] or reduce bandwidth using lossy data transformations such as resampling [101]. Unlike Feather, these approaches do not provide a flexible guarantee on data freshness. Moreover, they limit the complexity of queries that can be executed, as they are limited to time series data and to queries that allow error due to the aggregation.

Distributed Query Engines

Distributed query engines such as Apache Spark SQL: [17], Apache Impala [28] or Facebook's Presto [135] process queries on top of heterogeneous data stores such as Hive [144], PostgreSQL, Hadoop, MySQL and Apache Kafka [83] among others. Presto is an open source distributed SQL query engine that receives SQL queries and enables analytic querying against data in different sources of varying size. Similarly, Spark SQL provides support querying for structured and semi-structured data. However such systems are not designed for geo-distributed and edge computing settings: they assume data is co-located or is distributed over a flat topology comprised of few cloud datacenters. In addition they do not enable querying based on freshness

requirements.

Stream Processing

In the stream processing paradigm continuous queries are described as directed acyclic graphs (DAG) of operators, which are then instantiated across the datacenters in the network. Data is processed and aggregated as it is flows from the edge towards the cloud. However, while well-established in the cloud setting, existing frameworks [148, 33, 164] have not been designed for geo-distributed settings where communication is unreliable, datacenter resources are limited, and latency between datacenters limits performance and creates flow control issues [145, 51]. Recent research extends the stream process paradigm to the edge computing settings, as generic stream processing frameworks or bespoke applications [145, 129, 48, 128, 166, 161, 168, 117]. Despite progress, stream processing is better suited for processing a small set of continuous, recurrent global queries, rather than ad-hoc queries. This is because queries must be broken down into operators in advance, and then deployed, coordinated, and executed on various datacenters across the networks – all of which have costs. Additionally, stream processing frameworks do not support create, read, update, and delete (CRUD) operations or local queries at arbitrary nodes.

Wireless Sensor Networks

Many studies have discussed the idea of storing and querying data in a set of distributed sensor node networks [62, 78, 90, 104, 160, 103]. In these studies, the network itself is the database [62] and to extract information from the network, various methods [104, 103] are proposed to aggregate and propagate data resulting from a query to a single base station. TAG [103] and TinyDB [104] provide SQL-like APIs to express declarative queries over the network and the system aggregates queries over values while considering communication and storage requirements. Feather flexible freshness guarantee can be extended to this setting, since wireless sensor networks can be organized in a communication tree.

Freshness Threshold

Google Cloud Spanner [44] and Microsoft Cosmos DB [108] also allow users to specify bounded staleness to boost performance as a feature for read queries. Spanner is not a suitable choice for edge computing, however, since it is designed for a collection of resource-rich datacenters connected by high quality links, and because it aims to provide strong consistency. When edges are disconnected, for example, local writes cannot proceed. Moreover, when links between nodes have high latency writes are prohibitively expensive since they involve writing to multiple nodes. Spanner's freshness guarantee mechanism is much simpler than Feather's: it chooses a single replica that satisfies the freshness threshold to execute the query on, relying on strong consistency. It is therefore more equivalent to executing a query on the cloud in our setting. In contrast, Feather allows local queries to proceed unhindered even when edges are not connected, and for global queries it can combine results from multiple nodes, which allows fresher answers than available on any single replica. Cosmos DB similarly executes global queries in a single replica, and does not aggregate results from multiple datacenters. As with Spanner, it is designed for resource-rich datacenters.

The trade-off between freshness, accuracy, and performance in continuous (streaming) queries was investigated by Heintz et al. [70, 69]. They propose an online algorithm that determines how much data aggregation should be performed at the edge versus the center, where windowed grouped aggregation is used to minimize both staleness and bandwidth. Conversely, Feather is designed for ad-hoc queries and supports a larger set of queries including grouping, aggregation, and raw row retrieval.

Formal Consistency Properties

Golab et al. [59] propose the Δ-atomicity property for quantifying staleness, and describe algorithms for formally verifying and quantifying it. Our freshness guarantee is similar to Δ-atomicity, and Feather can be viewed as an implementation of it for tabular data in the edge computing setting. Rahman et al. [125] propose the t-freshness property which considers when operations begin rather than end, and use it to derive CAP-style impossibility results for the trade-off of partitioning, latency, and freshness. They also describe GeoPCAP, a distributed

key-value store with probabilistic guarantees. Unlike Feather, GeoPCAP assumes a flat structure where replicas contact each other directly, which may be infeasible in large hierarchical edge networks with high latency links. Moreover, Feather is a tabular store that supports querying multiple rows, and must therefore compose results from multiple data sources.

Chapter 3

CloudPath: A Multi-Tier Cloud Computing Framework

This chapter introduces *path computing*, a generalization of edge computing into a multi-tier cloud paradigm that supports processing and storage on a progression of datacenters deployed over the geographic span of a network. Figure 3.1 illustrates how path computing extends the traditional cloud architecture. At the top and bottom of the figure are the traditional wide-area cloud datacenter and the end-user devices, respectively. Path computing enables the deployment of a multi-level hierarchy of datacenters along the path that traffic follows between these two end points. Path computing makes possible different classes of applications, including workloads that aggregate data (such as IoT applications), or services that cache data and process information at different layers. Path computing provides application developers the flexibility to place their server functionality at the locale that best meets their requirements in terms of cost, latency, resource availability and geographic coverage.

We also describe CloudPath, a new platform that implements the path computing paradigm and supports the execution of third-party applications along a progression of datacenters positioned along the network path between the end device (e.g., smartphone, IoT appliance) and the traditional wide-area cloud datacenter. CloudPath minimizes the complexity of developing and deploying path computing applications by preserving the familiar RESTful development model that has made cloud applications so successful. CloudPath is based on the key observation that RESTful *stateless* functionality decomposition is made possible by the existence of a common storage layer. CloudPath simplifies the development and deployment of path computing applications by extending the common storage abstraction to a hierarchy of datacenters deployed over the geographical span of the network.

CloudPath applications consist of a collection of short-lived and stateless functions that can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. Developers determine where their code will run by tagging their application's functions with labels that reflect the topology of the network (e.g. edge, core, cloud) or performance requirements, such as latency bounds (e.g. place handler within 10ms of mobile users). CloudPath provides a distributed eventually consistent storage service that functions use to read and store state through well-defined interfaces. CloudPath's storage service automatically replicates application state on-demand across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

We evaluated the performance of CloudPath on an emulated multi-tier deployment. Our results show that CloudPath can deploy applications in less than 4.1 seconds, has routing overhead below $1ms$, and has negligible read and write overhead for locally replicated data. Moreover, our test applications experienced reductions in response time of up to 10X when running on CloudPath compared to alternative implementations running on a wide-area cloud datacenter.

The rest of this chapter is organized as follows. Section 3.1 introduces *path computing*. Section 3.2 introduces CloudPath, a new platform that implements the path computing paradigm. Section 3.3 describes the design and implementation of our CloudPath prototype. Sections 3.4 and 3.5 present our experimental setup and the results from our evaluation. Finally, Section 3.6 concludes the chapter and discusses future work.

3.1 Path Computing

Edge computing expands the traditional flat cloud architecture into a two-tier topology that enables computation and storage at a locale close to the end user or client device. We introduce *path computing*, a generalization of this design into a multi-tier cloud architecture that supports processing and storage on a progression of datacenters deployed over the geographic span of a network.

Figure 3.1 illustrates how path computing extends the traditional cloud architecture. At the top and bottom of the figure are the traditional wide-area cloud datacenter and the end-user devices, respectively. The figure also shows the path that traffic between these two end points follows over a collection of intermediate network links and routers. Path computing enables the deployment of a multi-level hierarchy of datacenters along this path, with the traditional wide-area datacenters at the root of the hierarchy.

We refer to a datacenter that is part of the hierarchy as a *node*. Nodes along the hierarchy can differ vastly in the amount of resources at their disposal, with storage and execution capacity expected to decrease as we descend levels in the hierarchy and move closer to the end-user device. Wide-area nodes are assumed to have access to virtually limitless computation and storage; in contrast, nodes close to the edge of the network may have just a handful of servers

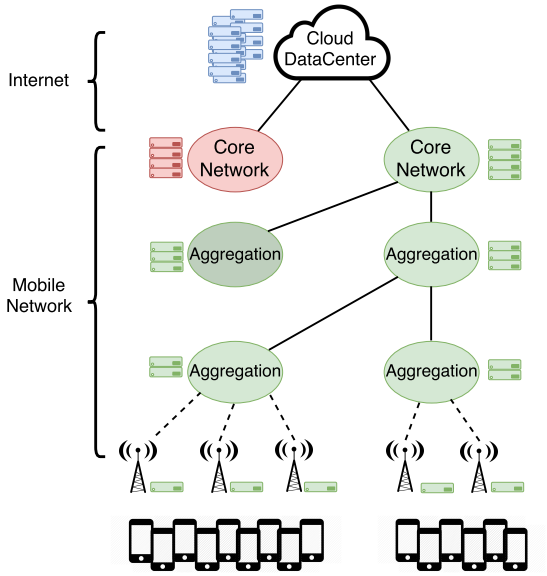


Figure 3.1: **Path Computing Architecture**: Path computing provides storage and computation along a succession of datacenters of increasing sizes, positioned between the client device and the traditional wide-area cloud datacenter.

at their disposal. The number of nodes at any given level of the hierarchy is expected to grow dramatically as we get further away from the root. For example, a path computing deployment may consist of a handful of wide-area nodes, tens of nodes running at the network-core of various mobile carriers, hundred of nodes running on region-level aggregation switches, and tens of thousands of nodes running on the edge of the network.

Path computing can be materialized in a variety of different topologies and networking technologies. For example, a simple two-tier topology that is the focus of most edge computing research, could consist of a layer of nodes running on or close to WiFi access points and a cloud layer. This simple topology could be expanded to include additional tiers inside the Internet Service Provider's network at convenient aggregation points, at the city and regional levels. Similarly, the architecture could be incorporated into mobile cellular networks. LTE networks by default encapsulate packets and send them to the network's core for processing; however, a growing number of product offerings, such as Nokia RACS gateway [75] and Huawei's Service Anchor [5], have the potential to enable in-network processing by selectively diverting packets for processing. Looking ahead, 5G, which is currently in the process of being standardized, opens the possibility for packet processing at the edge. CloudPath nodes could be incorporated

on the base station (EnodeB) or the Centralized Radio Access Network (C-RAN) [1], as well as at aggregation switches along the path to the network core and at the core itself.

3.1.1 Opportunities and Challenges

Path computing creates new opportunities for application developers. Today, mobile and IoT applications are typically developed based on the client-server model, which requires developers to partition application logic and state between a client running on the end-user device and a server located on the wide-area cloud. In contrast, path computing provides developers the opportunity to run their server-side functionality on a number of different locations making possible different classes of applications, including workloads that aggregate data (such as IoT applications), or services that cache data and process information at different layers. Path computing provides applications developers the flexibility to control the placement of their application components or tasks at the locations that best meet their requirements in terms of cost, latency, resource availability and geographic coverage.

It is generally accepted that the cost of computation and storage is inversely proportional to datacenter size [16]; therefore, it is reasonable to assume that the unit cost of deploying and managing computation and storage increases as we get closer to the edge and nodes become smaller and more numerous. Conversely, the network cost of serving a request goes down as we move closer to the edge and fewer links need to be traversed. To a first approximation, the cost of running a compute intensive task can be optimized by placing it on the datacenter node that is farthest away from the edge, but still meets the latency and hardware requirements of the task. On the other hand, the cost of a network intensive task can be optimized by running it on the datacenter node that is closest to the edge, while still meeting the task's hardware requirements (i.e., availability of a particular accelerator).

Optimal task placement may also depend on other factors such as the geographic coverage provided by a datacenter node, the size of the population it serves, and user mobility patterns. For example, the effectiveness of data reduction tasks, such as computing an average over streams of sensor data produced by a farm of IoT devices, is a complex product of the number

[1] C-RAN is a proposed architecture for future cellular networks that connects a large number of distributed low cost remote radio heads (RRH) to a centralized pool of baseband units (BBU) over optical fiber links [119].

of available incoming streams, the aggregation factor, and the cost of the computation and network bandwidth. On one hand, the network benefits of aggregation decrease as we get father away from the edge. On the other, the geographic coverage area served by a datacenter node grows as we get away from the edge creating more opportunities for data aggregation (i.e., there are more streams). Similarly, task placement affects how an application component experiences user mobility. For example, an application component running on a city-level node will experience a much lower level of user handover than one deployed on a node closer to the edge, such as WiFi access point.

Unfortunately, taking advantage of the added flexibility introduced by path computing is not easy. It requires developers to partition their server-side functionality, and manage the placement of code and data based on complex calculations that trade off proximity to the user with resource availability and cost. In addition, the limited capacity of the datacenters on the lower levels of the hierarchy puts a hard bound on the number of applications and datasets that can be hosted simultaneously requiring application code and data to be dynamically provisioned. Section 3.2 introduces CloudPath, a new platform designed to address these challenges.

3.1.2 Practical Considerations

Path computing datacenters need to be in or near the network of different ISPs or mobile network providers, so it is likely that they will be owned by the different network providers. In contrast, application developers are used to a deployment model where their application is globally available independently of the carrier used by an individual user [2]. Rather than having individual application developers negotiate service agreements with a myriad of network providers, it is likely that cloud providers (existing or new) will offer a one-stop shop that lets application developers run their code across datacenters managed by different carriers. This model follows the approach taken by content delivery network companies, such as Akamai, which let applications owners serve their content to users across different ISPs.

[2]Some carriers deploy applications that are only available to their customers on their own network, but this is a much less attractive deployment model for third-party applications.

```
face_detection_and_recognition_service {
    login(credential)->token            : any
    detect_faces(image)->coordinates[]  : 10 ms
    recognize_face(image)->label        : 50 ms
}
```

Figure 3.2: Server API with application entry points labeled with latency requirements.

3.2 CloudPath

CloudPath is a platform that implements the path computing paradigm, and supports the development and deployment of applications that run on a set of datacenters embedded over the geographical span of the network. CloudPath assumes a subscription model similar to that of existing wide-area network cloud platforms where anyone with an account on the system can deploy and run applications. In this scenario, the available applications and their data vastly outnumber the resources available at the smaller datacenters, which only have enough resources to run a limited number of applications at any time and can store only a fraction of the data. As a result, CloudPath deploys applications and replicates data on-demand.

CloudPath minimizes the complexity for developing path computing applications by preserving, as much as possible, the familiar development model that has made traditional cloud applications so successful. CloudPath builds on the observation that it is accepted practice for cloud applications to implement server-side functionality as services that are exposed to the client over an API consisting of stateless *entry points*, or functions, that are exposed as unique URIs. For example, Figure 3.2 shows a simplified server-side API for an application that performs face detection and recognition. The API includes three entry points that let the client device login and authenticate, upload an image on which to perform face detection, and upload an image of a face for recognition. The stateless nature of the entry points improves application modularity, makes it possible to dynamically scale each function independently, and increases fault tolerance.

Our key observation is that this functionality decomposition is made possible by the existence of a common storage layer. CloudPath simplifies the development and deployment of path computing applications by enforcing a clear separation between computation and state,

and expanding the common storage abstraction to a hierarchy of datacenters deployed over the geographical span of the network.

CloudPath applications consist of a collection of short-lived and stateless functions that leverage a distributed storage service that provides transparent access to application data. CloudPath functions are implemented using high level languages, such as Java or Python. Since CloudPath functions are small and stateless, they can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. CloudPath provides a distributed eventual consistent storage service that functions can use to read and store state through well-defined interfaces. CloudPath automatically migrates application state across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

Developers determine where their code will run by tagging their application's entry points (i.e., functions) with labels that reflect the topology of the network (e.g. edge, core, cloud) or performance requirements, such as latency bounds (e.g. place handler within 10ms of mobile users). For example, Figure 3.2 shows annotations that indicate that the authentication can run on any datacenter, whereas face detection and recognition need to run in a datacenter that can be reached within 10ms and 50ms of the mobile client, respectively.

CloudPath does not migrate a running function between datacenters. Instead, CloudPath supports code mobility by terminating an existing instance (optionally waiting for the current request to finish) and starting a new instance at the desired location. Similarly, for mobile users, network hand-off between cells may result in a change in the network path with traffic flowing through a different set of CloudPath datacenters. CloudPath does not migrate network connections between datacenters. Instead, it terminates existing connections and leaves it to the application to establish a new connection with the new datacenter. While this approach requires application modifications, CloudPath provides a client library that automates the re-connection process.

The next section describes the design and implementation of CloudPath in detail.

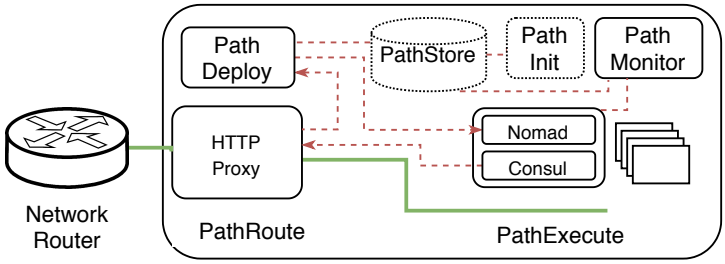


Figure 3.3: CloudPath Node Architecture. The dotted modules only belong to the cloud node. The dashed lines indicate the control path and the green line shows the data path

3.3 Design and Implementation

CloudPath organizes datacenters into a simple tree topology overlaid over a collection of underlying mobile networks and the public Internet. We refer to a datacenter that is part of the CloudPath deployment as a *node*. The CloudPath tree can have arbitrary depth, and does not have to be balanced; different branches of a CloudPath network can have different height. New nodes can be attached to any layer of the existing tree.

While simple, this structure can accommodate different classes of applications, including workloads that aggregate data (such as IoT applications), or content delivery applications that cache data at different layers. This simple topology is a natural fit to the way mobile networks are currently organized in the physical substrate, and it also simplifies routing and configuration, as a node only needs to know its parent to join the network. However, other topologies may improve fault tolerance, are more robust to failures and allow for optimizations (e.g., direct data transfer between siblings, or load-balancing between siblings). We leave the exploration of alternative designs for future work.

CloudPath nodes are expected to differ widely in the amount of resources at their disposal, with storage and execution capacity expected to decrease as we descend levels in the hierarchy and move closer to the end-user device. Irrespective of size, each CloudPath node is comprised of the following modules:

- **PathExecute**: Implements a serverless cloud container framework that supports the execution of lightweight stateless application functions.

- **PathStore**: Provides a distributed eventual consistent storage system that manages

application data across CloudPath nodes transparently. PathStore is also used internally by PathDeploy and PathRoute to fetch application code and routing information.

- **PathRoute**: This module routes requests to the appropriate CloudPath node. The user's location in the network, application preferences, and system state (e.g., application availability, load) are considered when making routing decisions.

- **PathDeploy**: Dynamically deploys and removes applications from CloudPath nodes, according to application preferences and system policies.

- **PathMonitor**: Provides live monitoring and historical analytics on deployed applications and the CloudPath nodes they are running on. Aggregates metrics from other CloudPath modules in each node, collects them using PathStore, and presents the results in a simplistic web interface.

In addition to the modules above, the root node located in the wide-area cloud also contains a module called `PathInit`. Developers upload their application to CloudPath through this module.

```
public class ClockService extends Action {
    public String getTimeZone() {
      Select s = QueryBuilder
                .select().all().from("clock");
      s.where(QueryBuilder
                .eq("userId", CurrentUserID));
      ResultSet results = pathstore.execute(s);
      Row row = rowList.results.one();
      int tzOffSet = row.getInt("tzOffset");
      return "<p>The zone is:" + tzOffSet + "</p>";
   }

    public String getPrefs() {
           ......
           ......
    }
}
```

(a) Function definition

```
<CloudPath_app>
  <mapping>
    <uri_pattern>/timeZone</uri_pattern>
    <function>ClockService.getTimeZone</function>
    <loc_pref>edge</loc_pref>
  </mapping>
  <mapping>
    <uri_pattern>/prefs</uri_pattern>
    <function>ClockService.getPrefs</function>
    <loc_pref>core</loc_pref>
  </mapping>
  <sub-domain>clockapp</sub-domain>
</CloudPath_app>
```

(b) Function registration

Figure 3.4: CloudPath application example. The application consists of two functions: `getTimeZone()` and getPrefs(). These functions are registered as CloudPath entry points(`/timeZone` and `/prefs`) by mapping the function to a URI using the `web.xml` file shown in part (b). The location where each function needs to run is also specified in this file. The full URI will include the application name and `cloudpath.com`, e.g., `clockapp.cloudpath.com/prefs`.

3.3.1 PathExecute

PathExecute implements a serverless cloud container framework that supports the execution of lightweight stateless application functions in each CloudPath node. Function as a Service (FaaS), also known as Serverless Computing, is a cloud computing approach in which the cloud provider fully manages the infrastructure used to serve requests, including the underlying virtual machines or containers, the host operating system, and the application run-time. Despite the Serverless moniker, FaaS applications do require a server to run. Serverless reflects the fact that the application owner does not need to provision servers or virtual machines for their code to run on. FaaS applications are composed of a collection of light-weight stateless functions that run on ephemeral isolated environments. We argue that the small size and stateless nature of FaaS functions make them ideal candidates for our multi-tier path computing deployment. One of the main benefits of using the serverless architecture for edge computing is that it has a small footprint as functions rather than the full applications are executed. In addition, it has provides us with the flexibility of breaking down applications into functions and deploying them on various datacenters.

Our current prototype requires functions to be implemented as Java Servlets and requests for these servlets arrive using HTTP. For each application running on a node, we spawn a separate Docker [107] Ubuntu container running Jetty web server [10]. Functions of the same application can share the same container, and the same container is reused across multiple requests; however, a container may be terminated by the framework without notice and developers should not make any assumption about the local availability of state generated by previous function invocations. Our applications can scale horizontally in a datacenter by adding a load balancer for the application.

We implement PathExecute based on Nomad [9], a cluster manager and task scheduler that provides us with a common workflow to deploy applications across each of our CloudPath nodes. Nomad interacts with Consul [8], a highly available service registry and monitoring system inside each node to spawn containers and run applications inside of them. Information about running applications required for running Nomad is stored in Consul and other CloudPath modules use the information stored in Consul for deployment, monitoring and routing.

While we anticipate that most CloudPath applications will be written from scratch to take advantage of the unique execution environment afforded by the platform, PathExecute lets application developers leverage existing code and libraries by including them in their deployment package as statically linked binaries. In addition, PathExecute containers can be configured by the CloudPath administrator to include popular binary libraries, such as OpenCV [79].

Cloudpath uses URIs (Uniform Resource Identifier) to identify individual functions. These URIs consist of the application's name concatenated with the suffix `cloudpath.com` followed by the name of the function. Developers determine how URIs are mapped to functions using a deployment descriptor file that should be included in the application package. In addition, developers also specify their preferences for where functions should be deployed in CloudPath hierarchy using the deployment descriptor file. In our current implementation, the standard Java deployment descriptor for web applications (the `web.xml` file) is used to describe how and where the application and its functions should be deployed. Figure 3.4 illustrates how two URIs are mapped to functions and their preferred location to run (edge for /timeZone, and core for /prefs).

3.3.2 PathStore

PathStore provides a hierarchical eventual consistent database that makes it possible for CloudPath functions running in PathExecute containers to remain stateless by automatically replicating application state close to the CloudPath node where the function executes.

PathStore's target environment poses three interesting challenges. First, most nodes can only store a small fraction of the data stored on the wide area cloud nodes that are at the root of the hierarchy. Nevertheless, most reads and writes executed by a CloudPath function should be executed locally; running code close to the edge of the network has little benefit if most data accesses have to go to the cloud. Second, the large number of nodes in the system requires keeping to a minimum the amount of meta-data regarding the current location of data replicas. Third, the geographic distribution of nodes, and the high network latency typical of many paths between nodes requires minimizing coordination and the ability to operate (albeit at diminished capacity) even in case of temporary network or node failure.

To address these challenges, we structured PathStore as a hierarchy of independent object

stores. The database of the PathStore node at the root of the hierarchy is assumed to be persistent, while all other levels act as caches. To simplify the implementation, PathStore requires the data replicated by a node to be a superset of the data replicated by its children. To provide low-latency, all read and write operations are performed against the local database node to which an application server is attached. PathStore supports concurrent object reads and writes on all nodes of the database hierarchy; updates are propagated through the node hierarchy in the background, providing eventual consistency.

Figure 3.5 shows a sample three layer PathStore deployment. PathStore consists of three main components: a *native object store*, the PathStore *server*, and the PathStore *driver*. The native object store provides persistent storage for objects that are temporarily (or permanently in the case of the root) replicated at a node. In our prototype we use Cassandra [85], but the design can be adapted to other storage engines (see 3.3.2). As the figure illustrates, the size of the local Cassandra cluster can differ between nodes. The PathStore server copies data between its local Cassandra instance and the Cassandra instance of its parent node. Finally, the PathStore driver provides an API that third-party applications running inside PathExecute containers can use to query the local PathStore node. Our prototype is based on CQL, Cassandra's SQL dialect, which organizes data into tables, and provides atomic read and write operations at row granularity. CQL lets users read and write table rows using the familiar SQL operation `SELECT`, `INSERT`, `UPDATE`, and `DELETE`; however, CQL operations are limited to a single table – there is no support for joins.

On-Demand Replication

PathStore replicates data at row granularity on demand in response to application queries. Applications issue queries using the PathStore driver which executes them against the local PathStore node; however, before a CQL query is locally performed, PathStore server replicates from the parent node all objects that match the query as determined by the conditions in the *where* clauses of the CQL statement. To prevent a node from fetching data on each query from its parent, the PathStore server keeps a *query cache* consisting of all recently executed CQL queries. Subsequent CQL queries that match an existing entry in the cache are directly executed on the local node. Queries in the query cache are periodically executed in the background by a

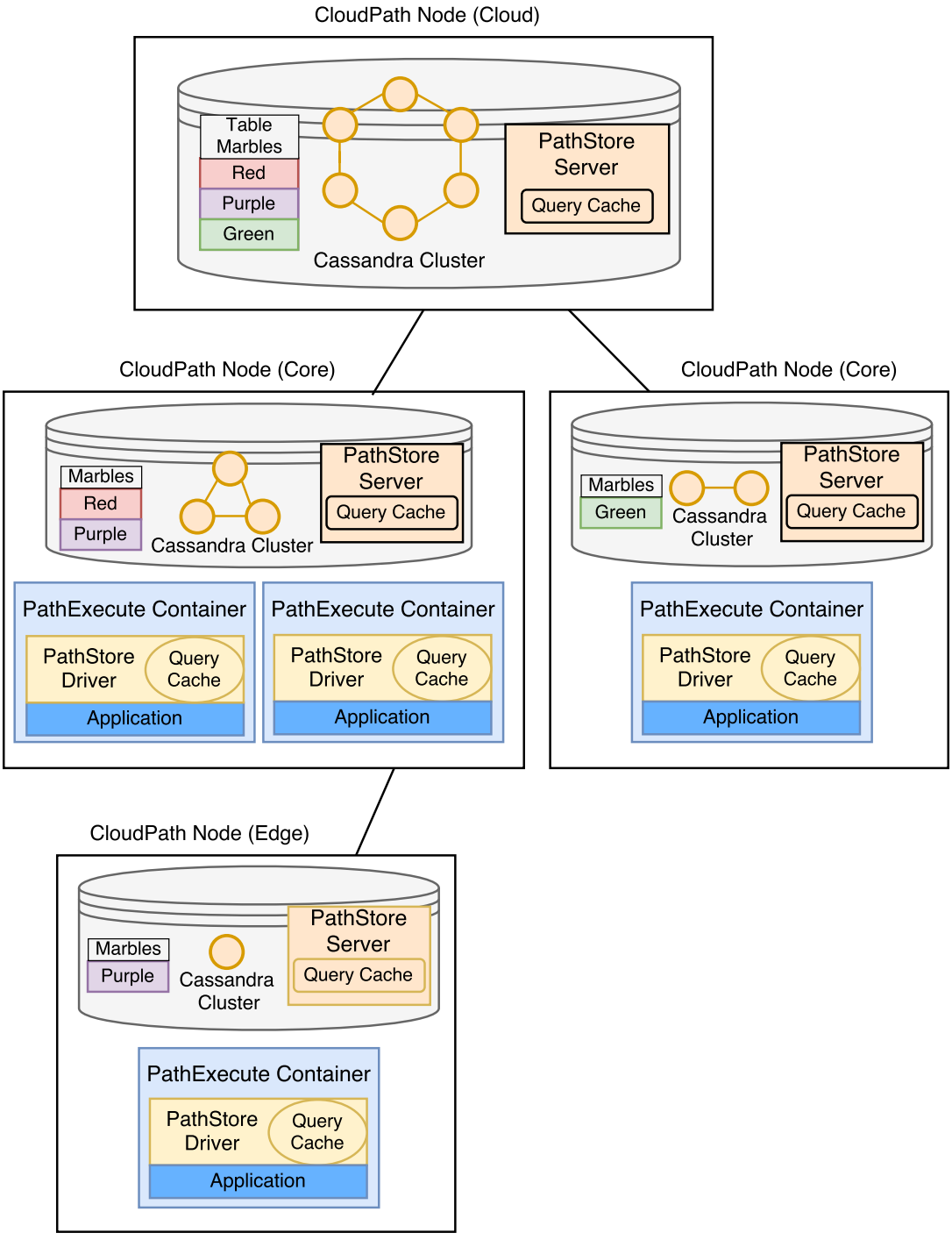


Figure 3.5: PathStore Architecture

pull daemon to synchronize the local node's content with that of its parent (i.e., fetch new and updated records from the parent node). To reduce unnecessary processing, PathStore keeps track of the coverage of cache entries and the pull daemon bypasses queries that are otherwise subsumed by other queries that have a wider scope. For example the query `SELECT * FROM marbles` subsumes the query `SELECT * FROM marbles WHERE color = red`.

Figure 3.5 illustrates this process for a simple table that keeps track of marbles of different colors. In the example, an application running at the edge node issues a query for the purple marble (`SELECT * FROM marbles WHERE color='purple'`). Assuming that this query does not match an existing entry in the edge node's query cache, the query is propagated to the core node's PathStore server, which in turn propagates it to the cloud node's PathStore server. Since the cloud node is the root of the hierarchy, it is assumed to contain all the data and the query does not propagate any further. The core node then executes the query against the Cassandra cluster of its parent node, stores the matching row(s) in its local Cassandra cluster, and stores the query in its query cache. This process is repeated by the PathStore server running on the edge node. Finally, the PathStore driver executes the query against the edge's Cassandra instance. As an optimization, the PathStore driver also keeps a query cache with recently executed queries. Since the driver's cache is guaranteed to be a subset of the server's cache, queries that match the driver's can run directly against the local Cassandra instance bypassing the need to first contact the PathStore server.

The obvious disadvantage of fetching data purely on demand in response to application queries is the significant latency associated with fetching data across multiple levels of the hierarchy. It is easy to imagine alternative approaches that pre-fetch data in anticipation of its use. PathStore could leverage its fine grain knowledge about the data used by applications in the past to predict future usage. For example, it may be possible for PathStore to identify data that is requested for each user served by an application. When a new user connects to a node, PathStore could eagerly fetch the data associated with the new user in anticipation of its use. We leave the exploration of prefetching alternatives for future work.

Update Propagation

PathStore applies all modifications locally, and a *push daemon* periodically propagates local updates to higher levels of the hierarchy. PathStore keeps track of modifications using a write log. In Cassandra, every table has a partition key that determines the host(s) in the Cassandra cluster where a given row will be stored. In addition a Cassandra table can have one of more clustering keys. Rows with the same partition key, but different clustering keys are stored together on the same Cassandra host, in a local order determined by the clustering keys. PathStore implements a write log for each row of a table by adding a *version* column as the last element of the table's clustering key. The *version*, is a UUID timestamp that records the time the row was inserted, and the ID of the PathStore node where the modification was originally recorded. PathStore assumes that nodes are tightly synchronized using some accurate mechanism, such as GPS atomic clocks. As modifications get propagated through the hierarchy (up by the push daemon and down by the pull daemon), PathStore uses the version timestamp to determine order between modifications. In the current prototype the modification with the most recent timestamp wins.

PathStore's write log is not visible to applications, and therefore developers do not have to modify their application queries. Instead, the PathStore Driver automatically collects the multiple versions of a row that match an application's query and returns the most recent data. For example, Figure 3.6 shows a table that keeps track of personalized movie ratings. Columns *user* and *movie* are the original partition and clustering keys, respectively. Column *version* is added by PathStore to implement the write log. The table show that user John initially assigned a rating of 8 to the movie Toy Story, but later updated this rating to 10. Running the query `SELECT * FROM movies WHERE user = 'John'` produces the tuples ['John', 'Toy Story', d33d7fe0-195f-5d569c585662, 10] and ['John','Cars', 825968c0-195d-5d569c585662, 8]; however, the PathStore driver returns only the most recent version of each row and hides any PathStore meta columns, i.e., ['John', 'Toy Story', 10]. Finally, to prevent the log from growing unbounded, PathStore runs a daemon at the root of the hierarchy that periodically trims the log.

user	movie	version	rating
John	Toy Story	d33d7fe0-195f-5d569c585662	10
John	Toy Story	825968c0-195d-5d569c585662	8
John	Cars	7adf7210-1958-59e16851d966	9
Susan	Finding Nemo	6833c850-1958-59e16851d966	8

Figure 3.6: Sample PathStore table.

Data Eviction

Cold query cache entries are deprecated periodically preventing the pull daemon from fetching unnecessary data. Similarly, locally replicated rows that do not match any query in the query cache are periodically deleted. In case of resource contention, our prototype uses a simple LRU policy to free space. Exploring other approaches is the subject of future work.

Local Table

PathStore also provides local tables for temporary storage. Updates to local tables are not propagated to other nodes. In Section 3.4.1 we describe an application that uses local tables to aggregate sensor data at the edge of the network.

Consistency Model

At the individual node level, PathStore preserves the storage semantics of its underlying native object store. Our current prototype, which is based on Cassandra provides local durability, row-level isolation and atomicity, and strong consistency based on Cassandra's quorum mechanism. Across nodes, however, PathStore propagates updates at row granularity following an eventual consistency model. The PathStore driver guarantees that code executing on a specific PathStore node will see monotonically increasing versions of a row (i.e., the driver returns only the most recent version of the row in the write log), and that given enough time without new modifications all replicas of a row on all PathStore nodes will converge to the same most recent value.

Whereas PathStore does not enforce system-wide strong consistency, an application can nevertheless achieve stronger consistency for requests emanating from a subset of the CloudPath hierarchy by instructing the platform to execute its sensitive functions at a common ancestor node. For example, a function running at city-level nodes will provide a consistent view of the

data for all users in any given city, irrespective of the edge node they each use to connect to the network; users in different cities, however, may see inconsistent data while updates propagate through the hierarchy. An application can enforce global consistency by limiting its functions to run at the root of the hierarchy. The stronger consistency, of course, comes at the cost of increased network latency and obviates the benefits of path computing. In the future, we plan to explore other consistency models that will enable applications to control how updates are applied across the storage hierarchy.

Fault Tolerance

PathStore can continue to serve read queries for data that is locally replicated even in the event of network partition; however, queries that are not already in the query cache (of the current node and its reachable ancestors) will fail if an ancestor becomes unreachable. On the other hand, write queries should be able to execute as long as the local Cassandra instance is reachable. A write returns when it is persisted in the local Cassandra instance, and it is guaranteed to remain stored in the local instance until it is propagated to the parent node. A row is marked *dirty* when it is inserted into the local Cassandra instance. PathStore only marks the row as *clear* when the parent acknowledges reception and storage of the write. If there is a failure, PathStore retries propagating the write. If a PathStore node experiences a temporary failure, upon recovery it will retry propagating all writes locally marked as dirty. Data is only permanently lost if a PathStore node experiences a permanent failure before a dirty update is successfully propagated to the parent. We anticipate that permanent PathSore node failure will be a very rare occurrence as PathStore relies on replication in Cassandra to handle individual machine failures.

Other Storage Engines

Whereas the current PathStore implementation leverage Cassandra other similar object stores could be adopted as long as they provide (at the local node level) persistent object storage, row-level isolation, and atomic timestamps.

3.3.3 PathRoute

PathRoute is responsible for routing CloudPath requests to running functions using the URI included in the request. In our current implementation, which uses HTTP to transfer requests to functions, CloudPath applications get an unique sub-domain within the `cloudpath.com` name-space after registration. CloudPath requests should consist of the application sub-domain concatenated with `cloudpath.com` followed by the function name in the web-address such as in: `app_name.cloudpath.com/function_name`. In the clock application example of Figure 3.4, requests to the `getTimeZone()` function should be made to the `clockapp.cloudpath.com/timeZone` web-address.

To divert CloudPath traffic from other traffic flowing in the network, we use a DNS `A record` entry for `cloudpath.com` to map all CloudPath sub-domains to a single IP address. Hence, all CloudPath application requests across the entire network will have the same destination IP address which is the IP address of the PathRoute module on every edge node. The network operator is required to route all packets destined for this IP address to the edge CloudPath node connected to the user. The major benefit here is that by using only one static route on the edge routers, CloudPath traffic can be diverted to the PathRoute module.

We implemented PathRoute using a NGINX [126] proxy. For each new HTTP request received by our NGINX, a look-up is made on a local in-memory state cache using a small script to determine whether a deployment request for that application on the node has been previously made or not. If not, the application identifier is extracted from the request and sent to the PathDeploy module where a decision for application deployment is made. When PathDeploy decides to deploy the application on the node and PathExecute completes the application deployment, the proxy cache is updated. Future requests are then proxied to the PathExecute container running the application (the function preferences should also match the node's location).

If PathDeploy decides against deploying the application on the node, subsequent requests will be proxied to the next CloudPath node in the hierarchy (we assume each PathRoute proxy has the address of the PathRoute module of its parent node). In CloudPath, requests can only move upwards towards the root cloud node.

Handover

When the user moves between edge nodes (e.g., handover), the IP address of the source client and the destination which is the PathRoute module on the edge node node is still valid, but the traffic will flow through a different set of routers and will therefore lead to the execution of the function at a different CloudPath node. In case of a hard handover the existing TCP connections would be terminated, and the client is forced to reconnect and restart their request. When a soft hand over happens, the connection is restarted at the edge CloudPath node. Because of the short-lived nature of the requests, restarting the connection would not lead to significant overhead.

3.3.4 PathDeploy

PathDeploy is responsible for initiating the process of deploying an application and its functions on a node. Application deployment decisions are triggered by PathRoute requests. The decision on whether to deploy the application on a particular node depends on higher level system policies, user preferences and the resource status on that node. One policy that we include in our prototype is that functions specified by the user to run on a certain level of the CloudPath hierarchy can also run on any higher level and all functions by default run on the cloud node. If the application is to be deployed on the node, PathDeploy retrieves the application code from PathStore, and sends the code to PathExecute. In parallel, a request is made to PathStore module of the node to create the application data tables based on the schema.

Our PathDeploy prototype is a Java HTTP server. When new requests for an application and function deployment arrive at PathDeploy, it retrieves the application and function information, including user preferences from PathStore to decide if the application and function should be running on the node or not. If the decision to deploy an application is positive, PathDeploy deploys the application locally by fetching the application code from PathStore and passing it to PathExecute along with the application meta-data. In parallel, it fetches the application database schema from PathStore and creates the application data tables on the node.

At present, when we deploy an application on a node, all its functions are deployed at once,

but requests are only forwarded to a subset of functions that should be deployed on that node. A fine grain deployment scheme will be implemented in future versions of CloudPath.

3.3.5 PathMonitor

PathMonitor is designed to provide insight into the lifecycle of a deployed application as well as the status of the CloudPath nodes themselves. It consists of both a back-end module that collects data from the various modules and third party applications of CloudPath, as well as a front-end web application to present the collected data.

PathMonitor pulls statistics such as CPU and memory metrics for containers and hosts from the PathExecute module. This data is pre-processed, aggregated, and stored on the PathStore module in the node. In addition, the various CloudPath modules and third party applications create logs depending on the application function; such as access, status, and error logs. PathMonitor acts as a central point for collecting and storing logs that are created by these other modules.

The front-end web interface lets us visualize the current and past state of the CloudPath system through various graphs and infographics, such as; the topology of the system, including the hierarchy of nodes; CPU and memory metrics for application containers; and the same metrics for the hosts and nodes themselves.

3.3.6 PathInit

In the root cloud node, the PathInit module is responsible for receiving the applications from the developers through a web interface, extracting application and function properties from the `web.xml` file, and saving them along with the application's code and database schema file on PathStore.

PathInit also creates augmented tables from the submitted application database schema file, and deploys the application on the root node. The augmented schema files and the information about the application are used by PathDeploy to deploy the application on other nodes.

Deployment Location	PathDeploy Processing	Nomad Processing	Container Spawning	Application Initialization	Consul Update	Database Initialization	Total Time
Cloud	463.9 (118.1)	131.6 (29.7)	1754.9 (28.3)	841.0 (22.8)	33.1 (13.6)	234.3 (22.1)	3531.2 (133.8)
Core	823.0 (140.1)	129.7 (32.8)	1772.1 (53.6)	887.0 (27.6)	36.6 (19.5)	622.5 (35.2)	3849.8 (154.5)
Edge	1005.2 (155.5)	133.6 (45.3)	1709.3 (45.4)	866.4 (18.9)	39.4 (20.2)	974.3 (55.4)	4084.3 (181.3)

Table 3.1: Breakdown of average time required to deploy an application in milliseconds on different locations. Standard deviation in parenthesis.

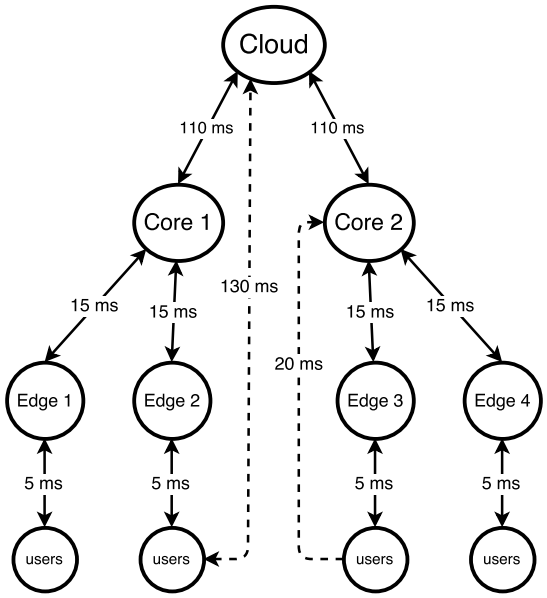


Figure 3.7: Topology of our experimental setup network with average round trip times of the links

3.4 Experimental Setup

Our experiments emulate a CloudPath deployment consisting of a cloud node, and two mobile networks each with one core node and two edge nodes. Figure 3.7 depicts this topology. Each CloudPath node is implemented in a separate computer. The only part of the experiment that is emulated is the network between the clusters which is done by using Linux's Traffic Control, that enabled us to configure the Linux kernel packet scheduler. Network latencies are chosen based on results from the paper by Hu et al. [75], and we use a normal distribution to describe the variation in delay. The average round trip times of the links is included in Figure 3.7.

3.4.1 Test-Cases

We created a series of microbenchmarks to measure deployment time, routing overhead, and the latency and throughput of PathStore. We have also implemented a series of sample user applications to show how our platform supports different categories of applications that can benefit from the architecture:

Face detection: Computational resources on edge nodes can be suitable for offloading resource-intense functions from the mobile end-user device. When offloaded to the edge, applications can benefit from an increase in execution speed and battery lifetime [23]. Our face detection application is deployed as a Servlet and uses the image processing library OpenCV [79] through its Java interface JavaCV to detect faces in an image. The input to this application is an image sent in an HTTP request. The application finds faces in the image and saves them in PathStore.

Localized face recognizer: Using PathStore, applications can push localized content to edge nodes based on the geographic location of end-users. One example is face recognition classifiers which have been trained on a specific dataset, relevant to a given geographical location. We use the AT&T face dataset [3] consisting of a total of 400 face images, of 40 people (10 samples per person) and divide it into 4 separate smaller datasets. We then train 5 different classifiers using the FisherFaces algorithm in OpenCV on these smaller datasets. We store the classifiers in PathStore and the face recognizer application running on each edge node retrieves them.

IoTStat aggregator: Another important benefit of having multiple processing edge nodes close to the user is their ability to filter and aggregate streams of data. As the number of IoT devices using the Internet is likely to raise significantly in the future, processing and filtering data on the edge will decrease the amount of traffic from these devices that need to go through the Internet. We implemented a sample application that performs aggregation functions (average, min, max) on data received from sensors on edge nodes. The first handler of this application receives and parses HTTP/JSON requests containing the sensor data, and stores the extracted information onto a local PathStore table. A second handler, that can be called periodically using HTTP requests, then performs MIN, MAX and AVERAGE queries on the data stored

Execution Location	Direct Connection	Direct Connection (no latency)	L7 Routing	L7 Routing (no latency)
Edge	5.38 (0.38)	0.413 (0.2)	5.755 (0.411)	0.89 (0.22)
Core	20.33 (1.37)	0.465(0.32)	20.96 (1.64)	1.23 (0.77)
Cloud	130.46 (6.65)	0.443(0.38)	132.20 (7.45)	1.45(0.93)

Table 3.2: Average RTT for HTTP requests in milliseconds. Standard deviation in parenthesis

by the first function within specific time frames. This processed data is then saved in another regular PathStore table, which is pushed to the core and cloud.

3.5 Results

We evaluate CloudPath and its applications from different aspects:

- The deployment time of applications on a specific node
- The minimum routing time for applications
- The performance of PathStore and its overhead
- Connection handover between edges
- Benefits for applications

3.5.1 Deployment Latency

We measure the performance of our system in terms of average deployment time of a sample Hello World application with one table. The process is initialized by an HTTP request received by PathRoute, which triggers a container deployment request in PathDeploy as the application is not already deployed on that node. Table 3.1 shows the amount of time required by PathDeploy and PathExecute to retrieve and deploy an application on a particular node. Each experiment was repeated 15 times. The initial time to retrieve application and function information from the database and make a deployment decision is shown in the first column (PathDeploy Processing). Then the next steps (Nomad Processing, Container Spawning and Application Initialization) are done in PathExecute while the application database initialization

from the stored schema file is done in parallel. As shown in this table, as we move from the cloud towards the edge, the average database initialization time and the PathDeploy processing time increases. We assumed the worst case scenario where the application data has to be fetched all the way from the cloud. In practice, an edge deployment will likely get a hit on the core tier. However the overall processing time is still between 3.5 seconds in the cloud to 4.08 seconds in the edge. If developers have larger applications with more complex databases, this time is likely to increase because more data should be retrieved from PathStore.

A non-FaaS approach requires the full VM or container to be downloaded on the edge node each time it is required. To compare that approach with ours, we measured the overhead time required to download a minimal container with only Java installed from a cloud repository, which on average was 13.2 seconds. In CloudPath, this time is saved during each deployment because all clusters are pre-loaded with the executing container.

Routing Overhead

To calculate the overhead that our systems adds to each packet, we measure the average response time of requests using an HTTP benchmarking tool called wrk [58]. We compare the latency of a baseline approach where no proxies exist between the user and the container (direct connection) to our method, where we use layer 7 routing using PathRoute. In our experiment, after creating a single TCP connection with the proxy, a new request is sent when an acknowledgment for the previous request is received. This is repeated for 10 seconds for 16 concurrent connections. The average round trip times (RTT) and standard deviation is presented in Table 3.2. Furthermore, we compare our L7 routing with the baseline approach, when there isn't any emulated latency in the network. As shown in this Table, our routing method only introduces a slight increase in latency compared to the baseline approach where packets are routed on layer 3. This is specially evident when no emulated latency exists in the network and shows the overhead introduced by our PathRoute proxies is about 0.9ms for the first layer and about 0.3ms for each additional layer.

Deployment Location	100 Entries			1000 Entries			10000 Entries		
	PathStore First query	PathStore Consequent queries	Native Cassandra driver	PathStore First query	PathStore Consequent queries	Native Cassandra driver	PathStore First query	PathStore Consequent queries	Native Cassandra driver
Edge	529.1 (9.7)	4.7 (0.4)	506.4 (7.4)	1664.2 (177.1)	29.5 (9.3)	1035.5 (181.9)	8857.5 (123.9)	113.0 (8.4)	2640 (195.0)
Core	510.4 (13.8)	4.4 (0.1)	398.9 (15.9)	1121.8 (47.3)	24.8 (0.7)	869.0 (34.3)	5244.8 (282.1)	111.1 (2.1)	2268.7 (147.4)
Cloud	6.3 (0.6)	4.9 (0.2)	6.1 (0.1)	29.7 (0.9)	26.9 (1.6)	27.1 (1.78)	137.5 (6.7)	112.8 (6.4)	124.9 (3.1)

Table 3.3: Time required for querying full tables in milliseconds (standard deviation in parenthesis).

PathStore Performance

We next measure the performance of PathStore using micro benchmarks. We present the time required to execute different types `SELECT` and `INSERT` queries both when data is located locally or is available on a parent node.

Local Read Latency

We compare the time to execute `SELECT` queries on a local node using PathStore versus using the native Cassandra driver. In Figure 3.8, we depict the Cumulative Distribution Function (CDF) of the time it takes to execute these queries. The blue line shows the performance of the native Cassandra driver while the green line shows the performance of the PathStore driver when we execute 1000 `SELECT` queries with `WHERE` clauses that match individual rows and result in a miss in the local client query cache. Each row of our table contains $1KB$ of data. The red line shows the performance of the PathStore driver with the difference that before 1000 individual `SELECT` queries, a single `SELECT` query without any clauses on the same table is made so that data would be cached locally. This results in hits on the client query cache. As shown in Figure 3.8, the PathStore driver is on average 1.6ms slower than the native Cassandra driver when it misses the client query cache. However the performance of our driver is close to the baseline if we have hits in the query cache.

Local Write Latency

We also measured the time required for executing local `INSERT` queries and there were no noticeable difference in terms of performance between the PathStore driver and the Cassandra

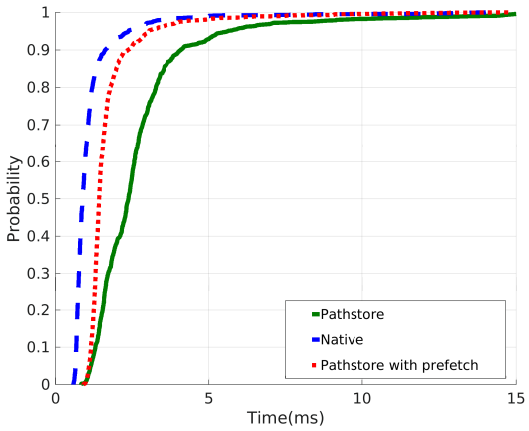


Figure 3.8: CDF chart for 1000 select queries on a local node.

driver. For `INSERT` queries, our PathStore driver does not add extra overhead to the native Cassandra driver.

Remote Read Latency

We next analyze the time required to retrieve data to the edge and core from the cloud node using `SELECT` queries that match individual rows. These queries are executed from the core and edge nodes where there is a miss in the client and server query cache of the PathStore unit . We measure the retrieval time for 1000 queries and present the CDF in Figure 3.9. The green and red figure show PathStore's execution time from the core and the edge. The blue and orange lines show the execution time for the native Cassandra driver from the core and the edge. We can see that in Figure 3.9, PathStore (red and green lines) is nearly twice as slow as the Cassandra driver (blue and orange lines) in retrieving the data. This is because PathStore first checks to see whether data is present on the parent node or not. Then when it gets the response back, it fetches the data from the remote node. This adds one RTT time to each query.

We next measure the time required to fetch different number of entries (a whole table) from edge, core and cloud nodes when the data is only initially located on the cloud node. Again each row or entry is 1KB. We do a `SELECT` query with no clauses to fetch the whole table at different nodes with PathStore and the native Cassandra driver. The experiment is repeated 20 times and the results are presented in Table 3.3. As shown in the table, PathStore's first query takes

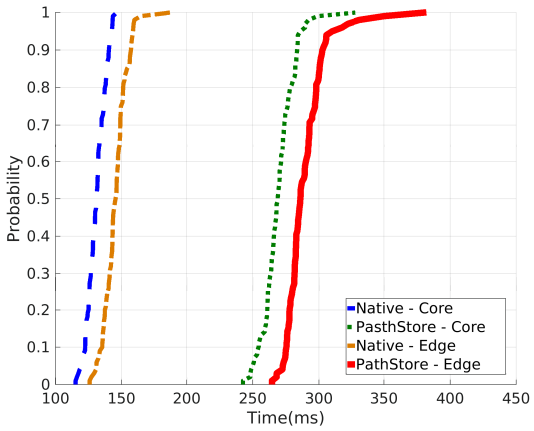


Figure 3.9: CDF chart for individual select queries on a remote node.

more time than the native Cassandra driver to retrieve the data, but for consequent queries, as the data is already fetched, we will have a hit on the local client cache and the queries would take the same time as a local `SELECT`. Furthermore consequent queries that are a subset of the first query will also be fetched locally.

Update Propagation Latency

We also measure the propagation latency of a single `INSERT` query. In this scenario we execute a single `INSERT` query at different nodes of the hierarchy and measure the time that the single row update takes to propagate to all other nodes that have previously issued a `SELECT` query. Figure 3.10 illustrates the results of this experiment . The links shown in this Figure are logical links between nodes from Figure 3.7. Meaning a query travelling between $Edge1$ and $Core2$ has to traverse nodes $Core1, Cloud1$. As shown in this Figure, moving down on the tree takes more time than moving up. This is because pull operations require 2 RTT's while push operations only need half an RTT. `INSERT` and `UPDATE` operations on a node get pushed all the way to the cloud, while nodes can express their interest in a certain query with a `SELECT` query. This results in periodic data pulls from parent nodes when updated information on the parent about the query exists.

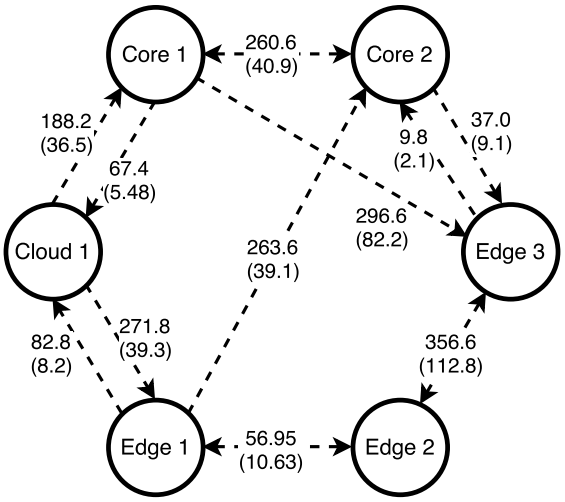


Figure 3.10: Propagation time (in milliseconds) of an `INSERT` query between nodes (standard deviation in parenthesis). Links are logical links between nodes from Figure 3.7.

Data Overhead

Finally we measure the total overhead of each table in PathStore. We add 5 columns to each of our PathStore tables and in total, 38 bytes is added to each entry.

Handover Latency

We examined the effects of a soft handover in case of mobile movement between two edge nodes A to B. When a soft handover happens, then TCP packets will continue to be sent from the user device (as explained before, all PathRoute modules on every edge, have the same IP address) however the data arriving at the PathRoute module of edge will not accept such packets because no such TCP connection exists, so it sends a TCP packet with the RST flag set and the connection will be re-initiated by the user device. We emulate a soft handover in our environment and measure the time required to re-initiate the connection. On average it takes 16.56 milliseconds (4.03 standard deviation for 100 experiments) for the device to start re-initiating the connection, and another 15.2 milliseconds to establish a new TCP connection.

3.5.2 Application Performance

In this section, we will show the benefits of running applications on CloudPath. These benefits come in the form of greatly reduced response time or reductions in data and network traffic.

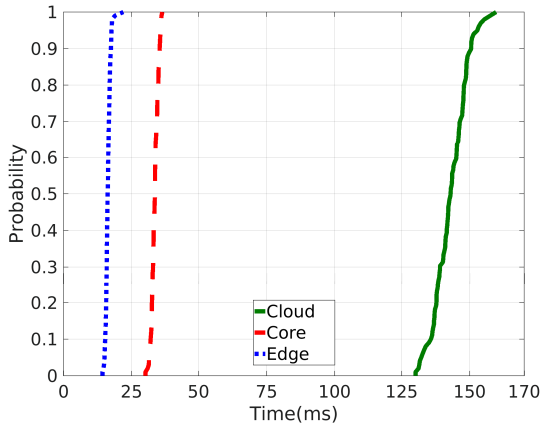


Figure 3.11: CDF for response time of the Face Recognition application.

Face Detection

We measure the average response time of the face detection and face recognition programs when they are deployed at different locations. For the face detection program, the average response time for 100 different requests when the program is running on the edge, core and cloud is: 13.6(2.1), 32.6(1.8), 141.9(6.4) milliseconds (standard deviation in parentheses). The same image was used and the file size was $2KB$. There is a substantial reduction in response time when running on the edge (closer to the client) compared to running on the core and cloud.

Face Recognition

The Face Recognizer program labels an input image (received through HTTP requests with a file size of 11KB) based on a trained model. The results for processing 100 requests are illustrated in Figure 3.11. Similar to the Face detection program, running on the edge lowers the latency by 88 percent.

IoTStat Aggregator

For the IOTStat aggregator application, the average processing time of each query received at the edge node is 1.6 ms for 3 different sensor values in the same request. This means that an application running on the container (1 core) can handle up to 900 queries per second. The processing time required for aggregating 1000 queries, is about 10ms. If there were no

aggregation, then n sensors sending k requests per second will send $n \times k$ messages to the cloud for processing. However if we assume a single layer of aggregation (p edge nodes), assuming that aggregation results are required every second, then the total number of messages sent to the cloud would only be: p. In our example we insert a row containing the aggregation results of each edge node on to PathStore which would push this data to the cloud.

3.6 Chapter Summary

In this chapter, we presented *path computing*, a new paradigm that enables processing and storage on a progression of datacenters interposed along the geographical span of the network. Path computing gives applications developers the flexibility to place their serve functionality at the locale that best meets their requirements in terms of cost, latency, resource availability and geographic coverage.

We also described CloudPath, a new platform that implements the path computing paradigm. CloudPath minimizes the complexity of developing path computing applications by preserving the familiar RESTful development model. CloudPath applications consist of a collection of short-lived and stateless functions that can be rapidly instantiated on-demand on any datacenter that runs the CloudPath framework. CloudPath makes this functional decomposition possible by providing an eventual consistent storage service that automatically replicates application state on-demand across the multiple datacenter tiers to optimize access latency and reduce bandwidth consumption.

Our experimental evaluation showed that CloudPath can deploy applications in less than 4.1 seconds and has negligible read and write overhead for locally replicated data. Moreover, our test applications achieve up to 10X reductions in response time when running on CloudPath compared to an alternative implementation running on a wide-area cloud datacenter.

Chapter 4

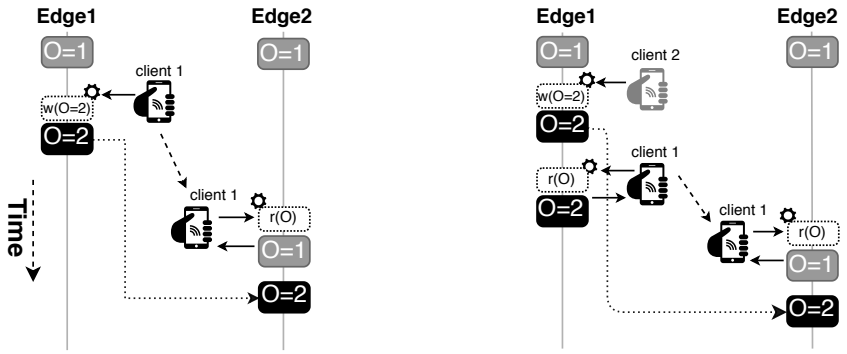
SessionStore: A Session-Aware Datastore for the Edge

Edge computing expands the traditional cloud architecture with additional datacenter layers that provide computation and storage closer to the end user or device. For example, a wide-area cloud datacenter which serves a large country can be augmented by a hierarchy of datacenters that provide coverage at the city, neighborhood, and building levels.

Recent storage systems for the edge [116, 42, 124, 64] generally rely on eventually consistent models [152, 11] to replicate data. These systems propagate updates in the background and guarantee that if no new updates are made to an object, eventually all replicas will converge to the same value. Eventual consistency works well for many applications where clients interact with the same replica for the duration of their sessions. The reason is that as long as the client interacts with the same replica, the storage system in effect provides *session consistency* [152], a stronger consistency model that has additional important properties: *read-your-writes*, where subsequent reads by a client that has updated an object will return the updated value or a newer one; and, *monotonic reads*, where if a client has seen a particular value for an object, subsequent reads will return the same value or a newer one. While session consistency does not guarantee that different clients will perceive updates in the same order, it nevertheless presents each individual client with an intuitive view of the world that is consistent with the client's own actions. Examples of applications that can benefit from session consistency on the edge include authentication services, file storage applications and messaging applications. We describe more usage scenarios for session consistency on the edge in Section 4.1.

Session consistency however, may not be guaranteed when consecutive client requests are sent to different replicas. This may occur in edge applications when: (i) a mobile client switches between edges [27, 120]; (ii) functionality is dynamically reallocated between edges [141]; or (iii) an application's functionality has been partitioned between different datacenters [26, 146, 167] (e.g., running some functions on the edge and others on the cloud). If consecutive client requests are sent to different replicas before data needed by the client request is replicated, the application may not be able to read its own writes or have monotonic reads.

Figure 4.1 illustrates two such scenarios. In Figure 4.1a, *client 1* writes object O on Edge1. As a result of mobility, *client 1* switches its association to a different Edge2 and observes the old value of O on its subsequent read. In the second scenario illustrated in Figure 4.1b, *client 2* issues a command that results in object O being overwritten on Edge1. *Client 1* then reads



(a) Read your writes failure

(b) Monotonic reads failure

Figure 4.1: In (a) the client writes and Edge1 but its consequent read on Edge2 return an old value. In (2) client 1 reads a value but when it makes the read on Edge2, an old value is returned

this value and moves to Edge2. If *client 1* issues another request that reads object O on Edge2, an old value will be returned. While in the previous examples, clients read and write directly to the replicas of the storage system, this is done purely for ease of explanation. In practice, clients instead communicate with a replica of a service (e.g., an HTTP server) deployed on each edge datacenter that runs application code that access the replicated datastore.

We present SessionStore, a distributed datastore tailored for fog/edge computing that ensures *session consistency* between a hierarchy of otherwise eventually consistent replicas. Whereas previous approaches [24, 20] that provide session consistency on top of eventual consistent storage systems target applications running on a relatively small number of cloud datacenters, SessionStore is designed for applications running on a large and variable number of edge/fog datacenters. SessionStore supports resource-limited datacentres by leveraging partial replication and only replicating data on demand. SessionStore supports session consistency using a *session-aware* reconciliation algorithm that only reconciles keys that a client either reads or writes at the source replica. SessionStore further minimizes the data transfer by not transferring up-to-date data already existing on the destination. In our example application use case, this saving is as much as 95% in terms of data transfer.

The main contributions of SessionStore are: (i) a session-aware reconciliation algorithm that enforces session consistency by only transferring relevant client data; (ii) a prototype that implements our algorithms; (iii) an experimental evaluation based on micro-benchmarks and

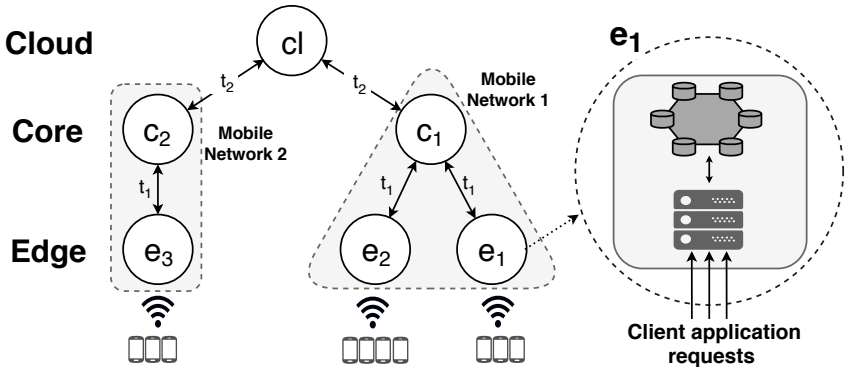


Figure 4.2: Hierarchical datacenter topology

the RUBBoS benchmark that shows SessionStore is able to guarantee session consistency at a fraction of the latency and bandwidth costs of a strongly consistent implementation.

The rest of this chapter is structured as follows. Section 4.1 discusses scenarios of edge applications that require session consistency. Sections 4.2 and 4.3 discuss design considerations and our implementation. Section 4.4 presents the results of our experimental evaluation. Section 4.5 summarizes the chapter and discusses avenues for future work.

4.1 Use Cases

In this section, we give examples of scenarios that require session consistency to guarantee that requests emanating from the same client experience a view of the underlying data that is consistent with the client's own actions. We consider applications deployed on an edge network with two or more levels. For example, Figure 4.2 shows a sample edge network consisting of a cloud datacenter, and two mobile networks each with a datacenters at its core, and one or two additional datacenters at edge location such as base stations. Each datacenter has a replica of the datastore, as well as additional servers to run application code. We assume that the cloud datacenter stores a persistent full replica of the datastore. Each of the other datacenters hosts a partial replica and data gets replicated on-demand. On a read access, if the data is not already available on the local replica, it is fetched recursively from its parent. Similarly, updates are applied to the local replica and get propagated through the replica hierarchy in the background.

We enforce session consistency by grouping related datastore accesses into a *session* based on

application-specific considerations. In the examples below, we uses a session to group together data accesses executed on behalf of the same user; however, it is possible to think of other applications where a session could be used to group together accesses executed on behalf of a device or a specific application module or function. We consider the case where requests that belong to the same session execute against different replicas due to: (i) user mobility; (ii) different parts of an application being deployed on different datacenters; or (iii) code mobility. The use cases below follow a stateless server design pattern where all application state is kept in the datastore, and applications are implemented as a collection of independent stateless functions.

Scenario 1: Mobile Client In this scenario, as a user moves around, his/her requests get routed to the closest edge datacenter. Session consistency is needed when the state that is read or written when connected to one edge datacenter is later accessed again after the user switches to a different edge. Consider the case of a user that leverages edge computing to edit a video. After recording a video on their phone, the user uploads it to an edge video-editing service which stores it in the datastore. The user then boards a bus, and proceeds to edit the video by applying a sequence of filters (e.g., image stabilization, cropping). By grouping the operations performed on behalf of the user into a single session, a datastore that provides session consistency guarantees that the effect of each of the filtering operations is preserved even as subsequent operation may run on different edges along the route as the bus travels.

Scenario 2: Functional Partitioning. In this scenario, an application's functionality is partitioned and deployed on different datacenters. Session consistency is needed when the results of executing one function on one datacenter should be made visible to another function running on a different datacenter. As an example, consider the case of a simple access control service that consists of three functions: *login*, *logout*, and *authorize*. A client logs into the system by providing a password to validate against a hash stored in the datastore. The *login* function is deployed on the *cloud* datacenter to ensure that sensitive password information is not replicated anywhere else. After successful validation, *login* adds a certificate with the user's permissions to the datastore. Similarly, to log a user out, *logout* modifies the certificate to indicate that it is no longer valid. Subsequent client requests (e.g., read an email, send a message) execute on one of the *edge* datacenters after first running *authorize*, which involves

reading the user's certificate from the datastore to verify its validity. By grouping the operations performed on behalf of a client into a single session, a datastore that provides session consistency would guarantee that the version of the certificate created by the most recent invocation to *login* or *logout* is the one that is read by *authorize*.

Scenario 3: Function Mobility. In this scenario an application (or an application component) is reallocated between datacenters. Migration may be done for load balancing purposes, when the demands of a task surpass the locally available resources on the current execution location, or to improve quality of experience. Session consistency is needed when after migration an application reads state from the datastore in the new datacenter that was either read or written in the old datacenter. As an example, consider the case of an interactive web-hosted game that stores the state of the game in the datastore. When the network is experiencing low queuing delay, the application runs on the cloud, but migrates to a datacenter on the edge when an increase in wide-area traffic degrades the user's experience. By grouping the operations performed on behalf of each user into their own session, a datastore that provides session consistency guarantees that after migration the state of the game presented to the user corresponds to the user's last move.

The previous use cases can also run correctly on top of a datastore that provides stronger consistency guarantees, such as sequential consistency or casual consistency; however, the stronger properties come at a large cost in terms of bandwidth and latency as we show experimentally in Section 4.4. The above scenarios do not require a globally consistent view of the world; instead, they only require a view of the world that reflects the actions of operations that belong to the same session. The rest of this chapter shows how session consistency can provide this guarantee with low overhead in terms of data transfer and replica switching cost.

4.2 Design Considerations

In this section, we elaborate on our design choices for adding support for session consistency to a replicated datastore that runs on a hierarchy of datacenters that facilitate edge computing. We consider three dimensions: when to synchronize state, what state to synchronize, and how to keep track or identify the state that needs to be synchronized.

Session consistency can be enforced either *proactively* or *re-actively*. In a proactive implementation, data is continuously sent to other replicas eagerly. This approach supports fast switching between replicas; however, it results in high bandwidth consumption. On the other hand, a reactive implementation ensures session consistency only after a client switches to a new replica. Before running code on behalf of a client on a new replica, all relevant state has to be synchronized, which may incur delay.

We argue that the reactive approach is more appropriate for edge computing because the latency and resiliency demands of edge computing may dictate that mobile clients must often be served by their closest replica – thereby the proactive approach necessities a large number of service replicas and hence a very high replication factor that results in more resources needed on the edge. Our experiments confirm that a proactive implementation incurs large update latencies and high data volumes even for a modest replication factor. Conversely, the latency to switch between replicas that are kept in sync using reactive replication is relatively small (see Section 4.4.3).

This choice, however, does not preclude eager replication to a small number of replicas that the system determines have a high likelihood of executing queries on behalf of the session. We leave the study of pre-fetching for future work.

State between replicas can be synchronized using either *full replica reconciliation* or *session-aware data reconciliation*. In the former, the destination replica will have the latest/synchronized union of all records available at both replicas before the switch occurs. The advantage of this method is that it is conceptually simple, however it may result in high switching time and high bandwidth consumption for the transfer. In the latter, only data relevant to the session, including any records that were read or written, are synchronized. This approach is efficient in terms of data transfer and switching time; however, it is more complex and requires application support to identify relevant data accessed by the session. We argue that for multi-user services where the same replica handles requests from multiple clients, the second option where only the session's data is synchronized is more beneficial. Our experiments show that this approach reduces bandwidth requirements and latency. Moreover, in our experience the effort to label queries is modest.

To keep track or identify the state that needs to be synchronized, we can either tag individual

records with read and write information, or use a higher level abstraction, such as user queries to capture access patterns. The benefit of tagging individual records is its simplicity, which comes at the expense of potential significant additional storage overhead for data objects. Instead, we opted to track data accesses by recording SQL-like queries executed against the replica. While this approach is more complex to implement, it has lower storage requirements as simple queries can identify many data objects.

4.3 SessionStore

In this section we describe SessionStore, our distributed datastore for edge computing which guarantees *session consistency* on top of otherwise eventual consistent replicas. The basic idea behind our approach to ensuring *session consistency* is simple, yet effective: we group related datastore operations into sessions, and we track all the rows either read or written to by a session through tracking the queries it executes. When a client switches from a source to a destination replica, we ensure that the same (or newer) versions of the rows associated with their session are present on the destination replica before executing new queries.

In the rest of this section, we first describe a distributed datastore that provides eventual consistency across a hierarchy of replicas that extend from the cloud to the edge of the network. We next describe how we add support for session consistency on top of this otherwise eventual-consistent datastore.

4.3.1 Eventual-Consistent Operation

Our session consistent datastore is based on PathStore, an eventual-consistent object store introduced in the previous chapter. PathStore is structured as a hierarchy of replicas configured as a tree with a persistent replica at the root, and an unlimited number of layers of partial replicas below it. Our implementation uses Cassandra [85] which can run on typical laptops [4] or even Raspberry Pi's [6], making it a feasible choice for edge deployments. Each replica runs a separate independent Cassandra ring, and our code is in charge of replicating data between otherwise independent rings. We replicate data at row granularity on demand in response to application queries. Each of the independent Cassandra rings may in turn consist of multiple

servers and data may be internally replicated by Cassandra for fault tolerance or performance. In the rest of this chapter, the term *replica* refers to a (potentially multi-server) Cassandra deployment on a datacenter in the PathStore hierarchy.

The datastore provides an API based on CQL, Cassandra's SQL dialect, which organizes data into tables, and provides atomic read and write operations at row granularity. CQL lets users read and write table rows using the familiar SQL operation `SELECT`, `INSERT`, `UPDATE`, and `DELETE`; however, CQL operations are limited to a single table – there is no support for joins.

Figure 4.3 illustrates a simple 3-level deployment of the datastore (a cloud replica, and two mobile networks each with a replica at its core, and one or two additional replicas at edge location such as base stations). To provide low-latency, all read and write operations are performed against the local replica. During a read query on a local replica, if the query has not been previously executed on the replica, we fetch it recursively from its parent. The query is then added in a *Query Cache* that keeps track of recently executed CQL queries. Subsequent CQL queries that match an existing entry in the cache are directly executed on the local node. Queries in the query cache are periodically executed in the background by a *pull daemon* to synchronize the local node's content with that of its parent (i.e., fetch new and updated records from the parent node). To reduce unnecessary processing, we keep track of the coverage of cache entries and the pull daemon bypasses queries that are otherwise subsumed by other queries that have a wider scope. For example the query `SELECT(*) FROM balloons` subsumes the query `SELECT (*) FROM balloons WHERE color=red`.

The datastore supports concurrent object reads and writes on all replicas of the hierarchy; updates are propagated up toward the root of the replica hierarchy in the background by a *push daemon*. Modifications are tagged with a version timestamp that records the time the row was inserted, and the ID of the replica where the modification was originally recorded. We assume that replicas are tightly synchronized using some accurate mechanism, such as GPS clocks. As modifications are propagated through the hierarchy (up by the *push daemon* and down by the *pull daemon*), we use the version timestamp to determine ordering – most recent timestamp wins.

Figure 4.3 illustrates the operation of PathStore for a simple table that keeps track of balloons of different colors and sizes. Initially (Figure 4.3a), the cloud replica stores 2 balloons,

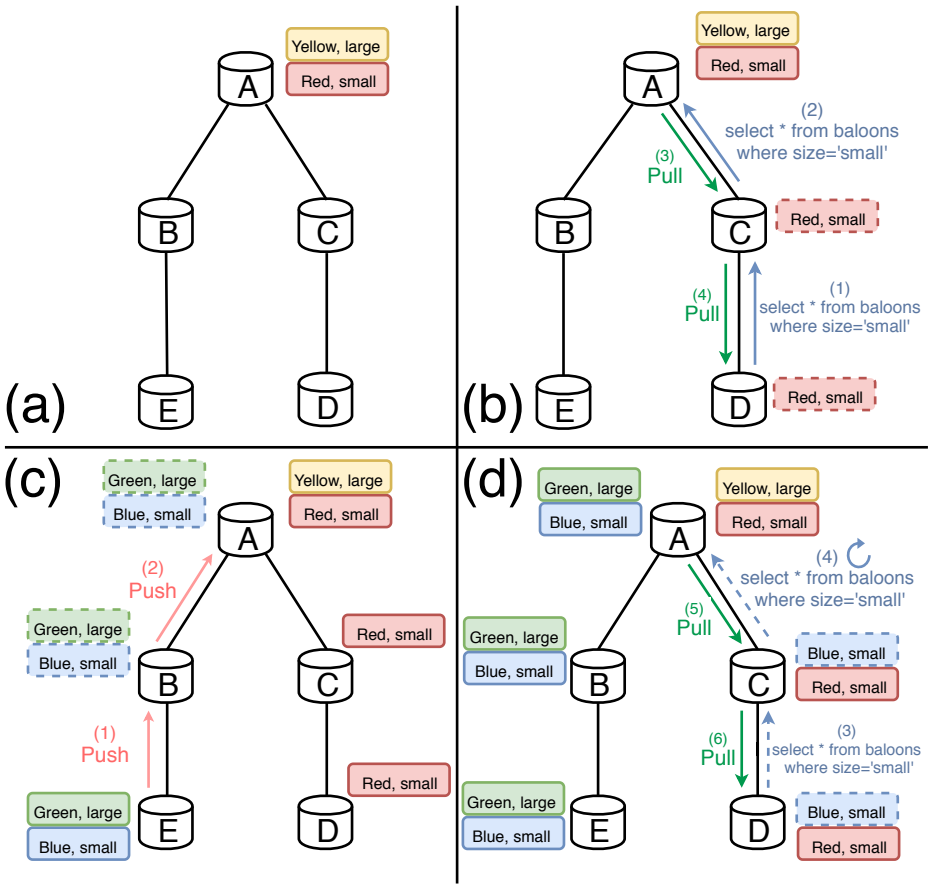


Figure 4.3: PathStore operation.

and all other replicas are empty. Figure 4.3b shows the result of running a query for small balloons (`SELECT(*) FROM balloons WHERE size=small`) on edge D: the small red balloon is first copied to the replica C and the query is added to node C's query cache. From there, it is then pulled on to edge D. Figure 4.3c shows how the state changes after an application running on edge E adds two new balloons, one large green and one small blue. The push daemon of edge E propagates these two new balloons onto B. From there, the push daemon of B replicates the balloons onto the cloud replica. Figure 4.3d shows how the pull daemon on node C identifies that there is a new balloon on the cloud replica that matches the query in its query cache, and pulls the small blue balloon to node C's replica. Similarly, the pull daemon on edge D also detects a new balloon on node C that matches the query in its query cache and automatically pulls the small blue balloon onto node D.

4.3.2 Session-Consistent Operation

We next describe how we expand the above eventual-consistent implementation with support for session consistency across replicas. We fist describe how users can group database accesses into sessions. We then describe how we track data related to a session. Finally, we discuss how we perform session-aware replica reconciliation.

Sessions

We enforce session consistency by grouping related CQL requests into a *session*. What constitutes a session, however, is left to the application developer to determine. For example, the developer can decide to make a session representing a user, a device belonging to a user, a set of commands executed by a function, or a subset of the requests issued by a device. Our system simply enforces session consistency semantics among those queries that are identified as belonging to the same session.

We identify each session using a custom token called the Session Token, or *stoken*. The *stoken* consists of a four fields: A unique session id (*SID*), *timestamp*, *current replica*, and *status*. The *stoken* is included in all messages sent by the devices, and can be encrypted and signed to prevent forging and misrepresentation by a centralized authentication system. Developers chose between eventual and session consistency by including (or not) the *stoken*

together with their queries. In our experiments, we use Java Servlets to run our server-side code and pass the *stoken* using an HTTP cookie.

State Tracking

To keep track of data related to a session, a *CommandCache* is added to each replica that stores all the queries that were executed on behalf of a session s.

For `INSERT`, `UPDATE` and `DELETE` commands, we keep track of modified rows affected by associated `SELECT` queries. For example if the session executes the command where $a1$ is the primary key (key):

```
INSERT INTO T1(key, v1) VALUES  (a1, b1)
```

we store the following query in *CommandCache[s]*:

```
SELECT * FROM T1 WHERE key = a1
```

This transformation creates a query that tracks the accessed key $a1$.

The entries in the *CommandCache[s]* precisely identify the data accessed by a session. To recover the rows associated with a session we just have to execute the queries without any projections (`SELECT(*)`) and without any aggregations (without any `GROUP BY`). Our database implementation is based on Cassandra where queries are limited to a single table (no joins).

To keep the *CommandCache* small, we don't keep queries for a given session that are subsumed by more general ones. We also keep queries only for data that is actually replicated by each site. A background garbage collection mechanism deletes queries for sessions that have been moved to other datacenters.

To support session consistency, our current implementation can only run queries for a *stoken* at only one replica at a time. We keep track of the location of this replica on the *stoken* itself (the *current replica* field) and every site also keeps track of sessions it is serving.

Session-Aware Reconciliation

We leverage the *stoken* to detect when a client switches between replicas (e.g. when it moves between edge replicas n_s, n_d as shown in Figure 4.4a). When a replica receives a query it checks the *stoken*. If the ID of the replica servicing the query does not match the replica ID in the

stoken then this is indicative that the client has switched replicas and the reconciliation process needs to start.

(a) non-siblings (b) siblings

Figure 4.4: Session transfer s between n_s, n_d

When the switching process for a session is initiated by a client request received with a *stoken*, the *status* field on the *stoken* changes to *Switching to* n_d where n_d is the ID of the destination replica. A separate thread then fetches the session data to n_d. In the meantime if the client requests are processed on a another edge n_e, n_e will wait for the switching process on n_d to finish and then fetch the data from n_d. n_e can detect that there is a switching process in progress if it has access to the *stoken* and it's *status* field.

To assure session consistency, when a switching process is triggered on n_s, n_s's SessionStore replica will not process further commands for that session. Furthermore, requests for the session are delayed on n_d until the switch is complete. When the switching process is finished, it is reflected in the *status* field. If during the switching process the client moves to a another edge n_e, n_e will wait for the switching process on n_d to finish and then fetch the data from n_d.

During a switching process, n_d sends a request to the source replica asking for all rows modified or accessed by the session s. Having recorded all the queries executed by s, the source replica re-executes theses queries from its *CommandCache[s]*. It will then transmit the resulting rows as well as *CommandCache[s]* to the destination replica.

Using queries to find accessed rows has the benefit of aggregation. While for writes we map every row modification to a separate query, for reads which usually dominate the accesses to the database, a single query can track many rows. Replication is done at full row level irrespective of columns projected in the select query.

Table 4.1 illustrates a database table *ratings* that keeps track of personalized movie ratings

Viewer	Movie	Version	Rating
John	WALL-E	825968c0-195d-5d569c585662	10
Bob	Lion King	7adf7210-1958-59e16851d966	9
Susan	Bambi	6833c850-1958-59e16851d966	8
Anna	WALL-E	38400000-b23e-000044004725	10

Table 4.1: Sample table *ratings* on n_s

Viewer	Movie	Version	Rating
John	WALL-E	d33d7fe0-195f-5d569c585662	8
Mark	Cars	8b5f2471-19a2-59e168456212	9
Sara	Peter Pan	1263ca45-1912-59e36a58d990	8
Anna	Lion King	15460690-de22-a80b17057344	9

Table 4.2: Sample table *ratings* on n_d

on n_s. Column *viewer* is the primary key and column *version* is added by SessionStore to determine ordering between updates to the same row. Now suppose that the following queries had been executed by session s on n_s:

```
SELECT * FROM Ratings WHERE viewer = ANNA
```

```
INSERT INTO Ratings(viewer, movie,rating) VALUES(Susan, Bambi, 10)
```

CQL *INSERT* provides upsert - that is, inserting a primary key that already exists will update the values associated with that key. SessionStore keeps track of updates (i.e., *INSERT*, *UPDATE*, *DELETE*) by turning them into *SELECT* queries so the *INSERT* command above is saved in the CommandCache as:

```
SELECT * FROM Ratings Where viewer = SUSAN
```

When the session switches from n_s to n_d, the two queries are executed on n_s (for the `INSERT` command, the associated `SELECT` query is executed) and only the third and fourth rows are copied to the *ratings* table on n_d as these are the only rows that match the recorded queries.

Optimizations

We implemented two optimizations to SessionStore's session-aware reconciliation that take advantage of data locality between different replicas and of SessionStore's hierarchical structure.

Δ-list optimization The previous algorithm can be optimized when many clients are accessing the same rows on different replicas. The Δ-list optimization does not copy data that is already present at the destination replica. During a session switch, the destination, n_d, selects all primary keys and latest *version* for data belonging to an application and sends them to n_s. With this data n_s can then calculate rows accessed by a session that are either not already on n_d or have a newer version.

Sibling optimization Finally, we provide a special *optimized-sibling-transfer* algorithm that works when the source and destination share a common parent node. This optimization takes advantage of the fact that in our design, all rows read by a node are also first replicated on its parent. In addition, the push daemon running on nodes, periodically propagates data written on a child onto the parent node. To take advantage of the fact that data written on n_s gets propagated on to n_p by the push daemon and considering the fact that usually the link from n_d to n_p (shown in Figure 4.4b), is closer and has more bandwidth in the underlay network than the link from $n_s to n_d$, this optimization tries to minimize the data transfer between n_s, n_d. Whenever a row is modified or created on n_s, the *push daemon* running on n_s will push data to n_p. If n_d and n_s both have n_p as parents, then rather than fetching all the rows accessed by s from n_s, n_d can fetch new and updated records from the parent node n_p. During a switch, only the rows that have not yet been pushed from n_s to n_p need to be replicated from n_s to n_d. Other rows can be accessed from n_p. Finally we synchronize any row on the destination that matches any query on the *CommandCache[s]* by fetching an update from the parent.

In addition, we have to make sure that existing rows on n_d that also exist on n_s and have been accessed by s on n_s, are at least as up to date. This is because some rows on n_d might be outdated. We transmit *CommandCache[s]* from n_s to n_d and compare each query q_s in *CommandCache* of n_s with the query q_d in the *QueryCache[s]* of n_d. If $q_d \subseteq q_s$, then q_d is immediately re-executed on the parent and the data matching the queries will be sent to n_d. If $q_s \subset q_d$ then q_s is executed on the parent and the rows will fetched by n_d.

4.3.3 Failures

If a source replica fails when a destination is replicating state from it, SessionStore has to wait for the source to be available again and continue the transfer for the rows that it could not already replicate. The application is informed about any issue through an exception. The application can then decide to wait and retry, or invalidate the session and restart. Combining proactive replication to a few replicas with SessionStore's reactive approach is an avenue of future work.

4.4 Experimental Evaluation

In this section we evaluate the performance of SessionStore and compare it to other alternatives for providing session consistency on a network of distributed replicas.

4.4.1 Platform

We conduct our experiments on an emulated hierarchical edge deployment shown in Figure 4.2. Our topology consists of a cloud datacenter, and two mobile networks each with a datacenters at its core, and one or two additional datacenters at edge location such as base stations.

Each (emulated) datacenter is implemented in a separate computer with $16GB$ of RAM and 8 CPU cores that runs either an instance of SessionStore, PathStore, or unmodified Cassandra, as well as an instance of Apache Tomcat. The network between the datacenters is emulated by using Linux's Traffic Control. Each link has a bandwidth of $1Gbps$. We assume that the underlay IP network has the same topology as the replica topology. This means that the point to point RTT between e_1, e_3 will be $t_1+t_2+t_2+t_1$. Unless stated otherwise, for the network latency between different datacenters, we optimistically assume two-way latencies $t_1 = 2ms, t_2 = 20ms$. These relatively low latency values tilt the comparison against SessionStore and in favor of Cassandra, which is more adversely affected by higher latency. Finally, requests are issued by clients running on additional computers that connect to one of the edge datacenters (e1, e2, e3) with negligible latency.

4.4.2 Workloads

Our evaluation uses a combination of locally-developed micro-benchmarks and RUBBoS [15], a benchmark application that models a discussion board. While RUBBoS was designed as a web benchmark, we use it because its data access pattern is representative of a typical multi-user application in three aspects: (i), it involves a large amount of state; (ii), it includes both read and write queries; and (iii), only a small fraction of the application's state is relevant to any given user.

The original RUBBoS benchmark is limited to text comments (1 KB in average), which are small compared to modern media-sharing standards. To better mimic the expected behaviour of a modern social media application, such as Snapchat or Instagram, which allow users to upload short videos and images, we create two new versions of the benchmark by adding an extra 10 KB or 100 KB of data to each comment to simulate a small and medium multimedia attachment. This increase the total size of the RUBBoS database from 540 MB to 23.9 GB and 240 GB, respectively. We used a RUBBoS database populated by over 2.34 million comments, 12000 stories, and 500000 users.

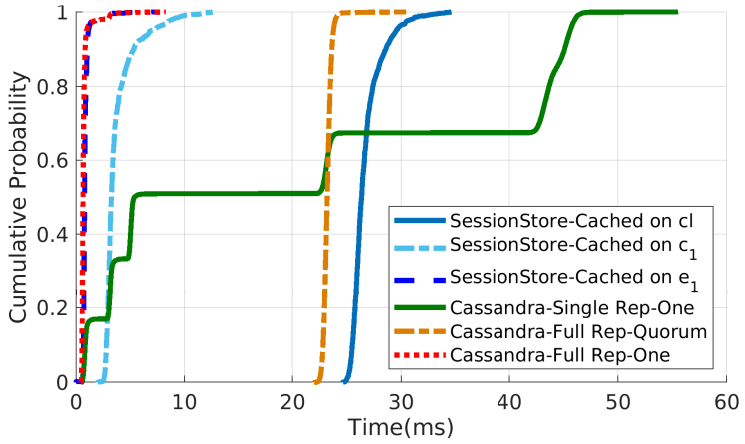
We used the Java Servlet-based RUBBoS implementation which was originally designed to store its state in relational database. We ported this code to use SessionStore instead. The ported benchmark uses eight tables and consists of roughly 40 different queries including `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

4.4.3 Results

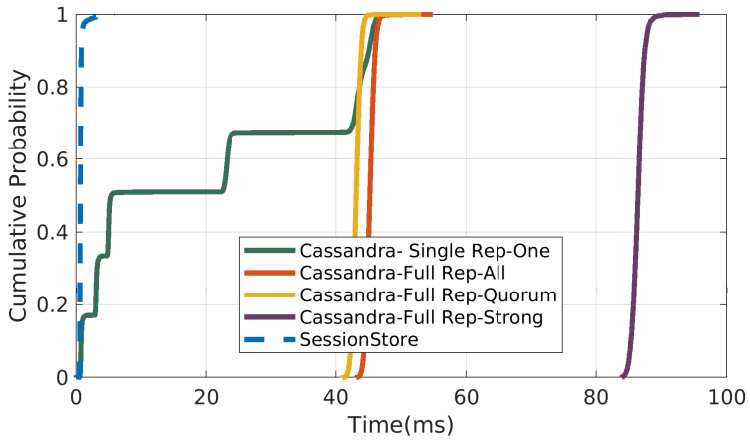
We next present results that quantify the overhead of keeping track of session information, the benefits of session-aware reconciliation, compare the approach to alternatives that enforce stronger consistency as the cost of higher overhead, and explore the sensitivity of SessionStore session-aware reconciliation protocols to the number of queries in the command cache.

Session Tracking Overhead

To measure the cost of keeping track of session state, we compared the latency for reading and writing single $1KB$ row on e_1 with SessionStore. The experiment is repeated for 10000 different



(a) Reads



(b) Writes

Figure 4.5: CDF of latency required to read and write a $1KB$ row.

rows. Figure 4.5a shows a CDF of the read latencies for SessionStore in three different scenarios that assume the rows being read are already replicated on e_1, c_1, and c_l, respectively. The read latency for SessionStore is indistinguishable from PathStore(not shown), which indicates that the session tracking overhead is negligible. As expected, the figure shows that replication at the edge reduces read times dramatically. The average time to read a row already available on the edge was 0.9 ms, compared to an average of 4.65 ms and 26.2 when the row had to be fetched from the core and cloud, respectively.

Figure 4.5b shows a CDF of the write latency for SessionStore. There is only one configuration as all writes are performed on the local replica (e_1). The average write time is 0.73 ms, and is similarly indistinguishable from write time in PathStore(not shown).

Session Migration

We use the RUBBoS benchmark to evaluate the costs in terms of latency and bandwidth of enforcing session consistency when a user switches between two replicas as a result of mobility. We consider four different approaches: Full-replica reconciliation, session-aware reconciliation, $\Delta-$list optimization, sibling optimization.

We use the client emulator in the RUBBoS package to simulate 100 clients connected to replica on e_1 that are browsing and commenting on the RUBBoS bulletin board. The client emulator sent HTTP requests to Servlets running on the edge node e_1 which generated 2203 queries on the SessionStore replica on e_1. This resulted in SessionStore fetching data from cl and replicating it on e_1. A total of 1.86 MB for the text-only version of RUBBoS, and 22.7 MB and 220.3 MB for the versions with the small and medium multimedia attachments was transferred. On average, each RUBBoS query resulted in 13.4 rows on the database, which exemplifies the benefits of using a query-based approach compared to tagging each row.

Table 4.3 shows the latency and data transferred for different replica reconciliation scenarios for a client that moves from e_1 to either e_2 or e_3. The experiment is repeated 100 times, once for every client. We first consider the worst case where sessions move from e_1 into a cold e_3 replica that does not have any data. Full-replica reconciliation (first column) requires sending the full $1.86, 22.7, 220.3$ MBs of application data which takes 562.1 ms, 2.24 s and 10.7 s for each of the three configurations of the benchmark. In contrast, session-aware reconciliation

		Full Reconciliation (e_1, e_3)	Session-Aware (e_1, e_3)	Δ-List (e_1, e_3)	Neighbor (e_1, e_2) No users on e_2	Neighbor (e_1, e_2) 100 users on e_2
Default RUBBoS rows	Data Transfer	1.86 MB (40 KB)	187.25 KB (73.3 KB)	141.22 KB (45 KB)	14.2 KB (1.2 KB)	198 KB (38 KB)
	Time	562.1 ms (20 ms)	343.9 ms (57.8 ms)	288.52 ms (45 ms)	15.9 ms (1.4 ms)	62.7 ms (17.9 ms)
Added 10KB to each row	Data Transfer	22.7 MB (0.10 MB)	2.56 MB (563.1 KB)	1.22 MB (220 KB)	16.3 KB (2.2 KB)	1.16 MB (70 KB)
	Time	2.24 s (32 ms)	534.0 ms (40.59 ms)	330.4 ms (30.1 ms)	19.2 ms (2.4 ms)	76.86 ms (12.48 ms)
Added 100KB to each row	Data Transfer	220.3 MB (1.1 MB)	24.32 MB (6.6 MB)	15.9 MB (4.2 MB)	13.1 KB (4.3 KB)	10.7 MB (1.7 MB)
	Time	10.7 s (76 ms)	1.09 s (169.1 ms)	741.51 ms (105.1 ms)	20.8 ms (3.7 ms)	153 ms (25.5 ms)

Table 4.3: Average reconciliation time and data transfer for a Rubbos client. Standard deviation in parenthesis.

(second column) only transfers an average of $0.18, 2.56, 24.32$ MBs of data, which takes only $343.6, 534, 1090$ ms. This major improvement, that is mainly because of the data overlap in the data accessed by different clients, represents a reduction in data and latency of close to 90%, and is strong evidence of the benefits of leveraging session-aware reconciliation for server applications where only a fraction of the replicated data is relevant to a given client.

The Δ-list optimization (third column) further improves these numbers. In this experiment, we assume that a different set of 100 clients send requests to e_3 before the transfer. e_1 calculates the rows it needs to send to e_3 for each user and on average transfers 141.2KB's of data. For the three version of RUBBoS, Δ-list optimization only transfers $0.14, 1.22, 15.9$ MB of data in $288, 330, 741$ ms, which is an additional $22-35$ percent improvement in each scenario compared to the Session-Aware approach. Δ-list performs best when each row contains a lot of data and saves on bandwidth and transfer time by not sending those rows that are already on the destination.

We next evaluate the benefits of the sibling optimization when a single client moves from e_1 to e_2. We first consider the case where there are no other users on e_2 (fourth column). In this particular application, many queries are common between sessions so more stale data has to be fetched from the parent. This results in only information about the queries transferred between the two nodes which is only 16.5 KB of data and takes less than 20 ms on average for the transfer (compared with 10.7s with the Full Reconciliation approach). This is extremely fast compared to other scenarios because no other data needs to be transferred between the

	Scenario	Average data transfer per row
Reads	SessionStore Fetch from cl	3245.8B
	SessionStore Fetch from c_1	1620.7 B
	SessionStore Fetch From e_1	0
	Cassandra Full Replication	0
	Cassandra Single Replication	1120.7 B
Writes	SessionStore	2346.8 B
	Cassandra Full Replication	6372.4 B
	Cassandra Single Replication	1213.6 B

Table 4.4: Data transfer

nodes. If the user executes their commands again, the data will be fetched from c_1 so the cost of fetching the data will be on demand and when the user requires it. Finally, we assume a scenario where another set of 100 users run the same application (and hence run similar queries) on e_2. Common queries between the moving user and users already running on e_2 may result in synchronizing data from that the parent. This on average increases the transfer time to $62, 76, 153$ ms and a further $0.19, 1.16, 10.7$ MB of data is transferred between e_2, c_1.

Comparisons with Eager Replication and Strong Consistency

In this section, we explore three alternative ways in which *unmodified* Cassandra could be deployed on our network of six datacenters that use eager replication or strong consistency to guarantee session consistency for a client that can move between replicas. In these configurations, all six datacenters form a single Cassandra ring, and each Cassandra server creates point to point connections to other servers using the underlying IP network.

Full Replication-All, uses a replication factor of six and Cassandra's *All* consistency model which requires all replicas to respond before a write operation returns. We can then use Cassandra's *One* consistency model for the reads which will fetch data from the local replica. *Full Replication-Quorum*, also uses a replication factor of six and Cassandras *Quorum* consistency model. This configuration requires a responses from a quorum of replicas for both reads and writes. *Single Replication-One*, uses a replication factor of one, and relies on Cassandra's standard hashing algorithm to uniformly distribute rows among replicas in the Cassandra ring. Reads and writes in this configuration involve a single server. Finally to compare to a strongly

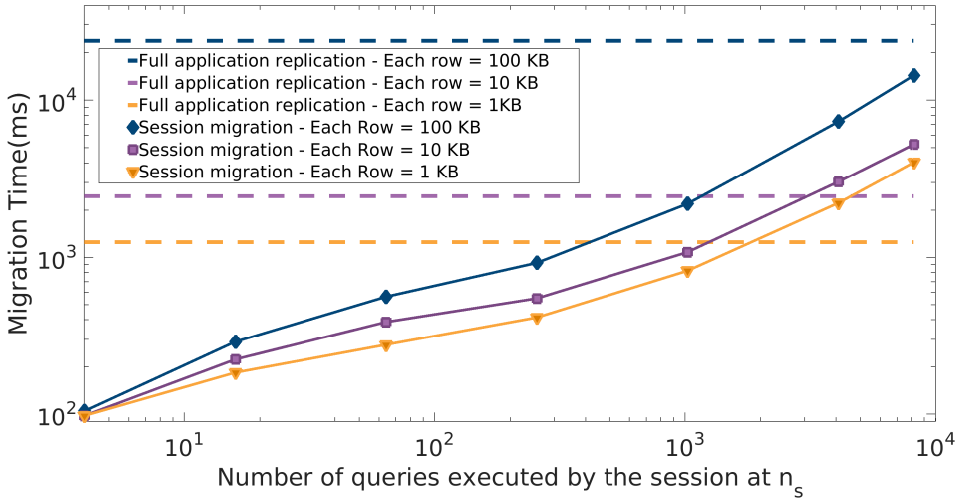


Figure 4.6: Comparing session reconciliation(solid lines) and full application data reconciliation that consists of 10000 rows (dashed lines). Both axes are in logarithmic scale

consistent database, *Full Replication-Strong* acts similarly to Full Replication-All with addition of linearizable consistency through the use of Cassandra's light weight transactions and the Paxos protocol [86].

Figure 4.5 shows the CDFs of the time required for writing or reading a single $1KB$ row on e_1. The experiment is repeated for 10000 different rows. Table 4.4 shows the average data transferred aggregated across all links to store or read a $1KB$ row for the various configurations. All Cassandra alternatives perform poorly, which is hardly surprising given that Cassandra is not designed to be used in this manner and requires communication between different servers. On the other hand, our results are optimistic as real-world edge deployments will likely consist of a much larger number of datacenters.

Full Replication-All handles reads very well, but pays for it with high latency and bandwidth cost for writes. *Full Replication-Strong* performs even worse as the Paxos protocol needs additional rounds of communication between nodes. *Full Replication-Quorum* is a little better for writes, but much worse for reads. Finally, *Single Replication-One* read and write performance varies widely between rows based on their random allocation across the various datacenters. In comparison, SessionStore provides low latency for writes and reads, particularly in cases where the rows are already available on e_1 or c_1, and uses much less bandwidth.

Size of Command Cache

We evaluate the benefits of session-aware reconciliation as a function of the fraction of data in the replica that is relevant to a session and the number of queries used to track this data.

Figure 4.6 plots the latency to reconcile 10000 rows when a session moves from $n_s = e_1$ to $n_d = e_3$. We consider two reconciliation strategies: Full reconciliation, depicted by the dashed lines, that does not keep track of data accessed by individual sessions, and as a result all 10000 application rows have to be copied when the client moves between replicas. This becomes especially expensive when the amount of data stored in each row increases $(1KB, 10KB, 100KB)$. Session-aware, displayed as solid lines, uses the *CommandCache* to keep track of rows accessed by the client that need to be moved between the replicas. We vary the number of commands executed by the client between $1, 8192$ and we assume each command only affects a single row. When the mobile client accesses only a fraction of the total data used by the service it is more beneficial to track session data. However, as the number of queries for a session increases, the overhead also increase because each query in the *CommandCache* has to be fetched and executed. As expected, the benefits of session-aware reconciliation is more distinguishable as the as the amount of data in each row increases. As shown in the Figure, when the rows are $1KB$, after around 1200 commands executed at n_s, it takes less time to transfer the full application data (orange lines). But when each row contains $100KB$s, even by executing 8192 commands for the session at n_s, it is still faster to use session-aware reconciliation (blue lines).

Figure 4.8 compares the transmission and processing delay for the session-aware reconciliation (left bar) and the sibling optimization (right bar) as a function of the number of queries executed by the session. The figure shows that the for session-aware reconciliation, when we increase the number of queries that a session has executed on n_s, the processing time to gather the data for that session at n_s increases. This is expected behavior as we have to execute all queries again and the transfer all the retrieved data to n_d. In comparison, the sibling optimization (hatched bars in Figure 4.8) does not re-execute all the sessions' commands on n_s as this done on demand after the transfer at n_d.

RTT Figure 4.7 illustrates the effect that increasing RTT has on reconciliation latency for a client that moves from $n_s = e_1$ to $n_d = e_3$. The red line in this Figure shows the time it takes

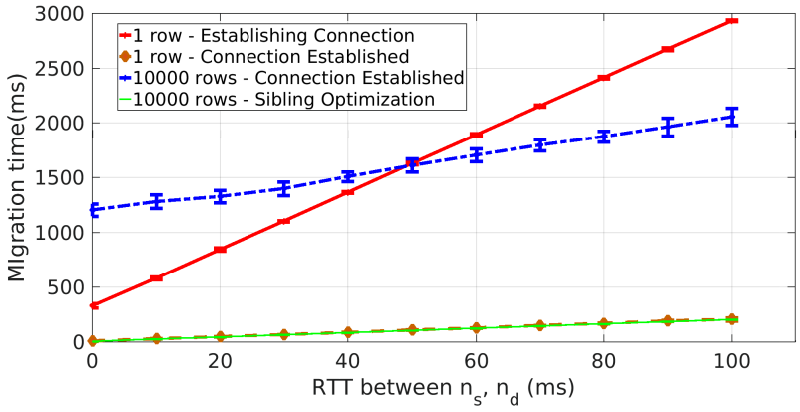


Figure 4.7: Session transfer time when the latency between n_s, n_d increases. The Figure includes charts for session aware transfer and sibling optimization.

to create required connections between n_s, n_d and then transfer a single row. This is the worst case and it emulates a scenario where the session moved to a destination node that did not already have an established connection with the source. This can happen when a client moves between those datacenters in the network that typically see very little traffic and transfers between them. The orange dashed line illustrates a similar process except n_s, n_d already have a pre-established connection. As a result, the transfer time is significantly lower. We expect this to be the common case where transfers happen between a small set of neighboring nodes. The blue line depicts the amount of time required to transfer 10000 rows between n_s, n_d using a single `SELECT` query with a cached connection as a function of RTT between n_s, n_d. The green line (which closely tracks the orange line) illustrates the transfer time when $n_s = e_1, n_d = e_2$ are sibling replicas. In this scenario, n_d does not include any other rows for the application and n_s does not have any dirty data (data not pushed to its parent replica). Hence, when s moves between n_s, n_d, no data transfer is required between n_d, n_s and only the queries executed by s are transferred to n_d and the session token is also updated accordingly. If s executes any query, data is fetched from the parent replica on demand. Using the sibling optimization is a major improvement to the session aware scenario with no optimizations as it involves minimal data transfer between siblings. This is particularly useful in scenarios where the link between a n_d, n_p is cheaper in terms of bandwidth cost compared to a link between n_s, n_d in the underlying network.

Figure 4.8: Reconciliation time breakdown (in percentile) for regular session aware (left bars) and sibling optimization (right bars - hatched). The x-axis represents the number of queries executed at n_s

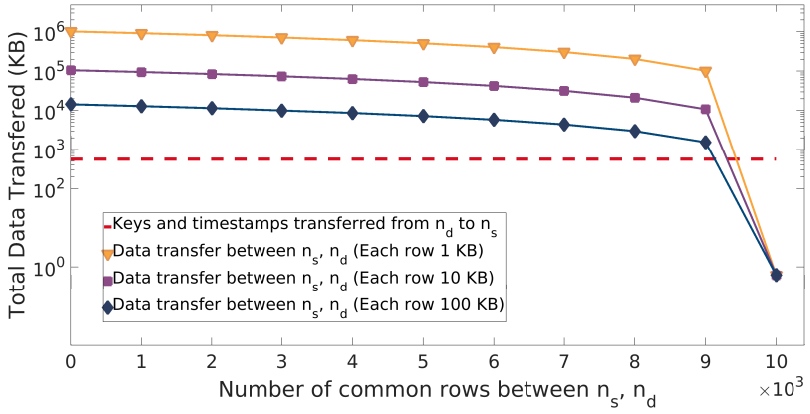


Figure 4.9: Data transfer between n_s, n_d when n_s sends all primary keys of an application along with their timestamps to n_d. Note that the Y-Axis is in logarithmic scale

Data Overlap Figure 4.9 illustrates the performance of the Δ-list optimization as we vary the number of common rows between e_1 and e_3. At the beginning of the processing for session s_i, we send all primary keys and their respective timestamps present on $n_d = e_3$ to $n_s = e_1$. We call this list l_d. We assume the application on n_d consists of 20000 rows. n_s receives l_d and only sends rows accessed by s_i that are either not in l_d or have a newer timestamp. As we increase the number of common rows between n_s, n_d (rows that have the same key and timestamp), the data bandwidth consumption on the link between n_s, n_d decreases. Comparing the blue line when each row is $100KB$ with the orange line where each row is $1KB$, illustrates the

effectiveness of our optimization when the amount of data needed to be sent increases. The Figure shows that the more common rows there are between n_s, n_d the less data needs to be transferred from n_s to n_d. The red dashed line indicates the overhead of sending l_d. Note that because we are only sending the keys and their timestamps, the overhead remains relatively small. The average processing time required to generate l_d is 370.8 ms.

4.5 Chapter Summary

A key tenet of fog computing is the ability for clients and application functions to be redirected seamlessly across the different edge datacenters hosting the data replicas of a service or application. In this chapter, we presented SessionStore, a novel storage system that provides session consistency even when the client switches between replicas in different edge locations. Our session-aware reconciliation algorithms enforces session consistency at minimal costs, by tracking the accessed or affected keys by a session and then performing fine-grain reconciliation on the destination replica with minimum overhead. Our results show that our approach provides session consistency at a fraction of the latency and bandwidth costs of a system with eager replication or strong consistency, with minimal transfer costs.

Chapter 5

Feather: Hierarchical Query Processing on the Edge

Data is increasingly stored in databases distributed across a wide area, separated by comparatively high-latency and bandwidth-limited links. In particular, in edge computing and IoT applications data is generated over a wide geographic area and is stored near the edge or across a hierarchy of datacenters. Querying such geo-distributed databases traditionally falls into two general approaches: push incoming queries down to the edge where the data is, or run them locally in the cloud. For example, stream processing reduces aggregation bandwidth while pushing data to the cloud. Alternatively, eventually-consistent databases can execute the query locally, providing a stale answer quickly.

Consider a hypothetical Industrial-Internet-of-Things (IIoT) application deployed over a 3-tier network [36], as shown in Figure 5.1. Machines on the factory floor generate large volumes of data, used locally for low-latency process control decisions on the production line. The data is also forwarded to a local aggregation center, perhaps one per factory or a group of factories, where more resource-intensive predictive maintenance models can be applied, and where latency requirements are less stringent. Finally, data is forwarded from the core to a cloud server, where a management back-end shows a web dashboard with global production status and inventory. It can also be used for training machine learning on historical data, since more resources are available in the cloud. Similar unidirectional data flow is common in other settings, such as urban sensing [139, 57, 109], smart grid [130, 110], IoT and wearable devices [132, 137, 127], and healthcare [155, 22].

Data management in this geographically-distributed environment can be challenging: network links have limited bandwidth, high latency variation, and can intermittently fail . Luckily, many applications exhibit strong locality: most reads and writes can be done locally, and changes need not be immediately replicated to the entire network. Therefore, a common scheme provides fast local reads and writes using a high-performance local data store (e.g., one per factory floor), then periodically propagate data it upwards using a best-effort or on-demand strategy [137, 149, 116, 111]. This scheme is eventually-consistent, handles link failures, and is relatively straightforward to implement and reason about.

This scheme, however, provides no guarantee on the freshness of data received from lower layers when executing read queries at the parent (e.g., cloud). Consider a read query initiated on the cloud by the management back-end in our example. Since the most up-to-date data

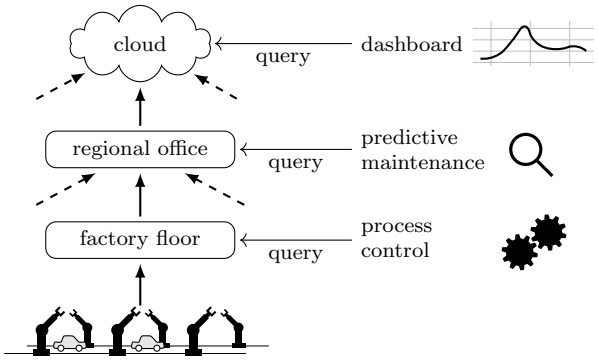


Figure 5.1: Example IIoT application components deployed over a 3-tier network. Data is collected from each factory floor and must be sent up the hierarchy. Process control functions run on the factory floor, since they require fresh local data and low latency. Predictive maintenance models consume more resources, but use staler data from multiple production lines. Global dashboard runs in the cloud, and requires balancing data freshness with answer latency.

is distributed over factory floors and local aggregation centers, it is difficult to guarantee the freshness and completeness of the read query.

One common approach to handling such queries is to execute them on the cloud's local replica: since all data will be eventually replicated to the cloud, we can answer the query using the data that has already been replicated. This provides an answer very quickly, but it might be very stale; there are no guarantees on data freshness. The other extreme is to fetch up-to-date data from edge devices to the cloud where the results can be aggregated [31]. This results in fresh data but incurs high latency, additional load on edge nodes, more bandwidth usage, and may even miss data if an edge is unreachable. Another alternative is stream processing: queries are decomposed to graphs of operators, and are distributed across the edge network. Data can be processed and aggregated as it is streamed from the edge towards the cloud. However this approach requires deploying, coordinating, and executing operators across various datacenters. Moreover, distributed stream processing across edge networks is difficult due to unreliable links, frequent re-configurations, and high latency [145, 51]. Stream processing therefore incurs high setup and ongoing costs, and is therefore better suited for processing a small set of recurrent or continuous queries that is known in advance. In contrast we are interested in enabling ad-hoc queries and data exploration.

We present a hybrid approach for efficient on-demand global queries with guaranteed freshness by exploiting the underlying hierarchical structure of edge networks.

Feather is an eventually-consistent tabular data management system for edge-computing applications that allows users to intelligently control the trade-off between data freshness and query answer latency. Users can specify precise freshness constraint for each individual query, or alternatively a deadline for the answer. We then execute this query over a subset of the network, using local replicas in intermediate network nodes as caches to avoid querying edge nodes. The result set is guaranteed to include, in the absence of failures, all data that is at least as fresh as the specified limit; we further return an actual freshness timestamp telling users how up-to-date the query answer is.

To deal with intermittent link errors, Feather also allows returning of partial answers, and provides users with an estimate of how much data was missed by the query. Our Feather prototype supports features typically available in high-performance tabular data stores: filtering, aggregation, grouping, ordering, and limiting of the result set. This allows existing read queries that currently run on centralized tabular databases to be easily ported.

We evaluate Feather by emulating a geo-distributed service that is deployed on an edge network, and use traces of geo-tagged data to mimic the request patterns of geo-distributed clients. In controlled experiments, we evaluate the effect of network, topology, and Feather parameters on the trade-off between latency, staleness, bandwidth, and work at edge nodes. We validate our findings by conducting a real-world experiment where we instantiate an edge network that spans North America, Europe, and Asia to process local Twitter data. Feather is able to combine the best of cloud and edge execution, answering queries with a fraction of edge latency, providing fresher answers than cloud, while reducing network bandwidth and load on edges.

5.1 Background

For clarity, we first define key concepts we will use throughout the chapter, and then review several examples of edge-computing scenarios where ad-hoc querying mechanisms can be beneficial.

Applications

Edge computing plays a key role in many upcoming application scenarios. We focus on a common scenario where data collected from from end-user devices or sensors is initially stored locally, and must be later forwarded to higher layers for ad-hoc querying and analysis. We give three such examples.

First, in advanced industrial automation scenarios, resource-limited IoT devices can log huge amounts of data metrics, but store it locally to save on bandwidth and other costs [37, 36]. Figure 5.1 is an example for such a scenario. An efficient ad-hoc global querying mechanism can allow remote monitoring and management without incurring significant bandwidth or computation overhead. For example, if a fault in certain class of equipment is suspected, an operator could query specific relevant metrics for that equipment for the last minute, hour, or day. Second, smart cities use distributed sensors to collect data used for pollution monitoring [139], transportation and traffic control [109, 38], and healthcare [155, 22]. These sensors produce large amounts of valuable data and sensory information, not all of it needed to be streamed in real-time for processing. Instead, data is often uploaded in batches. Some queries can be ad-hoc, in response to specific events. For example, an operator could query for the number of pedestrians and bikes in a specific area affected by a car accident. Finally, utility companies have been using smart meters to aggregate usage information from customers [130, 110]. While these meters periodically send data to a centralized location for coarse grained analysis, on-demand querying could allow for fine-grained analysis, which in turn could enable more responsive resources management.

Eventual-Consistency and Tabular Databases

While the above scenarios benefit from and efficient and accurate global querying mechanism, in practice strong consistency over a large geographical area is difficult to achieve [100, 87]. Data-heavy edge computing applications are therefore built to accommodate relaxed consistency guarantees such as eventual consistency [152]. Updates are not propagated immediately to all replicas, but are instead propagated periodically or on-demand. Similarly, edge computing applications often rely on distributed tabular or key-value stores, rather than classic

relational databases. Joining tables and transaction support can be prohibitively expensive in distributed settings [154], particularly when the volume of data is large. While relational and transactional databases in geo-distributed settings is an active area of research, many current high-performance distributed databases are tabular or key-value stores [106].

5.2 Design

In this section we describe the design considerations of Feather: an geo-distributed tabular data management systems for hierarchical networks that supports on-demand, global queries with guaranteed, user-specified lower-bound on data freshness.

Feather offers applications two types of queries, local and global. Both types can access data written locally and by descendent nodes, but differ in their guarantees. *Local queries* are fast reads and writes executed directly on the high-performance local data store. This is the type of queries ordinarily performed by applications, and are also supported by several edge-centric eventually-consistent distributed databases [111]. *Global queries* are the main contribution of Feather. These are on-demand read queries that provide user-specified freshness guarantees. When executed on a node, the query response will be computed from recent local and descendant data, up to a user-specified limit. By carefully keeping track of update times for data stored at intermediate nodes, Feather avoids querying remote edges, allowing for faster responses and conserving bandwidth.

Beyond the freshness guarantee, Feather provides additional features such as setting query deadlines, estimating result coverage, and handling link failures gracefully.

5.2.1 Semantics of Global Queries with Guaranteed Freshness

We first explain the querying semantics and the guarantees provided by Feather on-demand query mechanism.

Feather global queries include a freshness constraint provided by users, which we call *laxity* L. This constraint guarantees that data created up to a time t requested by the user will be in the result set, relaxing the freshness requirements on data.

Formally, if *query time* T_q is the time the query was sent for execution to the system, Feather

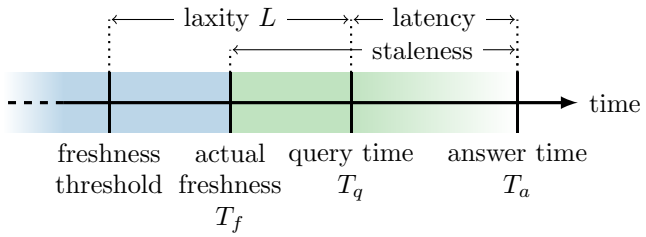


Figure 5.2: The freshness guarantees for Feather global queries. Actual freshness T_f is guaranteed to be between $T_q - L$ and T_a. Any row created before T_f (blue) is guaranteed to be included in the results, while rows created after T_f (green) may or may not be included.

guarantees that the set of rows used to process the query contains all data updates (insertions, deletions, and updates) that occurred before the *freshness threshold* time defined as $T_q - L$. While laxity gives a limit on data freshness, query results can in practice be more fresh than the limit. Thus query answers also include an *actual freshness* time T_f: all data updates that happened before T_f are included in the answer. Note that updates that happened after T_f may also be included in the result, but cannot be guaranteed to be so. The exact value of T_f depends on which data has already been replicated up the hierarchy. Additionally, even if we set $L = 0$ and had a fresh copy of all data, the answer could still be slightly out of date: queries take time to execute and data takes time to transfer between datacenters. Note that it is possible, though rare in practice, that $T_f > T_q$. Hence, we define *staleness* as the difference between answer time T_a and the actual freshness time: $T_a - T_f$. In summary, Feather guarantees:

$$T_q - L \leq T_f \leq T_a \quad .$$

Figure 5.2 illustrates these semantics.

For example, consider a dashboard query from the industrial monitoring application (Figure 5.1) that retrieves the average power consumed by arm robots in the last 10 minutes (600 seconds). Given the needs of the application, we may decide to allow the data to be out of date for up to 30 seconds, but no more. We therefore execute the query:

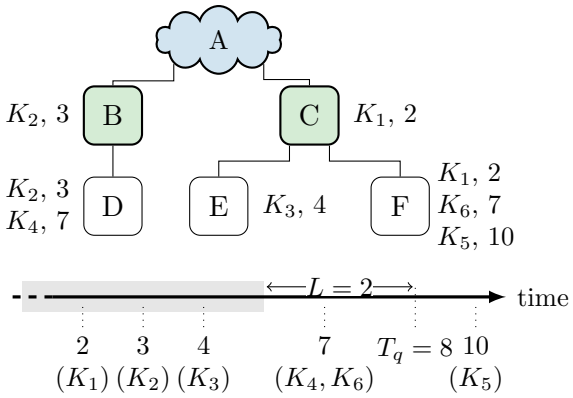


Figure 5.3: Edge network with 6 rows K_1 to K_6, with row update times (numbers next to keys). A query submitted to A at time $T_q = 8$ with laxity of $L = 2$ must retrieve all keys updated before time $T_q - L = 6$, and must therefore access nodes B, C and E, but not D and F.

```
SELECT AVG(power) FROM hardwareStats

WHERE machine = 'arm robot'

      AND timestamp >= NOW()-600

LAXITY = 30
```

This query asks for the average power in all rows created up to 600 seconds before query time T_q whose `machine` is `arm robot`. The laxity constraints guarantees the average includes all rows created 30 seconds before query arrived at time T_q, and perhaps even more recent rows. Suppose this query took 2 seconds to process and answer, so $T_a = T_q + 2$ seconds, and the result includes all data up to 20 seconds before T_q. Then we have laxity $L = 30$ seconds, actual freshness is $T_q - 20$ and staleness is $T_a - T_f = 22$ seconds.

By tuning the laxity constraint, system operators can fine tune the trade-off between query response time and freshness. Higher laxity thresholds can result in faster response latency and reduced bandwidth. To illustrate this, suppose the state of the system is as shown in Figure 5.3. A is the cloud, B and C are core nodes, and D, E and F are edge nodes. Power consumption events (rows K_1 to K_6) are created on the nodes D to F, and some rows such as K_1 have already been replicated to parent nodes. If a global query to retrieve all rows was executed at time $T_q = 8$ starting from node A with a freshness threshold $L = 2$, then Feather must guarantee that rows K_1, K_2, K_3 will be in the answer set. As a result, at least nodes B, C, and E will have to be queried because they all have rows that should be in the answer set, while F's last

propagation time to C is recent enough and therefore we need not query it. Suppose instead we were to execute the same query at the same time, $T_q = 8$, but with a more permissive laxity $L = 4.5$. In this case, it would be sufficient to query only nodes B and C to obtain K_1 and K_2, resulting in a faster response though staler data.

Returning the previous IIoT example: suppose we allowed laxity of $L = 120$ seconds, and received an answer from the cloud's local replica in 15 milliseconds with actual freshness of 90 seconds behind T_q. In such a case, staleness will be $T_a - T_f = 90.015$ seconds.

Our freshness guarantee is similar to formal treatments such as Δ-atomicity [59] and t-freshness [125]; we discuss these in Section 2.0.3.

5.2.2 High Level Design

Feather's assumptions are common in geo-distributed, eventually-consistent databases [116, 111, 125, 65, 124, 42]. As described in Section 5.1, we assume the system is deployed over a set of geographically distributed datacenters (nodes) that communicate through an underlying hierarchical network, and have synchronized clocks[1]. The hierarchical structure of our system follows the hierarchical topology of the underlying network. The local replica at each node need only know about its parent and keep track of its children, which allows for both horizontal and vertical scaling of the system. As with any eventually-consistent database, users can be insert, update, or read data at any node, and it will be eventually propagated up the hierarchy.

Local Persistent Storage

Each Feather replica contains a high-performance persistent store, which contains both user data as well as metadata used by Feather[2]. Local queries from applications are served directly from this persistent store. Thus, data updates are written to this local storage by either applications running at the same datacenter or by Feather when replicating.. Similarly, data is read by user applications as well as by Feather. The local data store is configured to be

[1]Feather requires that clock drift be lower than the minimum one-way latency of any link in the network, i.e., up to a few milliseconds. Such accuracy is well within the capability of GPS clocks and IEEE-1588 [55] which reach microsecond accuracy for even low-cost hardware [153].

[2]In practice, such sharing can have security and performance isolation implications in production systems. While it is not fundamental to our design, for simplicity, we describe a single local data store that runs a single application.

strongly consistent, for example using quorum reads and writes. Since the local storage in each Feather replica is independent of other replicas, it is easy to scale it horizontally within a local datacenter.

Pushing Upstream

Feather replicas periodically push batches of new or updated ("dirty") data upstream to their parents. The update period and batch size are configurable, and control the trade-off between data freshness and resource usage (such as link bandwidth, CPU, and storage).

Each push update from child to parent also contains the *update time*, a timestamp that describes how fresh is the data being pushed. The update time is defined recursively. For Feather replicas on non-edge node, the update time is set to the minimum of latest update times of all its children. For an edge node, the update time is set to the current timestamp if the push includes all dirty data, or the update timestamp of latest row pushed up to the parent if the update needed to be batched. The update time is used by the querying mechanism (Section 5.2.3) to guarantee freshness, and is inspired by how stream processing systems such as Flink [33] and Google Cloud Dataflow [84] use *watermarks* to manage operators. Even when there are no dirty rows, replicas send empty updates to their parent with the update time as defined above. This helps avoid spurious child queries in the querying mechanism.

Consider for example in Figure 5.3. Node C maintains the latest update time for nodes E and F. If E's pushed data at time T_2, it would push an empty update with update time T_2 to C (since K_3 has not yet been created). Now suppose F pushes data at time T_3: it would push K_1 to C with update time T_3. The update time for C is therefore T_2, the minimum of the latest update times from E and F, reflecting the fact that C has not received any data updates from E after T_2.

5.2.3 Answering Global Queries

The hierarchical global querying algorithm provides the query semantics defined in Section 5.2.1. Unlike local application queries, which are served directly from the persistent store, the global queries described in Section 5.2.1 are processed hierarchically. Each replica first determines the set of children needed to execute the query, and then recursively sends it to each child. Once

ALGORITHM 1: The hierarchical algorithm for global queries with freshness guarantee L.

Input: query q, query time T_q, laxity L, current node n
Output: result R, actual freshness time T_f

1 Initialize set of accessed children $A \leftarrow \emptyset$
2 Initialize result R
3 **foreach** *child* $c \in$ children(n) **do**
4 **if** *last update time from child* $T_u(c) < T_q - L$ **then**
5 Add c to accessed children: $A \leftarrow A \cup \{c\}$
6 Send global query q to child c
7 $R_{loc} \leftarrow$ execute q on local store on rows not from A
8 Update result R with local results R_{loc}
9 Set freshness time T_f to latest update time: $T_f \leftarrow \min_c\{T_u(c)\}$
10 **foreach** *response* R_c, T_c *from child* c *of node* n **do**
11 Update result R with child result R_c
12 $T_f \leftarrow \min(T_f, T_c)$
13 Return results R and actual freshness T_f

all partial results sets are received, the replica merges them and its own local answer, and sends the result to the parent.

Algorithm 1 describes the hierarchical querying algorithm. At its core, this algorithm is a recursive, parallel tree traversal. When a global query is received at a node at time T_q with laxity L, we must first determine whether it can answer the query locally, or does it need to recursively query any of its children. This decision depends on the latest update time received from each child c, denoted $T_u(c)$. If this time is larger than the freshness threshold $T_q - L$, we know that the data we already have from that node is recent enough that there is no need to query that child or its own children. If $T_u(c) < T_q - L$, then the data pushed by the child to the parent is too stale and we have to visit the child. This decision then plays out recursively on each child, which returns the result to its parent.

Nodes execute queries in parallel: queries are first dispatched asynchronously to child nodes (line 6), then the local query is executed (line 7), and finally we wait for child responses and add incorporate them into the query results (line 11).

Finally, the actual freshness time T_f for the result is defined recursively, similarly to the latest update time. It is the minimum between the latest update time for the current node $\min_c\{T_u(c)\}$ (line 9) and the freshness T_f returned by each of the sub-queries (line 12). T_f strongly depends on the push period and the depth of the hierarchical network. We explore

this in Section 5.4.

5.2.4 Reversed Semantics for Providing Latency Guarantees

Recall the example query from Section 5.2.1. Suppose this time the query is executed by a web dashboard with latency SLA, so we must return an answer within 150 milliseconds even if it does not include all the freshest data. We therefore replace the freshness constraint `LAXITY = 30` with the latency constraint `DEADLINE = 150ms`, which guarantees that the response will be sent to the client after 150ms. As before, every response comes with actual freshness time T_f, allowing the dashboard to display the freshness of this response to the user. Coverage estimation (Section 5.2.5 provides additional information as to how much data was included.

Latency guarantee is achieved by treating nodes that did not respond in time as failed links (Section 5.2.6), and by a small modification to Alg. 1. When a child receives a global query from parent, it decreases the deadline to take into account latency between parent and child, plus some headroom for processing. In addition to executing the query in line 7, we also execute one query on the local dataset for each child that we contacted in line 6 (i.e., a local query for every child in A). Finally, for every queried child whose response was not received by the deadline, we instead use the result of the respective local query to update R in line 11 and T_f in line 12.

5.2.5 Result Set Coverage

With each query result, Feather provides analytical information on how many nodes participated in the querying process, how many data rows were included in the query, and an estimate of the number of updated data rows that were not included in the query due to freshness constraints or link errors.

The first two are easy to provide: each replica knows how many children it must query (Algorithm 1), and the total number of rows received from children and its own local queries.

Estimating the number of new or updated data rows requires us to track the rate of row updates (and insertions) received from each child. We estimate the rate of updates from each child node $\rho(c)$ as the mean rate from the last K updates. In addition to recording the last

update time, each replica also records the timestamps of the last $K + 1$ pushes received from children, and the number of new rows reported on the child.

Let $T_0(c)$ be the time of the last push from the child, $T_1(c)$ be the time of the push before that, and so on until $T_K(c)$ Similarly, let $R_i(c)$, $i = 0 \dots K$, be the number of new rows on the same child reported during the respective pushes. We estimate the rate of new rows from the sub-tree at the child as

$$\rho(c) = \frac{\sum_{i=0}^{K-1} R_i(c)}{T_0(c) - T_K(c)} \quad .$$

The estimated number of new or updated in a child c at time $t > T_0(c)$ is therefore $\rho(c) \cdot (t - T_0(c))$. When returning an answer, we includes the sum of estimates for all children.

As we show in Section 5.4, this simple estimation technique is sufficiently accurate for the datasets we tested on. If more accurate estimation is needed, more sophisticated time series prediction approaches can be used [56, 52, 93].

5.2.6 Handling Failures

Failures are common in geo-distributed environments. In particular, since networks are large and intermittent link errors are not uncommon, it is important to have queries running even if connectivity to some datacenters is lost. In addition to a monitoring system that keeps track of the health of Feather nodes between datacenters, our queries can timeout. When Feather produces results for a query, it includes information about what datacenters it was not able to access.

If a link to a child that must be queried has failed or a sub-query timed-out, then we cannot provide the freshness guarantee for that particular query. In such cases, Feather provides either a complete but less fresh answer that includes old results for the missing child, or a partial but up-to-date answer.

In the first option, the result set is complete not for the freshness guarantee requested by the user, but rather a less strict one that depends on the last update time for the child connected by the failing link. In other words, the answer is guaranteed to be complete for the actual freshness T_f, but this actual freshness is below the freshness threshold: $T_f < T_q - L$. For example, though the user requested data that includes no less than 5 minutes ago, the system

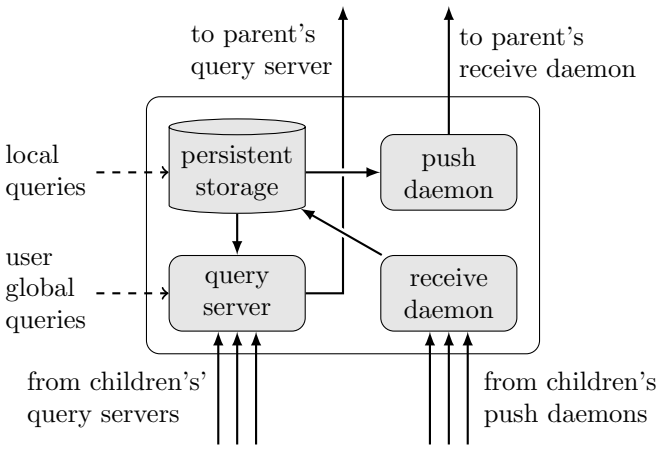


Figure 5.4: The main components of an Feather replica. Global queries are sent to the query server for execution. To provide fast local access, applications run local writes and reads directly on the persistent store using a user-level library that handles additional columns needed by Feather.

returns a complete result set for the data as of 15 minutes ago. Alternatively, the query result can fulfill the original freshness guarantee $T_f > T_q - L$, with the caveat that it is partial: it does not contain any new information from the sub-tree that cannot be queried.

In both cases, the failure is communicated to the user: the answer includes the sub-tree that was excluded, as well as the estimated number of rows that the query is missing (using the row coverage feature). Given the actual freshness returned and the number of missing rows, users can then intelligently use or discard the query results, depending on the application.

5.2.7 Adding and Removing Nodes

In Feather, modifications to the topology are local operations and only involve a parent and child node. Nodes can join the topology by connecting to their parent node and a parent node can remove a branch at any time.

5.3 Implementation

We implemented a prototype of Feather as a Kotlin standalone application that uses Cassandra 3.11.4 as its persistent storage. In this section we describe the details of our implementation.

5.3.1 Architecture

Feather is comprised of four components on each node, shown in Figure 5.4: *persistent storage* for local data, a *query server* to receive queries from parents and return results, a *push daemon* to push periodic data updates to parents, and a *receive daemon* to receive child updates.

To eliminate overheads, local reads and writes are executed directly on the local data store. Writes are done through a small client-side driver that adds the necessary metadata for the push demon and query servers.

Persistent Storage

Our implementation uses Cassandra[85] as the persistent storage component. Each local replica runs an independent single-datacenter Cassandra cluster, which allows horizontal scaling within a datacenter by simply adding nodes to the local cluster. We configure Cassandra to QUORUM reads and writes. Feather's design is independent of the choice of the underlying datastore, and can be adapted to use other systems.

Push Daemon

The push daemon is responsible for replicating new and updated data upstream towards the cloud. Whenever a row is written or updated on a local database, the row is marked as *dirty*, with an update timestamp. The push daemon runs in the background and periodically pushes dirty data to the receive daemon in the parent [116, 111]. To avoid saturating links or overwhelming the parent, dirty data that is too large is pushed in batches sorted by timestamp from older to more recent. After a row has been successfully pushed on to the parent receive daemon, it will be marked as clean.

Receive Daemon

The receive daemon is a background process running on each replica that is not located on an edge node. It is responsible for receiving data from the children's push daemons on the node and storing data on the persistent storage. It also records the latest update time as received from each child.

Query Server

The query server processes global queries, and is responsible to executing Algorithm 1 using information recorded by the receive daemon.

5.3.2 Writing and Replicating Data

User applications write data directly to the Feather local store at the node they are running at. To support replication and querying, the following columns are added to the client applications' schema, and added to user writes by a client-side driver: (i) a `timestamp` column;(ii) a Boolean `dirty` column to identify rows that have not yet been pushed up; and (iii) a `prev_loc` that determines from which node the row was received from. If the row was produced on the same node, it will be populated with that node's ID.

After data is written to a replica on a node, it is replicated (pushed) to ancestor nodes on the path to the cloud. Feather implements a write log for each row of a table by adding a `timestamp` column as the last element of the table's clustering key. This is a UUID timestamp that records the time the row was inserted. It is used to resolve write conflicts with a last-write-wins policy, and to determine update times (Section 5.2.2). As described in Section 5.2.2, Feather assumes that all replicas have sufficiently synchronized GPS clocks.

Modifications and updates are propagated through the hierarchy by the push daemon on each node. The push daemon periodically selects all rows that have not been pushed to the receive server (starting from the older ones) and sends them to the receive daemon on the parent node through ZeroMQ [73], which writes the data to the parent's persistent storage. Feather marks a row dirty when it is inserted into the local Cassandra instance by a local write or the receive deamon. The row is only marked as *clear* when the parent acknowledges reception and storage of the write.

5.3.3 Implementing Global Queries

As shown in Figure 5.5, Feather queries follow the format of CQL queries, with additional conditions on data freshness or result latency. To make porting applications easier, and since it is built on top of Cassandra, we support almost all features provided by CQL, specifically all

```
SELECT * | expression [, ...]
FROM table
[ WHERE condition  ]
[ GROUP BY expression [, ...]  ]
[ LIMIT count  ]
[ LAXITY time-delta | DEADLINE time-delta]
```

Figure 5.5: The syntax of a query in Feather.

aggregate functions (`*, MAX, MIN, AVG, COUNT, SUM`) and most clauses (`WHERE, GROUP BY, ORDER, LIMIT, DISTINCT`). This is sufficient for many eventually-consistent edge-computing applications, and for the kind of high-volume queries executed on cloud Cassandra installations. We do not support the CQL `IN` clause, as support of this clause is severely limited even in a centralized Cassandra installation.

Our Feather prototype implements global read queries using Algorithm 1 as described in Section 5.2.3. To support querying rows received from specific children (line 7 in Algorithm 1), cloud and core nodes use a materialized view that includes conditions on the freshness and from which children the query came from (edge nodes do not have this materialized view since they do not have children to query.) The materialized view allows us to use an efficient *IN* predicate, which Cassandra only supports on columns that are part of the primary key. When a query is received on Feather with a requirement on freshness, the query is executed on the materialized view rather than the original table. Consider an example query initiated on node A from Figure 5.3:

```
SELECT * FROM table WHERE key = value

LAXITY = L
```

Since F's data is already on C, it can be fetched from C's local store without querying the child F. Thus, the query executed locally on C will be:

```
SELECT * FROM table WHERE key = value

  AND timestamp > NOW() - L

  AND prev_loc IN ('F','C')
```

where `NOW() - L` implements the freshness requirement.

Finally, To support GROUP BY global queries, we execute them without the GROUP BY

condition and perform the GROUP BY operation in memory.

5.3.4 Merging Results

Algorithm 1 requires incrementally updating results sets (lines 8 and 11). For queries that do not aggregate rows, we simply add the rows into the result set. For aggregate queries and GROUP BY queries, we update the result based on the type of queries, for example by adding values for SUM, updating maximum/minimum, matching groups, and so on. Note our current implementation of aggregate queries assumes rows sets are disjoint: the same row (or key) is only created and updated by the same edge node. While this is sufficient for the scenarios we are targeting, we discuss this limitation in Section 5.3.5.

To perform aggregation queries such as `MIN, MAX, SUM, COUNT` on a node, only a single value is retrieved as a result from child nodes, for queries involving `AVG`, two values are required to perform the aggregation – the average and the number of elements in the set. If there is a `GROUP BY` clause, we compute the aggregation functions for each group, and send the results to the parent node, which merges results from each group with those from other children. Similarly, for a `WHERE` clause, the clause is applied locally on the data and then the result is sent to parent node for aggregation. However for the `DISTINCT, ORDER, LIMIT` clause, our current implementation aggregates result at the final layer of aggregation rather than at intermediate nodes. While there is rich literature on more efficient aggregation [97, 81, 77], this is not the focus of this work.

5.3.5 Prototype Limitations

Our current Feather implementation has certain limitations.

First, our implementation of aggregates (e.g., `COUNT`, `SUM`) currently assumes the set of rows (or keys) written to by different nodes are disjoint, which is the common case in our targeted applications. We plan to address this using data summary techniques such as Cuckoo filters [54] and Count-Min sketches [45] to detect conflicts.

Second, while Feather supports deletion by the application, unlike some other systems [116] we do not clear (i.e., evict) "live" data from intermediate nodes to reclaim space. For our target applications, very old data is seldom relevant for the kind of ad-hoc queries we are targeting.

Such data is often migrated from the cloud replica to a separate batch processing system or cold-storage system in the cloud for later analysis and then deleted, or simply deleted by edge nodes. Supporting such eviction is possible by only evicting data after it already been pushed up, and by modifying the query server to also include local results.

Finally, queries can only read data written locally or propagated from descendants. Again, this is by far the common case for the kinds of scenarios we target, where nodes make local decisions based on local or downstream data. For the rare cases where a query needs data from the whole network, we can offload it to the cloud and execute it there [137, 132]. Another option is downstream replication. While we currently do not replicate data updates down the hierarchy, this is not a fundamental limitation. In practice supporting periodic or on-demand updating from parent replicas to children is a matter of engineering, and has been addressed in prior works [116]. Alternatively, the global querying mechanism can be extended to perform an upwards traversal followed by the usual downward traversal.

5.4 Evaluation

Since applications execute local read or write queries directly on the local Cassandra store of each replica (Section 5.3), we focus instead on evaluating the performance of global queries. We issue global queries in the cloud, as this allows better exploration of the trade-offs.

We evaluate Feather's performance on several metrics:

- Latency, defined as the time between arrival of the user query and availability of results $T_a - T_q$. This time includes all execution times on local and remote persistent stores, as well as communication in the edge network.
- Staleness, defined as the difference between query answer time and the actual freshness time of the results: $T_a - T_f$.
- Bandwidth, which we define as the total number of rows sent over all links in the edge network.
- Work at edges, defined as the average number of rows retrieved by edge nodes from the local Feather replicas to answer a query.

Table 5.1: Toplogies in Controlled Experiments.

Topology	Depth	Split	Nodes per tier	Latency per tier
Wide	3	10	1-10-100	85, 45
Deep	5	3	1-3-9-27-81	70, 30, 20, 10
Medium	4	3	1-3-9-27	80, 85, 15

- Coverage estimation accuracy, our ability to correctly estimate how many data rows were needed to answer the query (Section 5.2.5).

We evaluate Feather's global query mechanism in both controlled and real-world experiments. The bulk of our evaluation is carried out using controlled experiments. The evaluation real-world experiment is detailed in Section 5.4.7.

5.4.1 Experimental Setup for Controlled Experiments

Our controlled experiments are designed to evaluate the benefits and limitations of Feather under controlled settings and on a publicly available dataset. Each experiment uses one of three topologies, summarized in Table 5.1: *wide* uses a network with depth of 3 and split of 10 (one cloud, 10 cores, and 10 edges for each core), *deep* with depth of 5 and split of 3, and *medium* with depth of 4 and split of 3. We run each node on the edge network as a collection of containers on an Amazon instance. The cloud node is an c5.xlarge AWS instance running Ubuntu 18.04 and the the rest of the network is emulated on three m5.16xlarge instances. Each topology has total edge-to-cloud latency of 130ms, divided between the network tiers as explained in Table 5.1 (the end-to-end latency and tier division is similar to real edge networks, such as in Section 5.4.7). The network delays and jitter between the containers is emulated using Linux's Traffic Control [14, 71], and each link has a bandwidth of $1Gbps$.

For end user data, we use the New York Taxi dataset which is a repository of nearly 7 million rides of taxi collected for the month of December 2019 [142], sped up $\times 30$ times to provide more dense data and to allow experiments to run faster. This data set contains geo-distributed labelled data (pickup and drop-off zones), as well as information such as fare amount, number of passengers, and so on. We map each of the 265 geographical zones to the nearest edge, assigning a roughly equal number of zones per edge. When inserting data, each row is

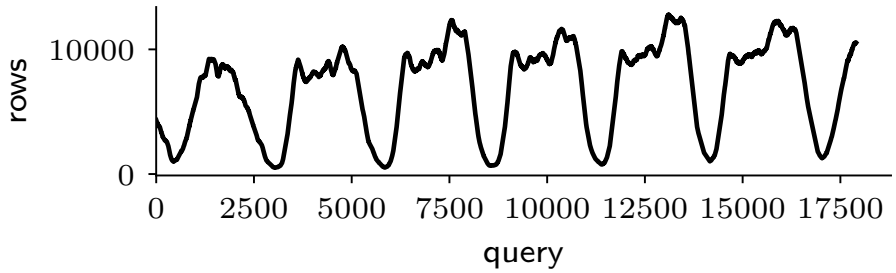


Figure 5.6: Number of rows covered by each query over the length of experiment.

added to the relevant edge by drop-off zone.

We issue 3 queries on the data, all filtered to a window of the last 90 seconds of real time, corresponding to 45 minutes of sped-up time. The SELECT query returns the fare amount and timestamp for all rides in the window for rides with distance larger than 8km. The GROUBPY query groups all rides in the window by passenger count and returns the count of rides and sum of fares, for computing average fare per passenger. The MIN query returns the minimum fare for all rides in the window. These queries were selected to demonstrate the selection, grouping, and aggregation mechanisms of Feather, and because they are representative of the kind of queries that might be run in an application.

In each experiment, we run Feather for about 18000 seconds, which covers about a week of recorded data. Figure 5.6 shows the number of rows covered by the 90 second window in each such query, showing a clear diurnal pattern. Every second, we issue a single query with laxity set between 0 and $(D-1) \cdot f$ where D is the depth of the topology and f is the period of the push demon. The query is selected in round robin order from the 3 possible queries described above. To better measure steady-state behavior, we discard measurements from the first 300 queries in each run. Unless otherwise noted, we set the push daemon interval between two pushes to $f = 30$ and jitter is set to 10% of link latency.

5.4.2 Latency/Staleness Trade-off

Feather is designed to provide controlled trade-off of answer latency and answer staleness in global queries. This trade-off depends on query laxity, network topology, period of the push demon, and data update distribution among the edges. To evaluate this trade-off, we run

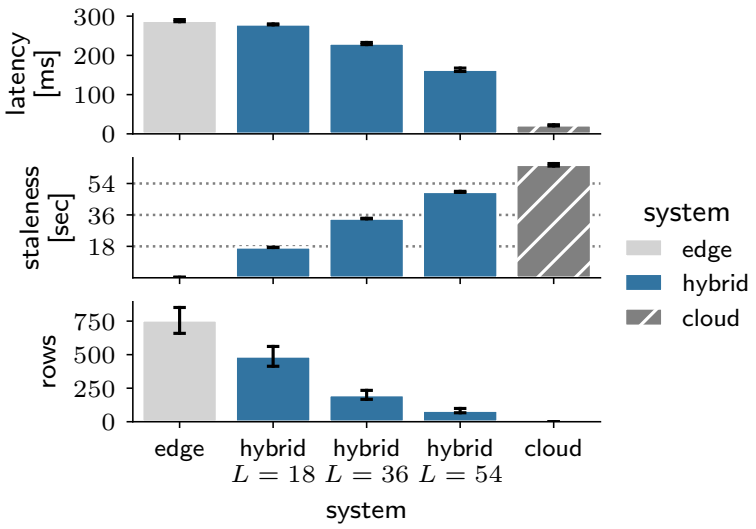


Figure 5.7: Mean query latency, result staleness, and bandwidth (rows sent over the network) when running global queries on the medium topology using different global querying systems.

controlled experiments where we vary the first three, while fixing the data distribution to the NYC Taxi data.

Figure 5.7 shows performance of global queries for the medium topology for several latency levels. Sending all queries to the edge (laxity $L = 0$) results in fresh answers but high latency and bandwidth usage. Running them on the cloud replica results in low latency and zero bandwidth, but stale answers. Feather freshness guarantee ("hybrid") provides flexible trade-off of latency, bandwidth, and staleness, while guaranteeing the freshness threshold L. Error bars show standard deviation. If the latency between nodes is known the optimal (minimum) staleness value can be chosen, as the query answer time depends on the latency between nodes.

Figure 5.8 shows a more complete picture across different topologies and push daemon period f. Each point depicts the answer staleness and latency for that query, and the color indicates the lowest tier involved in answering the query.

The most immediate observation is that query performance is clustered based on the depth of the lowest tier involved in answering them. This is partly because our controlled topologies have similar latency for all nodes in a tier, and the key factor is the round-trip time from cloud to the most distant node (we explore this in Section 5.4.7). We also observe that frequent pushes (top row) result in much fresher answers, at the cost of increased load on the network.

What is the effect of topology on the trade-off? The wider spread of latency for on-cloud

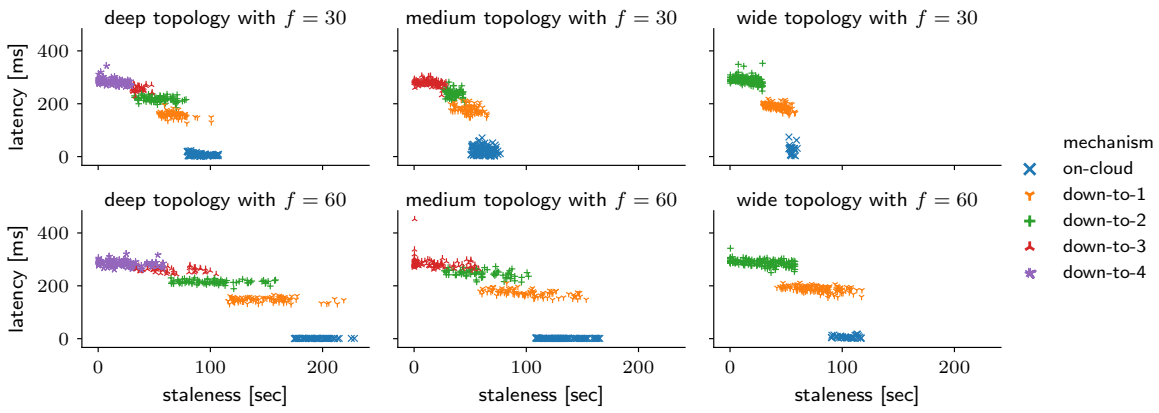


Figure 5.8: Staleness vs latency of the answer for each query. Colors/markers indicate the depth of most distant node which was involved in answering the query. For clarity, we only show a sample of the queries.

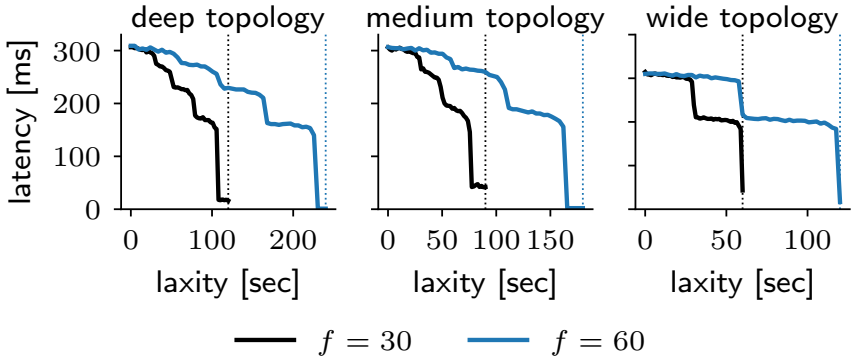


Figure 5.9: 95th percentile of latency as a function of laxity for different with push period $f = 30$ seconds and $f = 60$ seconds. Shaded areas show standard deviation. Dotted lines show the time it takes data to be pushed from edge to the cloud $(D - 1) * f$.

queries in the wide topology indicates increased load at the cloud. Thus, in wide and shallow topology, setting higher push deamon period f might make more sense if we aim to reduce load on the cloud. Finally, deeper network do not inherently result in larger overall latency, it is the round-trip time that counts. Rather, deeper network result in more performance clusters, allowing a more fine-grained trade-off of staleness vs. latency.

Different systems have different requirements: some aim to minimize average latency, while others must meet an SLO such as 95th percentile of latency below a threshold. Feather can help meet these objectives by setting a flexible upper limit of freshness. Figure 5.9 shows how system operators can tune required laxity and push demon period to meet latency requirements. For

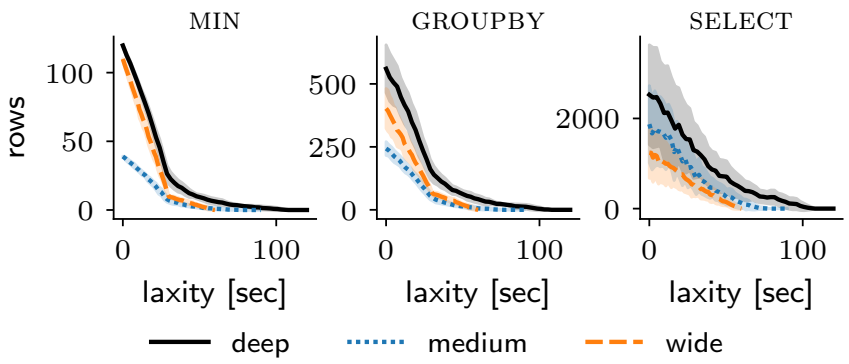


Figure 5.10: Number of rows sent over the network for each types of queries across a range of laxity values. Shaded areas show standard deviation.

example, on a medium topology, to have 95th percentile latency below 230ms with push period $f = 60$, laxity must be set to $L = 111$ seconds. If this is too stale for application requirements, using push period of $f = 30$ seconds with laxity of $L = 50$ will achieve the same thing. As before, deeper topologies offer more fine-grained trade-offs.

A single static laxity setting may not be sufficient as the network conditions and data distribution change. Since Feather provides freshness guarantee per query, it is amenable to dynamically varying the freshness threshold as the workload changes. We plan to explore such dynamic control policies in future work.

5.4.3 Bandwidth and Query Type

We measure the bandwidth used by each query as the number of rows sent over all links in the edge network to answer the query. This depends not only on the data, but also the type of query. Let rows(n) be the number of rows sent by the query sever in node n to the parent to answer a query q. The total number of rows sent over the network is thus $\sum_n \text{rows}(n)$. We define the number of rows returned by each participating node to be 1 for aggregate queries (MIN), and the number of groups in the node's replica for grouping queries (GROUPBY). For nodes not queried, rows(n) is 0.

Figure 5.10 shows the bandwidth reduction for each query type and topology. Feather reduces bandwidth across the board with even a modest laxity, since queries are answered by a smaller set of nodes. Note that for queries that aggregate multiple response from all children

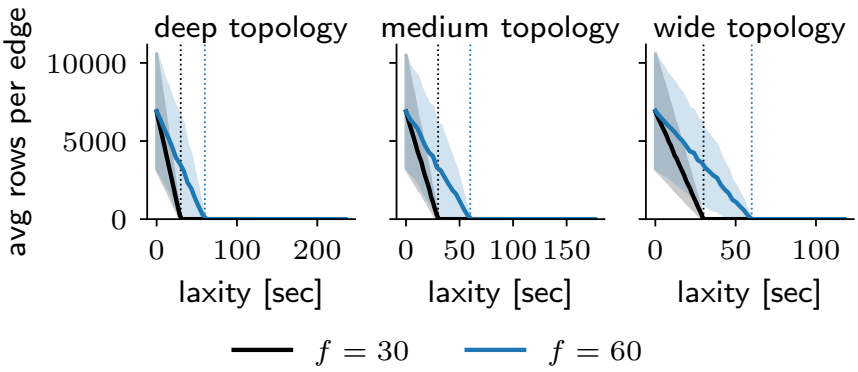


Figure 5.11: Average number of rows accessed by each local edge replica as a function of laxity for different with push period $f = 30$ seconds and $f = 60$ seconds. Feather shifts work from edges towards the core and cloud nodes, and once $L > f$ edges are seldom involved in answering global queries. Shaded areas show standard deviation. Dotted vertical lines show push period f.

to one response (MIN and GROUPBY queries), the number of rows is basically a multiple of the number of links in the network. Since the medium topology has fewer links than wide and deep topologies, we see that those queries requires less bandwidth.

5.4.4 Work at Edge Nodes

Edge nodes are often resource-constraint, and one of goals of Feather is to offload work from the edge nodes towards the inner nodes in the network (core and edge nodes). Figure 5.11 shows, for every laxity level, the average number of rows accessed on the persistent store of local edge replicas. As laxity increases, we observe a linear drop of rows accessed by global queries on edge nodes, leaving more resources to deal with local queries. When laxity grows above the push daemon period ($L > f$), practically all queries can be answered without involving edge nodes since all new data would have been pushed to the core. Note this figure does not show accesses to the persistent store by the push daemon itself. Such an access (once per updated row) would be present in some form in any eventually-consistent database, and we are interested in the extra work induced by ad-hoc queries.

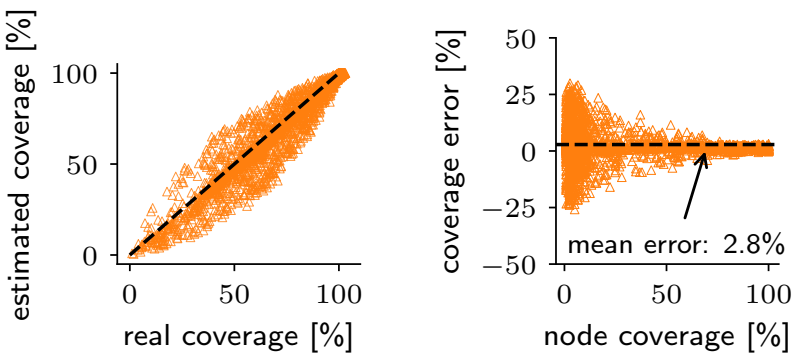


Figure 5.12: Rows covered per query across all topologies and laxity ranges. The left Figure estimated row coverage compared to real row coverage. Dashed line shows equality (Y=X). The right Figure estimation error (the difference between estimate and real row coverage) as a function of how many nodes participated in the query. Dashed line shows the mean coverage estimation error. For clarity, we only show a sample of queries.

5.4.5 Coverage Estimation

As detailed in Section 5.2.5, each global query returns an estimate of the number of rows involved in answering it, and the percentage of rows used to answer the query. When queries execute on the edge ($L = 0$), this number is accurate since we know how many rows were accessed. When $L > 0$, we must estimate the number of updated rows in child nodes not contacted by the algorithm, which we denote as E. Let Q be the true number of rows that would be accessed for each query, and R the number of rows accessed by nodes involved in the query. We define *row coverage* as $\frac{R}{E+R}$, i.e., the estimated fraction of rows needed to answer the query, and the *real coverage* as $\frac{R}{Q}$. We also define *node coverage* as the fraction of nodes that participated in answering the query.

Each point in the left Figure in 5.12 shows the real and estimated coverage for one query from the deep, medium, and wide experiments with $f = 30$. Despite the simplicity of the estimator for E, we see strong agreement between the real and the estimated row coverage. The right Figure in 5.12 shows the *coverage error*, defined as the difference between the real and estimated coverage. The mean coverage error is only 2.7%, confirming the accuracy of the estimator. Unsurprisingly, when more nodes are involved in answering a query, the estimate is more accurate. However, even when very few nodes participate in answering a query, coverage error is below 25%.

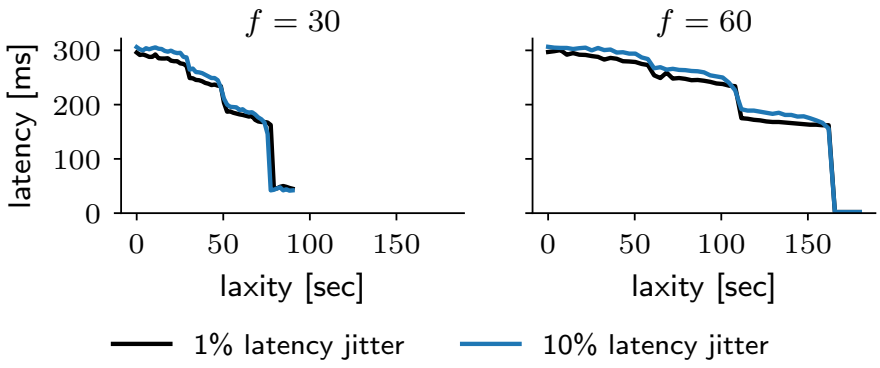


Figure 5.13: Effect of latency jitter on the 95th percentile latency with the medium topology.

5.4.6 Network Jitter

To evaluate the effect of network jitter on latency, we repeat the medium topology experiment with latency jitter set to 1% and to 10%. Figure 5.13 shows the 95th percentile of latency for different laxity levels. Since query servers must wait for the slowest child before replying to the parent, latency jitter slightly increases latency of queries in higher percentiles, but has little effect on mean latency.

5.4.7 Real World Experiment

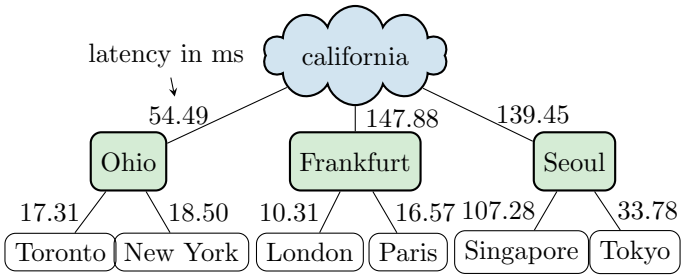


Figure 5.14: Topology of the real-world experiment. Numbers indicate mean measured round trip time in milliseconds.

In this experiment we deploy Feather over a real edge network, comprised of 10 datacenters from three different cloud operators spread over three continents. Figure 5.14 shows the topology of the edge network, and the mean round-trip latency between every two datacenters.

We use geo-tagged public tweets as the dataset for this experiment to simulate the pattern of event arrivals. In this emulation the creation of a tweet is a local event. We scraped a total

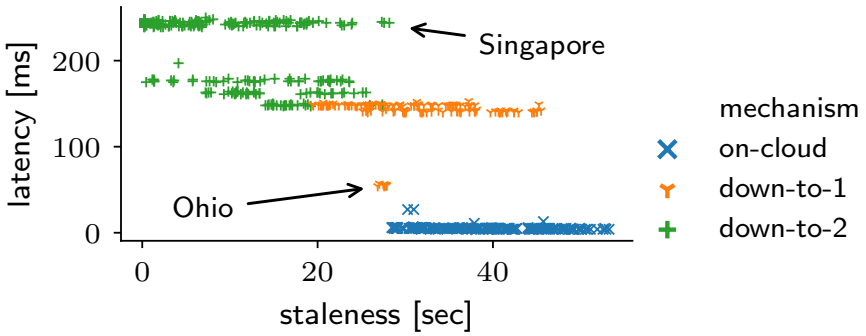


Figure 5.15: Staleness vs latency of the answer for each query in the real-world experiment. For clarity, we only show a sample of the queries. The varied latency from cloud to the different nodes is reflected in the different latency-staleness clusters in the figure.

of 1 million tweets from New York City, Toronto, London, Paris, Singapore, and Tokyo over a one week period from December 2019 using Twint [121]. We up experiment time by $\times 7$. As with the previous experiment, we run over 33000 queries at a rate of 1 query per second, and set the push daemon period to $f = 30$ seconds. The query we run is a MAX query on the number of likes tweets have received in the past 10 minutes. For this experiment we do not add any artificial network delay or jitter.

Figure 5.15 shows the latency/staleness trade-off for queries in the twitter experiment. While the overall shape of the curve remains similar to those seen in Figure 5.8, we observe many more clusters. Though the depth of the lower tier still determines query performance, we observe that the key factor is the round-trip time from the cloud to the most distant node that participated in answering the query. Since the link latency in this experiment is much more varied, we can observe more clusters and even associate some of them with specific nodes.

Figure 5.16 shows the mean latency for each laxity level, which can be used to determine laxity and push period to maintain SLOs. It also shows the average work per edge for this experiment drops linearly with increased laxity, as with previous experiments, reaching zero when $L = f$.

Finally, Figure 5.17 confirms that coverage estimation remains very accurate in the real-world, with a mean error of 0.4%.

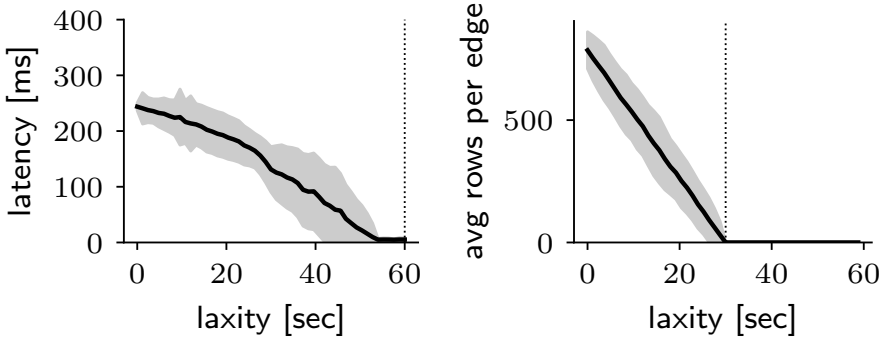


Figure 5.16: Mean latency (left) and average work per edge (right) as a function of laxity. Shaded areas show standard deviation. Dotted lines on the right show push period f and on the left the time it takes data to be pushed from edge to the cloud $(D-1)*f$.

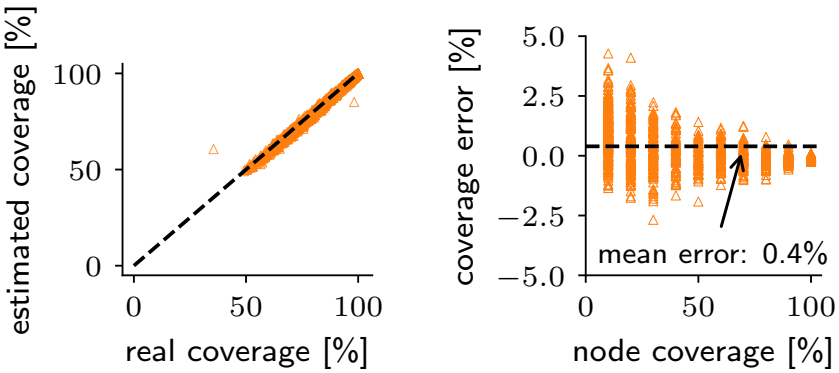


Figure 5.17: Accuracy of coverage estimation for all queries in the real-world experiment. For clarifty we only show a sample of the data.

5.5 Chapter Summary

We proposed Feather: a geo-distributed, eventually-consistent tabular data store for edge computing applications that supports efficient global queries using a flexible freshness guarantee. While most existing work execute global queries on one replica or push to edges, Feather executes global queries on a subset of the network required to meet the user-provided freshness guarantees. Our evaluation of Feather on real and controlled settings shows that it is able to a user-controlled trade-off between latency, staleness, bandwidth, and load on edge nodes.

Chapter 6

Conclusion and Future Work

Edge computing promises highly responsive service by moving the data processing and management resources closer to the end-users and devices. This decentralization of data and process calls for new architectures for running applications as well as storing and managing their data. In this thesis, we have identified the intrinsic properties of a suitable deployment, execution, and storage layer for the edge and presented CloudPath, PathStore, SessionStore, and Feather. These systems take the first steps to make the edge computing vision a reality by providing a new model for structuring and deploying applications and managing their data. In particular, our contributions can be summarized as follows:

- Presenting Path Computing and deploying applications on a path from the edge towards the cloud based on the principle of separating code and data. We implement this architecture with CloudPath that consists of an execution environment that enables the dynamic installation of light-weight stateless event handlers.

- Managing application data in a scalable manner by creating an eventually consistent database that supports storage on a progression of datacenters deployed over the geographic span of a network. PathStore enables transparent data access across the hierarchy of cloud and edge datacenters.

- Providing data consistency for mobile users and applications by presenting a data storage layer that ensures session consistency on a top of otherwise eventually consistent replicas. SessionStore groups related data access into a session and using a session-aware reconciliation algorithm to reconcile only the data that is relevant to the session.

- Enabling distributed querying across a hierarchical edge computing platform by designing and implementing a distributed query engine that exploits the hierarchical structure of eventually-consistent geo-distributed databases to trade temporal accuracy (freshness) for improved latency and reduced bandwidth. Rather than pushing queries to the edge or executing them in the cloud, Feather pushes queries selectively towards the edge while guaranteeing a user-supplied per-query freshness limit.

Below we summarize some potential directions for future research:

CloudPath addresses the problem of application provisioning, deployment, and code execution on the edge. Caching application code on nearby datacenters will improve CloudPath's performance in terms of deployment time. The deployment time can also be reduced by using pools of warm containers and rather than spinning a new container for each application when requests arrive. Additionally, scheduling algorithms could decide where to run functions based on the current load of the system. Moreover, while our current implementation relies on the application developer to provide the system with hints on where to deploy functions, the system should ultimately make this decision automatically. Embedding application functions on the physical plane is an open research question. Finally, while we have implemented a series of applications in section 3.5, and in other studies [50] for CloudPath, it would be interesting to see how current applications can be modified to fit the CloudPath architecture.

In PathStore reads and writes are executed locally, and data transfer between datacenters is managed automatically, but better caching polices could decide on when to evict data based on load and usage. Improved caching policies can significantly improve the performance of PathStore. Moreover, The current PathStore implementation is based on Cassandra; for future work, it would be interesting to apply the concepts used in PathStore on other underlying databases in addition to Cassandra. Another essential improvement for PathStore would be automatically choosing the push period based on network bandwidth and other resources available on datacenters. Currently, this value is static, but future work can have a dynamic approach in setting this parameter.

SessionStore addresses the problem of data consistency and provides session consistency on top of an eventually consistent database. Ensuring read your own writes and monotonic reads/writes is particularly beneficial for mobile applications where devices move between base stations. SessionStore can be extended to provide a range of consistencies from eventual to session to strong consistency. This is a challenging problem, especially given the scale and geo-distribution of edge datacenters. Further, a combination of pro-active replication techniques in conjunction with SessionStore's session aware transfer algorithms to further mitigate the latency of reconciliation can be explored.

Feather addresses the problem of data processing and querying in a geo-distributed database and supports efficient global queries using flexible freshness guarantees. For future work, Feather

can be improved in several ways. First, to address limitations, Feather should allow non-disjoint keys to support more applications. Second, Feather can be improved by placing dynamic control policies for the latency/staleness trade-off. By tuning the laxity parameter dynamically, we can better adapt to changes in data distribution and query patterns. Finally, parameter adjustment, such as finding the minimum staleness based on the network latency between nodes, can improve the guarantees that are given to users.

Edge computing incorporates different technologies from various research areas such as distributed systems, wireless networks and virtualization hence there are wide-ranging challenges for research and innovation in both academia and industry. In addition to potential extensions mentioned above, we believe our systems can improve in the following ways:

- Improved replication strategies: As storage and process get cheaper, it may be beneficial to proactively replicate and cache data and processes on regional datacenters rather than centralized cloud datacenters with the pattern of data consumption determining the caching policy.

- Privacy: One of the main benefits edge computing provides is that storage and processing can be performed locally. For example, data gathered by sensors inside a house can be processed locally, and only some analytical data is sent to the cloud [131]. Many applications, including healthcare applications, can benefit from a processing and storage layer that supports privacy and protects end-user data.

- Resource allocation: Datacenters on the edge have limited processing, storage, and bandwidth resources, and they can be overwhelmed with requests. In a distributed environment, global resource allocation strategies that properly manage the processing, storage, and network resources are required to ensure performance and reliability guarantees.

- Service level agreements (SLA's) have not yet been defined for edge computing, a potential direction for research is to identify new and compatible SLA's for databases and process on the edge that guarantee throughput and data availability and performance.

- Billing: With multiple service providers involved in the underlying network, an interesting question is how to define billing mechanisms for edge computing? Different parameters

can be considered, for example: how much resources are used, what resources are used, what service guarantees are offered, etc.

- Fail-over capabilities: Networks on the edge of the network are more prone to error, and maintaining the hardware on thousands of edge datacenters can be challenging. However, the data storage layer should continue to provide uninterrupted data delivery services when failures happen. An exciting avenue for research is how to balance between providing reliability and performance at the same time.

Edge computing is evolving from merely providing some computational resources to nearby devices to an essential infrastructure with services that significantly amplify the capabilities of applications and devices. While many studies have offered ideas with limited real-world implementations, this thesis offered solutions to bridge this gap by implementing systems that fit the current real-world networks and solve some of the existing edge computing challenges. When this gap is fully bridged, edge computing can emerge as a disruptive technology benefits organizations, businesses, and users for decades to come.

Bibliography

[1] Cisco global cloud index: Forecast and methodology, 2016–2021 white paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[2] Cockroachdb: Ultra-resilient sql for global business. https://www.cockroachlabs.com.

[3] AT&T database of faces. http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html, 2008.

[4] Cassandra hardware choices, December 2017.

[5] Huawei service anchor. http://carrier.huawei.com/en/products/wireless-network/small-cell/service-anchor, 2017.

[6] Installing datastax enterprise on raspberry pi 2 with ubuntu core os, December 2017.

[7] Configure sticky sessions for your classic load balancer. https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-sticky-sessions.htm, 2018.

[8] Consul by hashicorp. https://www.consul.io/, 2020.

[9] Nomad by hashicorp. https://www.nomadproject.io/, 2020.

[10] The Eclipse foundation. http://www.eclipse.org/jetty/, April 2020.

[11] Daniel J Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, (2):37–42, 2012.

[12] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. *ACM SIGOPS Operating Systems Review*, 33(5):186–201, 1999.

[13] A. Ailijiang, A. Charapko, M. Demirbas, B. O. Turkkan, and T. Kosar. Efficient distributed coordination at wan-scale. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1575–1585, June 2017.

[14] Werner Almesberger. Linux traffic control-implementation overview. Technical report, 1998.

[15] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *5th Workshop on Workload Characterization*, number LABOS-CONF-2005-016, 2002.

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[17] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[18] Victor Bahl. Emergence of micro datacenter (cloudlets/edges) for mobile computing. *Microsoft Devices & Networking Summit 2015*, 2015.

[19] Victor Bahl. Cloud 2020: The emergence of micro datacenters for mobile computing. *Online: https://www. microsoft. com/en-us/research/wp-content/uploads/2016/11/Micro-Data-Centers-mDCs-for-Mobile-Computing-1. pdf. Accessed*, 12, 2017.

[20] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[21] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 287–288. ACM, 2016.

[22] Rabindra K Barik, Harishchandra Dubey, and Kunal Mankodiya. Soa-fog: secure service-oriented edge computing architecture for smart health big data analytics. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 477–481. IEEE, 2017.

[23] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. Citeseer.

[24] David Bermbach, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 114–123. IEEE, 2013.

[25] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing*, Washington, DC, October 2016.

[26] Giuseppe Bianchi, Erez Biton, Nicola Blefari-Melazzi, Isabel Borges, Luca Chiaraviglio, Pedro de la Cruz Ramos, Philip Eardley, Francisco Fontes, Michael J McGrath, Lionel Natarianni, et al. Superfluidity: a flexible functional architecture for 5g networks. *Transactions on Emerging Telecommunications Technologies*, 27(9):1178–1186, 2016.

[27] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2):26–35, 2017.

[28] MKABV Bittorf, Taras Bobrovytsky, CCACJ Erickson, Martin Grund Daniel Hecht, MJIJL Kuff, Dileep Kumar Alex Leblang, NLIPH Robinson, David Rorke Silvius Rus, JRDTS Wanderman, and Milne Michael Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th biennial conference on innovative data systems research*, 2015.

[29] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[30] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[31] Maxim Buevich, Anne Wright, Randy Sargent, and Anthony Rowe. Respawn: A distributed multi-resolution time-series datastore. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 288–297. IEEE, 2013.

[32] Kenneth L Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, 1998.

[33] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[35] W. Chang, J. Su, L. Chen, M. Chen, C. Hsu, C. Yang, C. Sie, and C. Chuang. An ai edge computing based w earable assistive device for visually impaired people zebra-crossing

walking. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–2, 2020.

[36] Djabir Abdeldjalil Chekired, Lyes Khoukhi, and Hussein T Mouftah. Industrial iot data scheduling based on hierarchical fog computing: a key for enabling smart factory. *IEEE Transactions on Industrial Informatics*, 14(10):4590–4602, 2018.

[37] Baotong Chen, Jiafu Wan, Antonio Celesti, Di Li, Haider Abbas, and Qin Zhang. Edge computing in IoT-based manufacturing. *IEEE Communications Magazine*, 56:103–109, 09 2018.

[38] Min Chen, Yixue Hao, Kai Lin, Zhiyong Yuan, and Long Hu. Label-less learning for traffic control in an edge network. *IEEE Network*, 32(6):8–14, 2018.

[39] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. Appscale: Scalable and open appengine application development and deployment. In *International Conference on Cloud Computing*, pages 57–70. Springer, 2009.

[40] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[41] Cisco. Cisco annual internet report (2018–2023) white paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html, 2020.

[42] Bastien Confais, Adrien Lebre, and Benoît Parrein. Performance analysis of object store systems in a fog and edge computing infrastructure. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXIII*, pages 40–79. Springer, 2017.

[43] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[44] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[45] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[46] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[47] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[48] Marcos Dias de Assunçào, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1 – 17, 2018.

[49] Eyal de Lara, Carolina S. Gomes, Steve Langridge, S. Hossein Mortazavi, and Meysam Roodi. Poster: Hierarchical serverless computing for the mobile edge. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing*, Washington, DC, October 2016.

[50] Joel Dick, Caleb Phillips, Seyed Hosein Mortazavi, and Eyal de Lara. High speed object tracking using edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–2, 2017.

[51] Shlomi Dolev, Patricia Florissi, Ehud Gudes, Shantanu Sharma, and Ido Singer. A survey on geographically distributed big-data processing using MapReduce. *IEEE Transactions on Big Data*, 5:60–80, 07 2017.

[52] Boubacar Doucoure, Kodjo Agbossou, and Alben Cardenas. Time series prediction using artificial wavelet neural network and multi-resolution analysis: Application to wind speed data. *Renewable Energy*, 92:202–211, 2016.

[53] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[54] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, 2014.

[55] Fang Gao, Zhangqin Huang, Shulong Wang, and Zheng Wang. A hybrid clock synchronization architecture for many-core cluster system based on GPS and IEEE 1588. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 2645–2649, 2016.

[56] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[57] Nam Ky Giang, Victor CM Leung, and Rodger Lea. On developing smart transportation applications in fog computing paradigm. In *Proceedings of the 6th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications*, pages 91–98, 2016.

[58] Will Glozer. wrk - a modern http benchmarking tool. https://github.com/wg/wrk, 2017.

[59] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, page 197–206, 2011.

[60] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: code offload by migrating execution transparently. In *Presented as part of the*

10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pages 93–106, 2012.

[61] Abhimanyu Gosain, Mark Berman, Marshall Brinn, Thomas Mitchell, Chuan Li, Yuehua Wang, Hai Jin, Jing Hua, and Hongwei Zhang. Enabling campus edge computing using geni racks and mobile resources. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing*, Washington, DC, October 2016.

[62] Ramesh Govindan, Joseph Hellerstein, Wei Hong, Samuel Madden, Michael Franklin, and Scott Shenker. The sensor network as a database. Technical report, Citeseer, 2002.

[63] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, et al. Mesa: a geo-replicated online data warehouse for google's advertising system. *Communications of the ACM*, 59(7):117–125, 2016.

[64] Harshit Gupta, Zhuangdi Xu, and Umakishore Ramachandran. Datafog: Towards a holistic data management platform for the iot age at the network edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[65] Harshit Gupta, Zhuangdi Xu, and Umakishore Ramachandran. Datafog: Towards a holistic data management platform for the iot age at the network edge. In *HotEdge*, 2018.

[66] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[67] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 153–166. ACM, 2013.

[68] Kiryong Ha and Mahadev Satyanarayanan. Openstack++ for cloudlet deployment. 2015.

[69] Benjamin Heintz, Abhishek Chandra, and Ramesh K Sitaraman. Trading timeliness and accuracy in geo-distributed streaming analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 361–373. ACM, 2016.

[70] Benjamin Heintz, Abhishek Chandra, and Ramesh K Sitaraman. Optimizing timeliness and cost in geo-distributed streaming analytics. *IEEE Transactions on Cloud Computing*, 2017.

[71] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, pages 18–23, 2005.

[72] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[73] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[74] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology*, 65(6):3860–3873, 2016.

[75] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 5. ACM, 2016.

[76] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[77] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), October 2008.

[78] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67. ACM, 2000.

[79] Itseez. Open source computer vision library. `https://github.com/itseez/opencv`, 2015.

[80] Minsung Jang, Hyunjong Lee, Karsten Schwan, and Ketan Bhardwaj. Soul: An edge-cloud system for mobile applications in a sensor-rich world. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 155–167. IEEE, 2016.

[81] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, page 41–52, 2010.

[82] Leonard Kawell Jr, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, page 395. ACM, 1988.

[83] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[84] SPT Krishnan and Jose L Ugia Gonzalez. Google cloud dataflow. In *Building Your Next Big Thing with Google Cloud Platform*, pages 255–275. Springer, 2015.

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

[86] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[87] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[88] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular composition of coordination services. In *USENIX Annual Technical Conference (ATC)*, 2016.

[89] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 265–278, 2012.

[90] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 63–75. ACM, 2003.

[91] Yujin Li and Wenye Wang. Can mobile cloudlets support mobile applications? In *Infocom, 2014 proceedings ieee*, pages 1060–1068. IEEE, 2014.

[92] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Enhancing edge computing with database replication. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 45–54. IEEE, 2007.

[93] Chenghao Liu, Steven CH Hoi, Peilin Zhao, and Jianling Sun. Online arima algorithms for time series prediction. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[94] Peng Liu, Bozhao Qi, and Suman Banerjee. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, pages 1–6, 2018.

[95] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Proceedings of the 1st IEEE/ACM Symposium on Edge Computing*, Washington, DC, October 2016.

[96] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 1–13. IEEE, 2016.

[97] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 101–114. Association for Computing Machinery, 2016.

[98] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[99] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[100] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual consistency. *Commun. ACM*, 57(5):61–68, May 2014.

[101] Ping Lou, Liang Shi, Xiaomei Zhang, Zheng Xiao, and Junwei Yan. A data-driven adaptive sampling method based on edge computing. *Sensors*, 20(8):2174, 2020.

[102] Andrew Machen, Shiqiang Wang, Kin K Leung, Bong Jun Ko, and Theodoros Salonidis. Live service migration in mobile edge clouds. *IEEE Wireless Communications*, 25(1):140–147, 2018.

[103] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[104] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[105] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al.

Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015.

[106] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. Fogstore: Toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6. IEEE, 2017.

[107] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[108] Microsoft. Consistency levels in azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2020.

[109] Nader Mohamed, Jameela Al-Jaroodi, Sanja Lazarova-Molnar, Imad Jawhar, and Sara Mahmoud. A service-oriented middleware for cloud of things and fog computing supporting smart city applications. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–7. IEEE, 2017.

[110] Rosario Morello, Claudio De Capua, Gaetano Fulco, and Subhas Chandra Mukhopadhyay. A smart power meter to monitor energy flow in smart grids: the role of advanced sensing and IoT in the electric grid of the future. *IEEE Sensors Journal*, 17(23):7828–7837, 12 2017.

[111] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. Toward session consistency for the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[112] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. Toward session consistency for the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[113] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Puzhavakath Narayanan Shankaranarayanan. Poster: Pathstore, a data storage

layer for the edge. Accepted at the The 16th ACM International Conference on Mobile Systems, Applications, and Services (Mobisys), 2018.

[114] Seyed Hossein Mortazavi, Mohammad Salehe, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan PuzhavakathNarayanan. Sessionstore: A session-aware datastore for the edge. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 59–68. IEEE, 2020.

[115] Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel, and Eyal de Lara. Feather: Hierarchical querying for the edge. In *Proceedings of the Fifth ACM/IEEE Symposium on Edge Computing (SEC)*. IEEE, 2020.

[116] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[117] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. Frontier: Resilient edge processing for the internet of things. *Proceedings of the VLDB Endowment*, 11(10):1178–1191, 2018.

[118] Opeyemi Osanaiye, Shuo Chen, Zheng Yan, Rongxing Lu, Kim-Kwang Raymond Choo, and Mqhele Dlodlo. From cloud to fog computing: A review and a conceptual live vm migration framework. *IEEE Access*, 5:8284–8300, 2017.

[119] Mugen Peng, Yong Li, Zhongyuan Zhao, and Chonggang Wang. System architecture and key technologies for 5g heterogeneous cloud radio access networks. *IEEE network*, 29(2):6–14, 2015.

[120] Jan Plachy, Zdenek Becvar, and Emilio Calvanese Strinati. Dynamic resource allocation exploiting mobility prediction in mobile edge computing. In *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6. IEEE, 2016.

[121] Francesco Poldi. Twint-twitter intelligence tool. *URL: https://github.com/twintproject/twint (visited on 01/02/2020)*, 2020.

[122] Ravi Prakash and Mukesh Singhal. Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems. *Wireless Networks*, 3(5):349–360, 1997.

[123] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *arXiv preprint arXiv:1011.5808*, 2010.

[124] Ioannis Psaras, Onur Ascigil, Sergi Rene, George Pavlou, Alex Afanasyev, and Lixia Zhang. Mobile data repositories at the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[125] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 11(4), January 2017.

[126] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[127] Ju Ren, Hui Guo, Chugui Xu, and Yaoxue Zhang. Serving at the edge: A scalable iot architecture based on transparent computing. *IEEE Network*, 31(5):96–105, 2017.

[128] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40. IEEE, 2017.

[129] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178. IEEE, 2016.

[130] Yasir Saleem, Noel Crespi, Mubashir Husain Rehmani, and Rebecca Copeland. Internet of things-aided smart grid: technologies, architectures, applications, prototypes, and future research directions. *IEEE Access*, 7:62962–63003, 2019.

[131] Mohammad Salehe, Zhiming Hu, Seyed Hossein Mortazavi, Iqbal Mohomed, and Tim Capes. Videopipe: Building video stream processing pipelines at the edge. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 43–49, 2019.

[132] Farzad Samie, Vasileios Tsoutsouras, Lars Bauer, Sotirios Xydis, Dimitrios Soudris, and Jörg Henkel. Computation offloading and resource allocation for low-power iot edge devices. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 7–12. IEEE, 2016.

[133] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[134] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.

[135] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019.

[136] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, Oakland, CA, 2015. USENIX Association.

[137] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[138] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[139] Baihaqi Siregar, Ahmad Badril Azmi Nasution, and Fahmi Fahmi. Integrated pollution monitoring system for smart city. In *2016 International Conference on ICT For Smart Society (ICISS)*, pages 49–52. IEEE, 2016.

[140] Sandeep K Sood and Kiran D Singh. An optical-fog assisted eeg-based virtual reality framework for enhancing e-learning through educational games. *Computer Applications in Engineering Education*, 26(5):1565–1576, 2018.

[141] Xiang Sun and Nirwan Ansari. Edgeiot: Mobile edge computing for the internet of things. *IEEE Communications Magazine*, 54(12):22–29, 2016.

[142] NYC Taxi and Limousine Commission. New york city trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2020.

[143] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[144] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[145] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. Reconfigurable streaming for the mobile edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 153–158, New York, NY, USA, 2019. Association for Computing Machinery.

[146] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. Reconfigurable streaming for the mobile edge. In *Proceedings of the 20th Interna-*

tional Workshop on Mobile Computing Systems and Applications, pages 153–158. ACM, 2019.

[147] Sana Tonekaboni, Mjaye Mazwi, Peter Laussen, Danny Eytan, Robert Greer, Sebastian D Goodfellow, Andrew Goodwin, Michael Brudno, and Anna Goldenberg. Prediction of cardiac arrest from physiological signals in the pediatric icu. In *Machine Learning for Healthcare Conference*, pages 534–550, 2018.

[148] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[149] Animesh Trivedi, Lin Wang, Henri Bal, and Alexandru Iosup. Sharing and caring of data at the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[150] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36. ACM, 2012.

[151] Mario Villamizar, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, Mery Lang, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 179–182. IEEE, 2016.

[152] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[153] Peter Volgyesi, Abhishek Dubey, Timothy Krentz, Istvan Madari, Mary Metelko, and Gabor Karsai. Time synchronization services for low-cost fog computing applications. In *Proceedings of the 28th International Symposium on Rapid System Prototyping: Short-*

ening the Path from Specification to Prototype, RSP '17, page 57–63. Association for Computing Machinery, 2017.

[154] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. Wanalytics: Analytics for a geo-distributed data-intensive world. In *CIDR*, 2015.

[155] Haoyu Wang, Jiaqi Gong, Yan Zhuang, Haiying Shen, and John Lach. Healthedge: Task scheduling for edge computing with health emergency and human behavior consideration in smart homes. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1213–1222. IEEE, 2017.

[156] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173. IEEE, 2018.

[157] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[158] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[159] Yang Yang, Qiang Cao, and Hong Jiang. Edgedb: An efficient time-series database for edge computing. *IEEE Access*, 7:142295–142307, 2019.

[160] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod record*, 31(3):9–18, 2002.

[161] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 15. ACM, 2017.

[162] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms: a complete survey. *Journal of Systems Architecture*, 2019.

[163] Rong Yu, Yan Zhang, Stein Gjessing, Wenlong Xia, and Kun Yang. Toward cloud-based vehicular networks with efficient resource management. *IEEE Network*, 27(5):48–55, 2013.

[164] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[165] John K Zao, Tchin Tze Gan, Chun Kai You, Sergio José Rodríguez Méndez, Cheng En Chung, Yu Te Wang, Tim Mullen, and Tzyy Ping Jung. Augmented brain computer interaction based on fog computing and linked data. In *2014 International Conference on Intelligent Environments*, pages 374–377. IEEE, 2014.

[166] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

[167] Quan Zhang, Qingyang Zhang, Weisong Shi, and Hong Zhong. Firework: Data processing and sharing for hybrid cloud-edge analytics. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2004–2017, 2018.

[168] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In

Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom '15, pages 426–438, 2015.