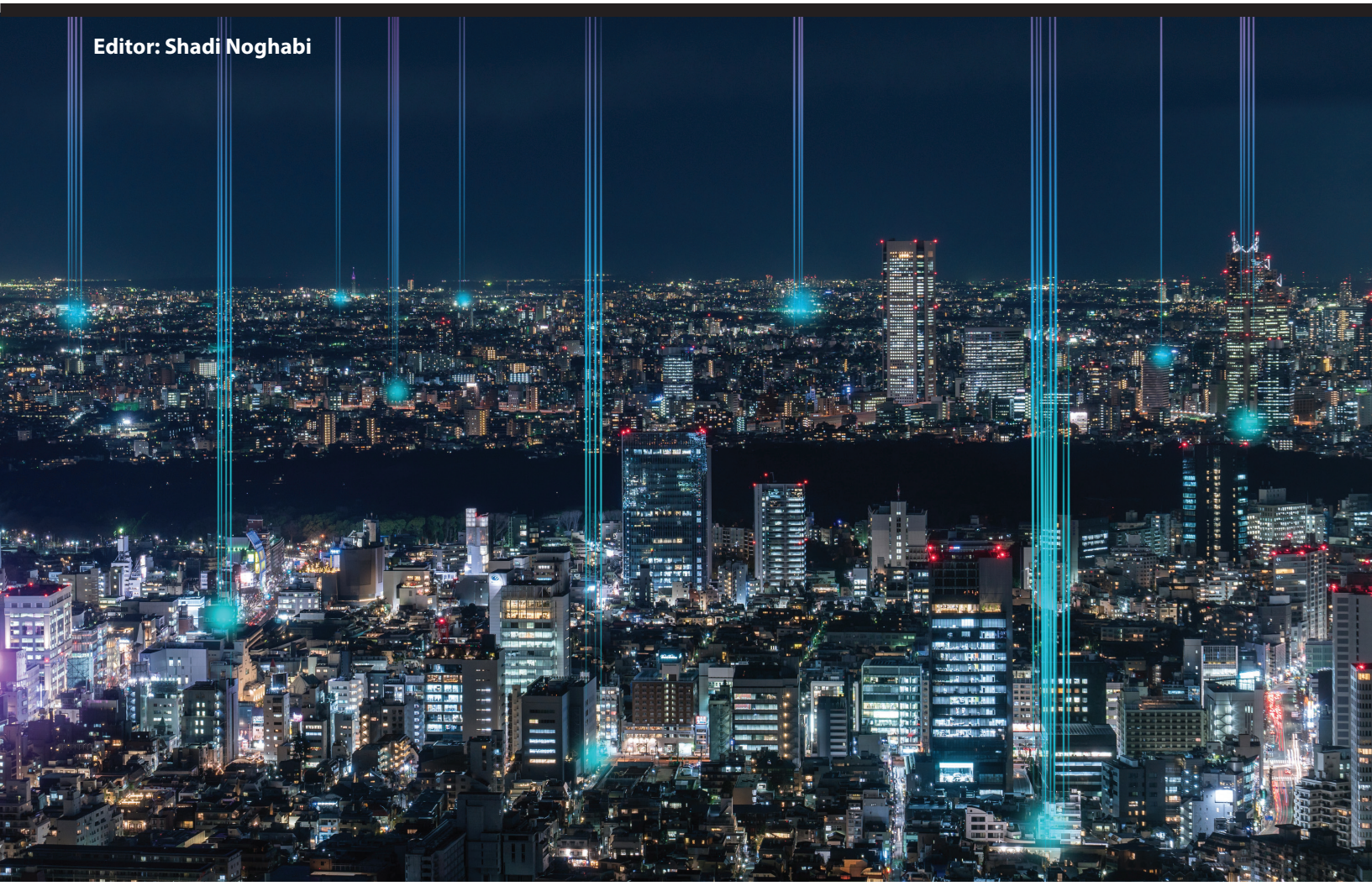
Seyed Hossein Mortazavi *Department of Computer Science, University of Toronto, Toronto, Canada*
Mohammad Salehe *Department of Computer Science, University of Toronto, Toronto, Canada*
Moshe Gabel *Department of Electrical Engineering and Computer Science, York University, Toronto, Canada*
Eyal de Lara *Department of Computer Science, University of Toronto, Toronto, Canada*

Editor: Shadi Noghabi



Photo, istockphoto.com

DATA MANAGEMENT SYSTEMS FOR THE HIERARCHICAL EDGE

In recent years, there has been an exponential increase in the generation of data at the edge of the network. The International Data Corporation (IDC) estimates that the Global Datasphere, which was 33 zettabytes in 2018, will rise to 175 zettabytes by 2025, and there will be more than 150 billion connected devices worldwide [10]. The Internet of Things (IoT) segment is expected to experience the fastest growth, with data creation at the edge of the network projected to increase almost twice as fast as in the cloud. As a result, worldwide spending on edge computing is forecasted to reach $317 billion by 2026, as per IDC projections [1].

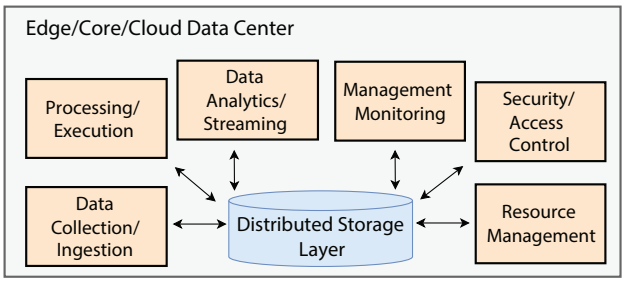
The concept of edge computing involves distributing computation and storage services by positioning resources in proximity to the data sources. The term "edge networks" refers to a hierarchical structure of data centers, edge devices, intermediate nodes, and cloud systems that are organized according to their processing and storage capabilities [14]. This setup facilitates efficient data processing and analysis. By using local data centers, processing of data is faster with decreased

latency and increased bandwidth, leading to improved performance for a range of applications. [2, 11, 12].

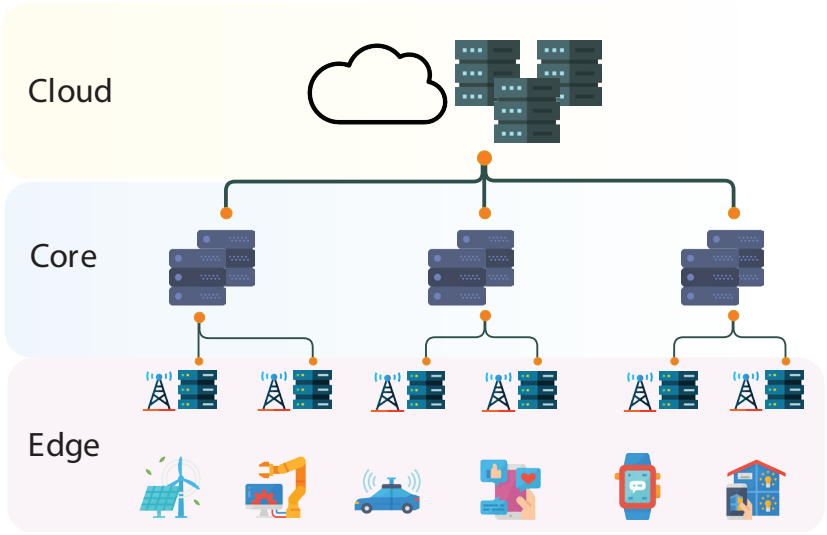
In a multitier edge computing environment, these datacenters house multiple components including processing, streaming, management, security, and more as shown in Figure 1a. These components are interconnected through a unified distributed storage layer, which facilitates the flow of data vertically between different layers and horizontally between different components.

This decentralization poses challenges in effectively managing the data and ensuring reliable service for end-users, which include:

- **Storage:** Edge data centers have lower storage and processing capabilities compared to cloud nodes. Despite this, accessing data locally is crucial to leverage the benefits of edge computing. Deciding which data to store at the edge can be a challenging task.
- **Consistency:** Synchronizing multiple distributed resources is necessary in edge computing, but maintaining consistency among data stored in different locations and ensuring timely updates presents a challenge. This challenge is especially crucial in maintaining data consistency across geographically distributed data centers.
- **Partitioning and Offloading:** The distribution of computing and storage resources across different locations presents a challenge in deciding how to partition and deploy application logic effectively. This challenge involves determining whether to consolidate data on central nodes or partition it based on geographical location, depending on the specific requirements of each application.
- **Reliability and Quality of Service:** Edge computing is preferred over cloud computing only if it provides superior services. However, edge computing faces the challenge of maintaining high throughput and low latency across various environments and scenarios, particularly when data centers are located in different regions with high network latency.
- **Privacy:** Data privacy is crucial in edge computing because decentralization allows sensitive data to be stored and processed locally. Robust security measures are essential to maintain data privacy.



(a) Overview of different components in an edge datacenter.



(b) A multi-tiered Edge Computing architecture

FIGURE 1. Edge computing architecture and its component. A series of data centers arranged between the client device and the cloud data center, which increase in size as they move towards the cloud.

In this paper, we provide an overview of three projects executed over 3 years that have aimed to improve the performance of distributed database management systems for the edge computing by addressing some of the challenges mentioned above.

We first present *PathStore* [3, 9] a data storage layer that provides eventual consistency for a multitier cloud architecture. PathStore enables data storage across a range of data centers, from the edge to the cloud. Additionally, we propose *SessionStore* [5–7], which addresses consistency issues and improves storage capabilities by providing session consistency for mobile users at the edge. Lastly, we present *Feather* [8], a hybrid querying scheme that capitalizes on the hierarchical structure of geo-distributed systems. It allows for a trade-off between temporal accuracy (freshness) and improved latency and decreased bandwidth utilization.

PATHSTORE

Data management is a vital aspect of edge computing platforms, as it handles the transformation, aggregation, and consumption of data. In this section, we introduce PathStore, a shared database abstraction that enables seamless data access across different levels of data centers. PathStore [3, 9] enables data storage across a range of data centers, from the edge to the cloud and offers developers the flexibility to run their server-side operations on various locations, enabling diverse application types. These applications can include data-aggregating workloads, such as those found in IoT applications, as well as services that cache and process data across different layers, and stateless applications.

The root node of the PathStore hierarchy has a persistent database, while other levels act as caches. To simplify implementation,

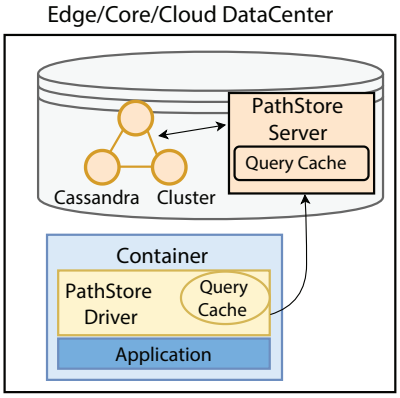


FIGURE 2. PathStore components and its relation with the application. The Cassandra cluster can scale horizontally, based on the location and available resources.

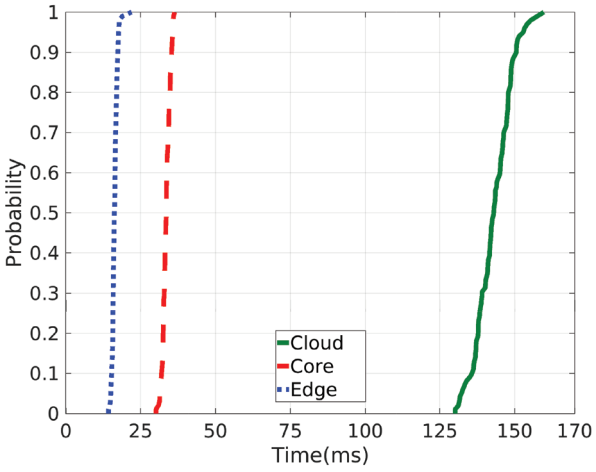


FIGURE 3. CDF for response time of the Face Recognition application when PathStore is used.

data replicated by a node must be a superset of its children. Low latency is ensured by executing read and write operations locally. PathStore supports concurrent reads and writes and updates are eventually propagated through the hierarchy in the background.

Figure 2 illustrates a sample three-layer PathStore deployment. PathStore comprises 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 [4], but the design can be adapted to other storage engines. The PathStore server copies data between its local Cassandra instance and the Cassandra instance of its parent node. The PathStore driver offers an API for edge applications to query the local PathStore node, with the prototype based on CQL (Cassandra's SQL dialect) for data organization into tables and atomic read and write operations at the row level.

PathStore replicates data at the row level 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 performed locally, the 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 maintains 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 *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 minimize 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. PathStore also provides local tables for temporary storage.

The system regularly removes cold query cache entries and locally replicated rows that do not match any query in the query cache to prevent unnecessary data from being fetched. If there is a resource contention issue, the system uses a least recently used (LRU) policy to free up space.

PathStore receives write queries on a node and applies all changes locally. Periodically, a *push daemon* transmits these local updates to higher levels of the hierarchy. To keep track of modifications, PathStore uses a write log. As changes propagate up and down the hierarchy via the push and pull daemons, PathStore uses the version timestamp to establish the order of modifications.

Consistency

At the individual node level, PathStore maintains the storage semantics of its underlying native object store. Our current prototype provides local durability, row-level isolation and atomicity, and strong consistency based on Cassandra's quorum mechanism. However, across nodes, PathStore propagates updates at the 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.

Experimental Results

PathStore allows applications to deliver localized content to edge nodes based on the end-users' geographic location. One example is face recognition classifiers, which are trained on a dataset specific to a particular geographical location and we have developed a face detection and recognition application that leverages OpenCV and JavaCV to demonstrate the performance of Pathstore.

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. Running on the edge lowers the latency by 88 percent.

SESSIONSTORE

PathStore utilizes an eventual consistency model to replicate data where updates are propagated in the background and if no new updates are made to an object, eventually

all replicas will converge to the same value. Eventual consistency is suitable for many applications where clients interact with the same replica for the duration of their sessions. As long as the client interacts with the same replica, the storage system in effect provides session consistency, which is 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 presents each individual client with an intuitive view of the world that is consistent with the client's own actions. Applications that can benefit from session consistency on the edge include authentication services, file storage applications, and messaging applications.

Session consistency is particularly important for mobile users on the edge as it provides a seamless and intuitive experience for the user. Mobile users often rely on edge storage systems for critical applications, such as authentication services, file storage, and messaging.

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 moves between different edges; (ii) functionality is dynamically reallocated between edges; or (iii) an application's functionality has been partitioned between different data centers (e.g., running some functions on the edge and others on the cloud). If consecutive client requests are sent to different replicas before the data needed by the client request is replicated, the application may not be able to read its own writes or have monotonic reads.

To address the issues of consistency and scalability in edge computing environments, we introduce SessionStore. SessionStore [5–7] is a specialized datastore designed for edge computing that aims to improve upon PathStore by providing session consistency across a hierarchy of eventually consistent replicas. It supports session consistency through a session-aware reconciliation algorithm that reconciles only the keys that a client reads or writes at the source replica, eliminating the need for full reconciliation of all replicas. Furthermore, it minimizes data transfer by not transferring up-to-date data already existing on the destination. In our example application use case, this saves as much as 95% in terms of data transfer.

Design

The concept behind our approach to ensuring session consistency is straightforward but efficient: we organize related datastore operations into sessions, and we keep track of all the rows either read or written by a session through monitoring the queries it executes. To minimize the overhead, we aggregate the queries used to keep track of the data. When a client moves 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.

We enforce session consistency by grouping related CQL requests into a *Session*. The definition of what constitutes a session is left up to the application developer to decide. For example, the developer can decide to make a session represent a user, a device belonging to a user, a set of commands executed by a function, or a subset of 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 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 can choose 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 $T_1(key, v1)$ **VALUES** $(a1,b1)$

we store the following query in CommandCache[s]:

SELECT * **FROM** T_1 **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 simply 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).

Switching. We use a token (stoken) to track when a client switches between replicas, such as moving from n_S to n_D. The new replica checks the stoken and begins the reconciliation process if the client has switched. The source replica (n_S) is halted and the new replica (n_D) waits for the reconciliation process to finish before fetching data. The process involves the destination replica (n_D) sending a request to the source replica for all session data, which is retrieved by re-executing the session's queries. The resulting data and queries are then transmitted to the destination replica. Queries are used to track accessed rows, with writes mapped to separate queries and reads being aggregated. We also implemented other optimizations to lower data transfer between (n_S,n_D), by monitoring the data that is already present on the n_D.

If a source replica fails while a destination is replicating data from it, SessionStore waits for the source to become available again to continue transferring the rows that were not already replicated.

The application is notified of any issues via an exception and can choose to wait and retry or restart by invalidating the session.

Experimental Results

We conduct our experiments on an emulated hierarchical edge. Our topology consists of a cloud datacenter (cl), and two mobile networks each with a datacenters at its core (c_1,c_2), and one or two additional datacenters at edge location such as base stations (e_1, e_2, e_3). We assume the latency between the cloud and the core is 20 ms and between the core and the edge is 2 ms.

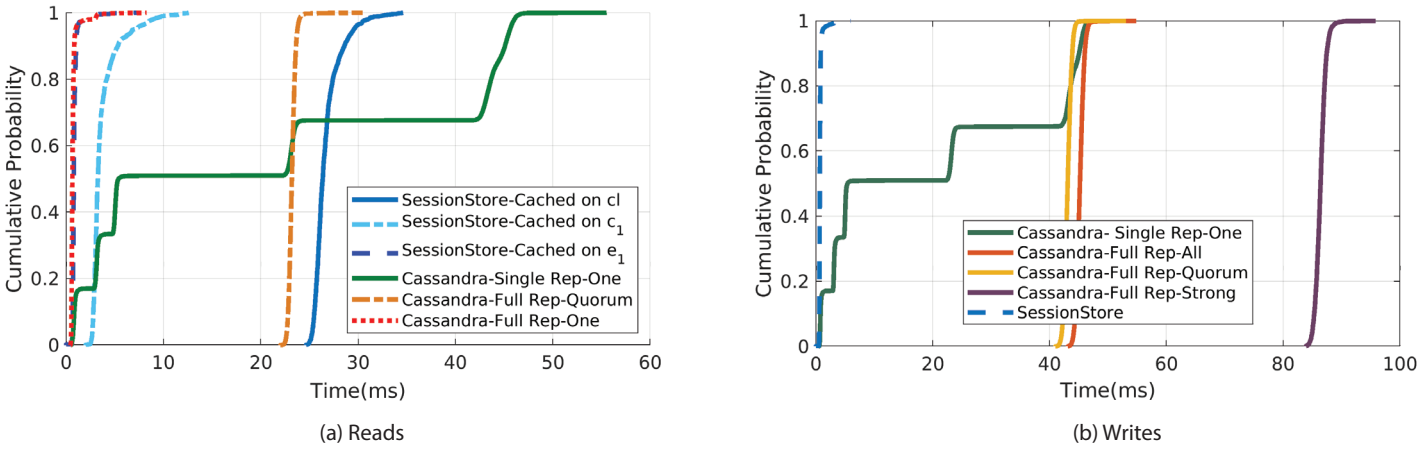


FIGURE 4. CDF of latency required to read and write a 1*KB* row on a cloud node (cl) a core node (c1) and an edge node (e1).

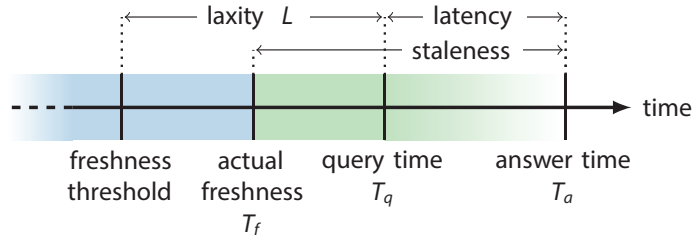


FIGURE 5. The freshness guarantees for Feather global queries. Actual freshness T_f is guaranteed to be between $T_q - L$ and T_a. Any row created before T_f (blue) is guaranteed to be included in the results, while rows created after T_f (green) may or may not be included.

Here we present results that quantify the overhead of keeping track of session information and compare the approach to alternatives that enforce stronger consistency as the cost of higher overhead. To measure the cost of keeping track of session state, we compared the latency for reading and writing single 1*KB* row on e_1 with SessionStore. The experiment is repeated for 10 000 different rows. Figure 4a 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 e_l, respectively. 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 and 26.2 ms when the row had to be fetched from the core and cloud, respectively.

Figure 4b 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.

FEATHER

Managing data in a geographically distributed environment poses challenges due to limitations in network links, such as limited bandwidth and high latency variations. However, many applications have a strong locality where most reads and writes can be done locally and changes do not need to be immediately replicated throughout the network. The traditional approach of storing data locally and replicating it periodically to higher layers can provide fast local reads and writes but can't guarantee freshness and completeness of read queries when executed at the parent layer such as in PathStore. Executing the query on the cloud's local replica can result in stale data, while fetching data from edge devices incurs high latency, added load, and potential data loss if an edge is unreachable. Here, we present a hybrid approach for efficient on-demand global queries with guaranteed freshness by utilizing the hierarchical structure of edge networks.

Feather [8] is a data management system that manages the trade-off between data freshness and query latency in edge computing applications. Users can specify freshness constraints or deadlines for each query and Feather will execute it over a subset of the network using local replicas as caches, returning a result set that meets the freshness requirements. The system can handle intermittent link errors and provides an estimate of missed data. It supports features found in high-performance tabular data stores and can be used to port existing read queries from centralized databases.

Design

Feather is a storage system that offers both local and global queries. Local queries are similar to those of other edge-centric databases and are executed directly on the high-performance local data store. Global queries, however, provide freshness guarantees and are computed from recent local and descendant data up to a limit, avoiding remote queries for faster response and conserving bandwidth. Feather also includes features like query deadlines, result coverage estimation, and graceful link failure handling.

Feather guarantees that the set of rows used to process a query will contain all data updates (insertions, deletions, and updates) that occurred before a user-defined freshness threshold, $T_q - L$, where T_q is the time the query was sent for execution and L is the specified limit on data freshness. The query results also include an actual freshness time, T_f, which represents the time when all data updates included in the answer were made. The exact value of T_f may vary depending on the replication status and transfer time between datacenters. Additionally, the answer

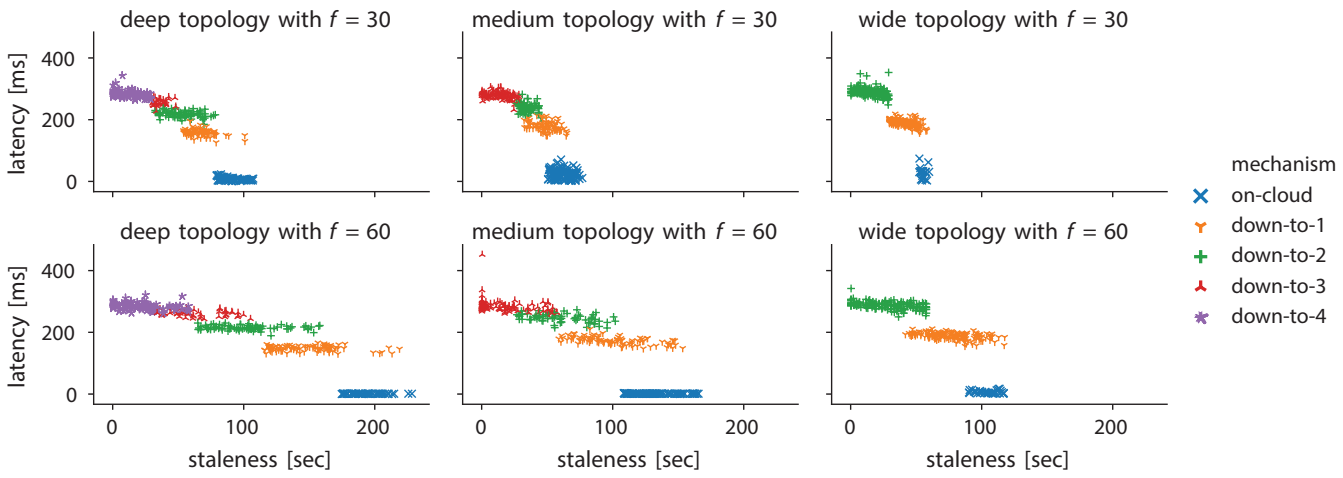


FIGURE 6. Staleness vs latency of the answer for each query. Colors/markers indicate the depth of most distant node, which was involved in answering the query. For clarity, we only show a sample of the queries.

may be slightly out of date due to query execution and data transfer time, and the difference between the answer time, T_a, and actual freshness time is defined as staleness, $T_a - T_f$. Feather assumes that all replicas have sufficiently synchronized GPS clocks.

In summary, Feather guarantees:

$T_q - L \leq T_f \leq T_q$

Figure 5 illustrates these semantics. 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.

Feather consists of four components on each node: a persistent storage, a query server to receive and execute queries, a push daemon to push periodic updates to higher-layer nodes, and a receive daemon to receive updates from lower-layer nodes. The push daemon is responsible for replicating updated data upstream, marking it as dirty and sending it in batches sorted by timestamp. The receive daemon is responsible for receiving updates from the push daemon and storing it in the persistent storage. The query server is responsible for executing global queries.

The global queries are processed hierarchically, with each replica determining the set of children needed to execute the query and then recursively sending it to each child. The querying algorithm is a recursive, parallel tree traversal. The nodes execute queries in parallel and determine if the data they have is recent enough to answer the query locally or if they need to visit a child.

TABLE 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

The actual freshness time for the result is defined by the minimum of the latest update time for the current node and the freshness returned by each of the sub-queries on sub-trees. This depends on the push period and depth of the hierarchical network.

The result sets are updated incrementally by adding rows for non-aggregate queries and updating values for aggregate and group by queries (e.g. sum, maximum/minimum, matching groups). The current implementation assumes rows sets are disjoint and only updated by the same edge node. More complicated aggregation algorithms are required to handle disjoint data that is left for future work. For aggregation queries (*MIN, MAX, SUM, COUNT*), a single value is retrieved as result, while for AVG, two values are required. If there is a *GROUP BY* clause, the results are computed for each group and sent to the parent node for merging. *WHERE* clause is applied locally and sent to parent node, while *DISTINCT, ORDER, LIMIT* clause is aggregated at the final layer.

Feather also offers analytical information for each query result, including the number of participating nodes, the number of included data rows, and an estimate of excluded data rows due to freshness constraints or link errors. We implemented a prototype of Feather as a Kotlin standalone application that uses Cassandra as its persistent storage.

Results

Feather's performance is evaluated through controlled experiments and a real-world deployment processing Twitter data across multiple continents. It is able to provide fresher answers than cloud-based execution while reducing network bandwidth and load on edge nodes.

We conduct controlled experiments to assess the performance of Feather using the New York taxi dataset [13]. The experiments use one of three network topologies: wide, deep, and medium, with varying depths and splits as shown in Table 1.

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.

Figure 6 shows the performance 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 also observe that frequent pushes (top row) result in much fresher answers, at the cost of increased load on the network.

CONCLUSION AND OPEN DIRECTIONS

Edge computing aims to provide highly responsive service by moving data processing and management resources closer to end-users and devices. This requires new architectures for running applications and storing/managing their data. In this manuscript, we presented PathStore, SessionStore, and Feather, which are systems that 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. We believe that the systems presented in this research can be improved in the following terms:

- **Improved replication strategies:** Replicate and cache data and processes on regional datacenters with data consumption determining caching policy as storage and process get cheaper.
- **Privacy:** Benefit of edge computing is local storage and processing, with only some analytical data sent to the cloud, protecting end-user data in healthcare and other applications.
- **Resource allocation:** Need global strategies to manage limited processing, storage, and bandwidth resources in edge datacenters to ensure performance and reliability.
- **Service level agreements (SLAs):** Research direction to identify SLAs for databases and processes on the edge that guarantee throughput, data availability, and performance.
- **Billing:** How to define billing mechanisms for edge computing, considering parameters such as resources used, service guarantees, etc.
- **Fail-over capabilities:** Balance between reliability and performance in maintaining uninterrupted data delivery services when networks on the edge of the network experience failures.

This paper focused on bridging the gap between the limited real-world implementations of data management on the edge and the many theoretical ideas for its use. By implementing systems that fit current real-world networks and address challenges, edge computing can evolve into a disruptive technology that offers significant benefits to organizations, businesses, and users for years to come. ■

Seyed Hossein Mortazavi is a senior researcher at Huawei Canada in Toronto. He received his PhD in Computer Science at the University of Toronto. His interests include designing next generation edge computing systems, as well as improving networks for the Data Center.

Mohammad Salehe was a PhD student at the University of Toronto. His interests included Distributed Cloud Systems. Salehe passed away in the PS752 flight crash.

Moshe Gabel is an assistant professor in the Department of Electrical Engineering and Computer Science at York University. His research lies in the intersection of distributed algorithms, systems, and machine learning as well as edge computing, specifically making geo-distributed data analysis more practical and accessible to typical software developers.

Eyal de Lara is a professor in the Department of Computer Science at the University of Toronto. His focus is on experimental research on mobile and pervasive computing systems. His research has been recognized with an IBM Faculty Award, an NSERC Discovery Accelerator Award, the 2012 CACS/AIC Outstanding Young Computer Science Researcher Prize, and two Best Paper awards.

REFERENCES

[1] New idc spending guide forecasts edge computing investments will reach $208 billion in 2023. February 2023. https://www.idc.com/getdoc.jsp?containerId=prUS50386323

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ACM, 13–16.

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

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

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

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

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

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

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

[10] David Reinsel, John Gantz, John Rydning. The digitization of the world from edge to core. Framingham: International Data Corporation, 2018, 16.

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

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

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

[14] Liang Tong, Yong Li, and Wei Gao. 2016. A hierarchical edge cloud architecture for mobile computing. *IEEE INFOCOM 2016 —The 35th Annual IEEE International Conference on Computer Communications*, 1–9.