Feather: Hierarchical Querying for the Edge

Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel and Eyal de Lara
Department of Computer Science
University of Toronto
Toronto, Canada
Email: {mortazavi, salehe, mgabel, delara}@cs.toronto.edu

***Abstract*—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 data centers. 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 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.

We evaluate Feather in controlled experiments using real-world geo-tagged traces, as well as a real system running across 10 datacenters in 3 continents. Feather combines 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.

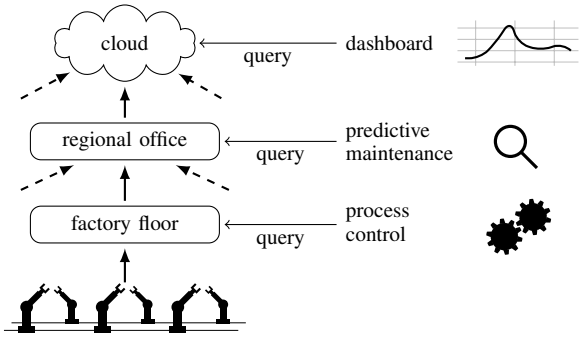


Fig. 1. Example IIoT application components deployed over a 3-tier network. Data is collected from each factory floor and must be sent up the hierarchy. Process control functions run on the factory floor, since they require fresh local data and low latency. Predictive maintenance models consume more resources, but use staler data from multiple production lines. Global dashboard runs in the cloud, and requires balancing data freshness with answer latency.

I. Introduction

Consider a hypothetical Industrial-Internet-of-Things (IIoT) application deployed over a 3-tier network [1], as shown in Figure 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 backend 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 [2]–[4], smart grid [5], [6], IoT and wearable devices [7]–[9], and healthcare [10], [11].

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 [8], [12]–[14]. 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 backend in our example. Since the most up-to-date data 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 [15]. 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 reconfigurations, and high latency [16], [17]. 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.

Our Contribution: 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 absense 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 tradeoff 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.

II. Background

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

Edge Networks: By *edge network*, we mean a hierarchical network where each node is a datacenter in which 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 [18], [19]. We refer to datacenters on the path from the cloud to an edge as *core nodes*. Note we do not consider user 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.

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 Internet-of-Things (IoT) devices can log huge amounts of data metrics, but store it locally to save on bandwidth and other costs [1], [20]. Figure 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 [2], transportation and traffic control [4], [21], and healthcare [10], [11]. 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 [5], [6]. 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 [22], [23]. Data-heavy edge computing applications are therefore built to accommodate relaxed consistency guarantees such as eventual consistency [24]. 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 [25], 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 [26].

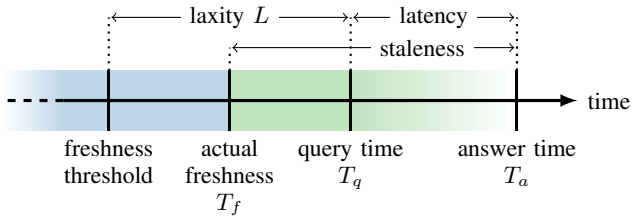


Fig. 2. The freshness guarantees for Feather global queries. Actual freshness T_f is guaranteed to be between $T_q - L$ and T_a. Any row created before T_f (blue) is guaranteed to be included in the results, while rows created after T_f (green) may or may not be included.

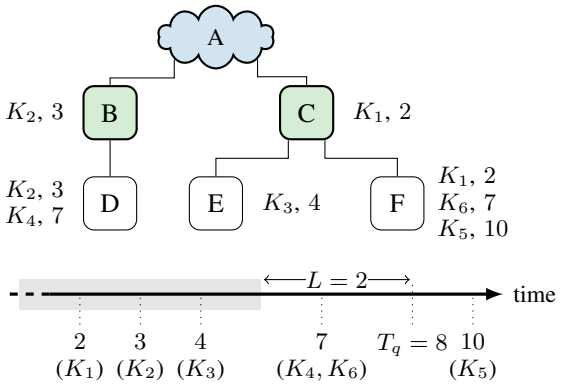


Fig. 3. Edge network with 6 rows K_1 to K_6, with row update times (numbers next to keys). A query submitted to A at time $T_q = 8$ with laxity of $L = 2$ must retrieve all keys updated before time $T_q - L = 6$, and must therefore access nodes B, C and E, but not D and F.

III. 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 [14]. *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.

A. 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 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 2 illustrates these semantics.

For example, consider a dashboard query from the industrial monitoring application (Figure 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:

```
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 that while 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 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 laxity $L = 2$, 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 [27] and t-freshness [28]; we discuss these in Section VI.

B. High Level Design

Feather's assumptions are common in geo-distributed, eventually-consistent databases [13], [14], [28]–[32]. As described in Section II, 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. Specifically, clock drift must be lower than the minimum one-way latency of any link, i.e., up to a few milliseconds. While requiring certain engineering effort, such accuracy is well within the capability of GPS clocks and IEEE-1588 [33] which reach microsecond accuracy for even low-cost hardware [34]. The hierarchical structure of our system follows the topology of the underlying network. The local replica at each node need only know about its parent and keeps track of its children, which allows for both horizontal and vertical scaling of the system. As with any eventually-consistent database, users can insert, update, or read data at any node, and it will be eventually propagated up the hierarchy.

1) Local Persistent Storage: Each Feather replica contains a high-performance persistent store for both user data as well as metadata used by Feather[1]. 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 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.

2) 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 III-C) to guarantee freshness, and is inspired by how stream processing systems such as Flink [35] and Google Cloud Dataflow [36] 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 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.

C. Answering Global Queries

The hierarchical global querying algorithm provides the query semantics defined in Section III-A. Unlike local application queries, which are served directly from the persistent store, the global queries described in Section III-A 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 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

[1]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.

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 ← ∅
2  Initialize result R
3  foreach child c ∈ children(n) do
4      if last update time from child T_u(c) < T_q − L then
5          Add c to accessed children: A ← A ∪ {c}
6          Send global query q to child c
7  R_loc ← 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 ← 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 ← min(T_f, T_c)
13 Return results R and actual freshness T_f
```

$\min_c\{T_u(c)\}$ (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 V.

D. Reversed Semantics for Providing Latency Guarantees

Feather also lets users specify limits on the query response time. Recall the example query from Section III-A. 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 III-E) 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 III-F), 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.

E. 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 V, 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 [37]–[39].

F. 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 data centers 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 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

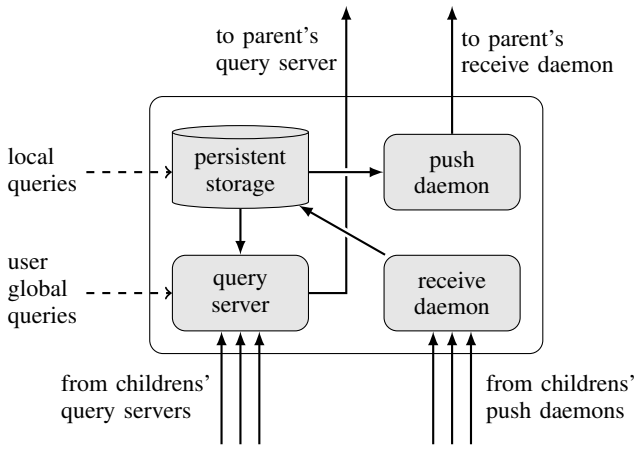


Fig. 4. The main components of an Feather replica. Global queries are sent to the query server for execution. To provide fast local access, applications run local writes and reads directly on the persistent store using a user-level library that handles additional columns needed by Feather.

the number of missing rows, users can then intelligently use or discard the query results, depending on the application.

G. Adding and Removing Nodes

In Feather, modifications to the topology are local operations and only involves 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 anytime.

IV. Implementation

We implemented a prototpe 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.

A. Architecture

Feather is comprised of four components on each node, shown in Figure 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.

1) Persistent Storage: Our implementation uses Cassandra [40] 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 to provide strong consistency within the datacenter. Feather's design is independent of the choice of the underlying datastore, and can be adapted to use other systems.

2) 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 [13], [14]. 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.

3) 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 daemonson the node and storing data on the persistent storage. It also records the latest update time as received from each child.

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

B. 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 nodes' 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 III-B). As described in Section III-B, 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 [41], 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.

C. Implementing Global Queries

As shown in Figure 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

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

Fig. 5. The syntax of a query in Feather.

almost all features provided by CQL, specifically all 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 III-C. 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 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.

D. 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 IV-E.

To perform aggregation queries such as `MIN, MAX, SUM, COUNT` on a node, only a single value is retrieved as a result from a children nodes, for queries involving `AVG`, two values: the average and the number of elements in the set is required to perform the aggregation. 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 [42]–[44], this is not the focus of this work.

E. 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 [45] and Count-Min sketches [46] to detect conflicts.

Second, while Feather supports deletion by the application, 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.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 [13]. Alternatively, the global querying mechanism can be extended to perform an upwards traversal followed by the usual downward traversal.

V. Evaluation

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

We evaluate Feather's performance on several metrics:

TABLE I
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

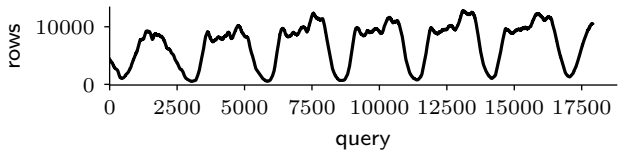


Fig. 6. Number of rows covered by each query over the length of experiment.

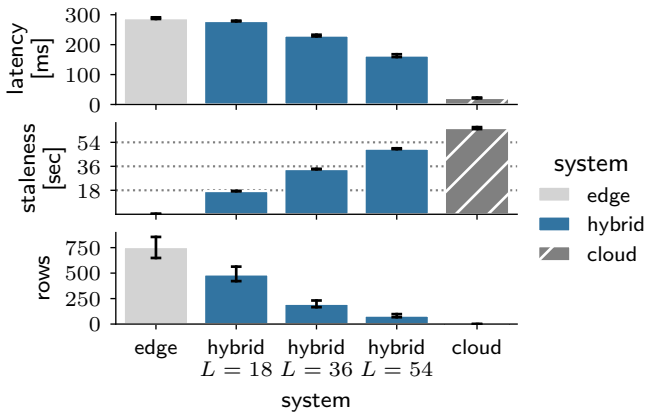


Fig. 7. Mean query latency, result staleness, and bandwidth (rows sent over the network) when running global queries on the medium topology using different global querying systems.

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

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

A. 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 experiments uses one of three topologies, summarized in Table I: *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 I (the end-to-end latency and tier division is similar to real edge networks, such as in Section V-F). The network delays and jitter between the containers is emulated using Linux's Traffic Control [47], [48], 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 [49], 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 (pick-up and drop-off zones), as well as information such as fare amount, number of passengers, and so on. When inserting data rows, we use each row's drop-off zone to determine which edge node to add it to. The dataset contains 265 such geographical zones. We distribute the edge nodes geographically such that when we map each zone to the nearest edge node, there is a roughly equal number of zones per edge node.

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 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.

B. Latency/Staleness Tradeoff

Feather is designed to provide controlled tradeoff of answer latency and answer staleness in global queries. This tradeoff

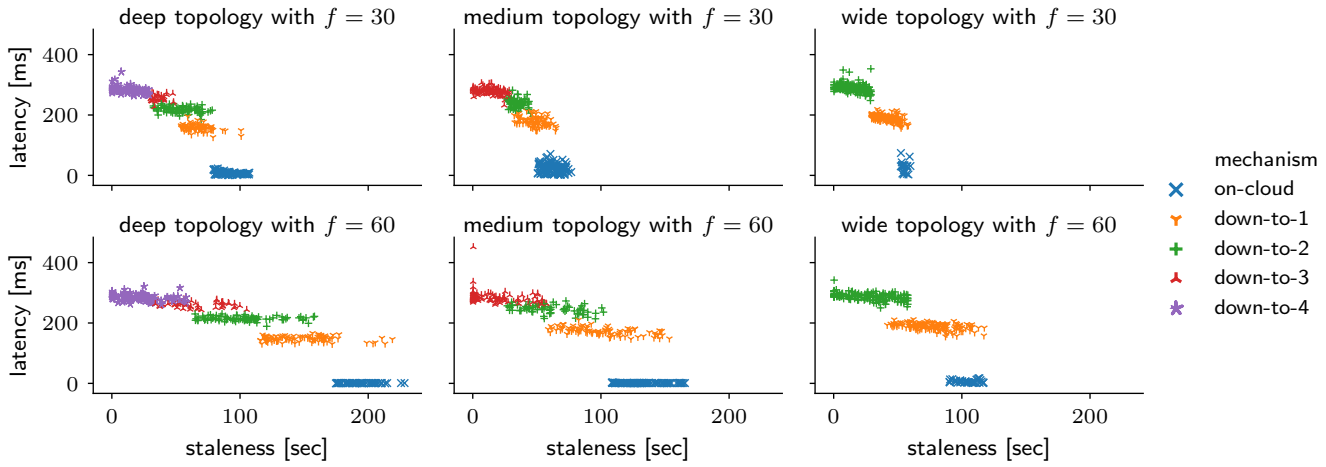


Fig. 8. Staleness vs latency of the answer for each query. Colors/markers indicate the depth of most distant node which was involved in answering the query. For clarity, we only show a sample of the queries.

depends on query laxity, network topology, period of the push demon, and data update distribution among the edges. To evaluate this tradeoff, we run controlled experiments where we vary the first three, while fixing the data distribution to the NYC Taxi data.

We compare Feather's hybrid approach to the two prevailing approaches in edge computing: querying the cloud replica [14]), and querying the edge nodes [15]. Figure 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.

Figure 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 V-F). 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 tradeoff? The wider spread of latency for on-cloud 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 tradeoff of staleness vs. latency.

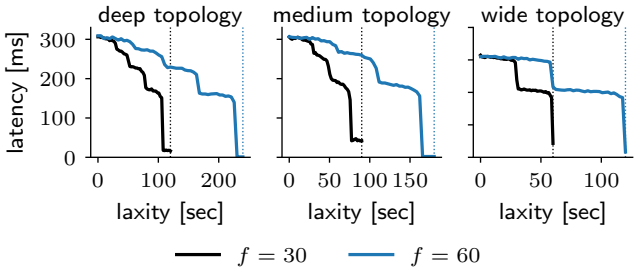


Fig. 9. 95th percentile of latency as a function of laxity for different with push period $f = 30$ seconds and $f = 60$ seconds. Shaded areas show standard deviation. Dotted lines show the time it takes data to be pushed from edge to the cloud $(D - 1) * f$.

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 9 shows how system operators can tune required laxity and push demon period to meet latency requirements. For 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 (alternatively, we can set a deadline constraint of 230ms directly). 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 tradeoffs.

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.

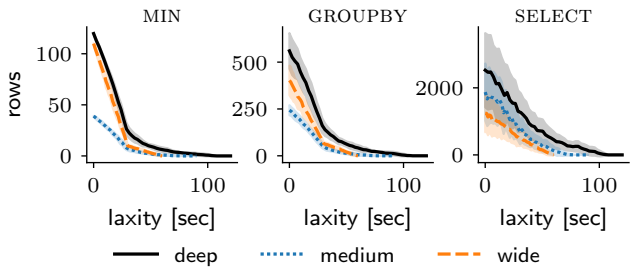


Fig. 10. Number of rows sent over the network for each types of queries across a range of laxity values. Shaded areas show standard deviation.

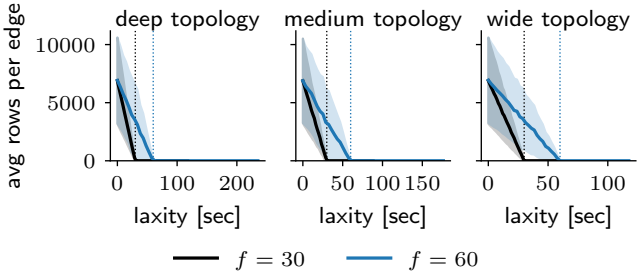


Fig. 11. Average number of rows accessed by each local edge replicas as a function of laxity for different with push period $f = 30$ seconds and $f = 60$ seconds. Feather shifts work from edges towards the core and cloud nodes, and once $L > f$ edges are seldom involved in answering global queries. Shaded areas show standard deviation. Dotted vertical lines show push period f.

C. 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 these query. This depends not only on the data, but also the type of query. Let $\text{rows}(n)$ 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, $\text{rows}(n)$ is 0.

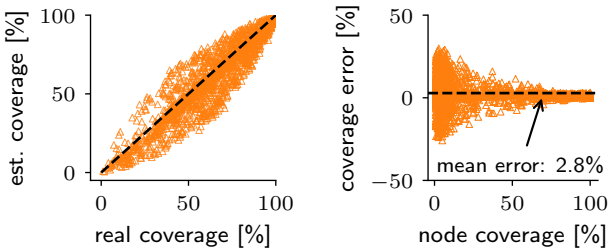
Figure 10 shows the bandwidth reduction for each query type and topology. Feather reduces bandwidth across the board with even a modest laxity, since queries are answered by a smaller set of nodes. Note that for queries that aggregate multiple response from all children to one response (MIN and GROUPBY 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.

D. Work at Edge Nodes

Edge notes are often resource constrained, and one of Feather goals is to offload work from the edge nodes towards the inner nodes in the network (core and edge nodes). Figure 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 in 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.

E. Coverage Estimation

As detailed in Section III-E, 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.



(a) Vs. actual coverage. (b) Vs. node coverage.

Fig. 12. Rows covered per query across all topologies and laxity ranges. (a) estimated row coverage compared to real row coverage. Dashed line shows equality (Y=X). (b) 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.

Each point in Figure 12(a) 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. Figure 12(b) 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%.

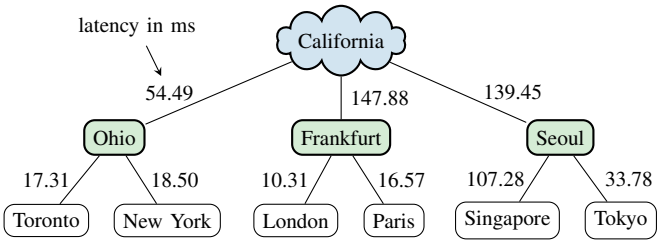


Fig. 13. Topology of the real-world experiment. Numbers indicate mean measured round trip time in milliseconds.

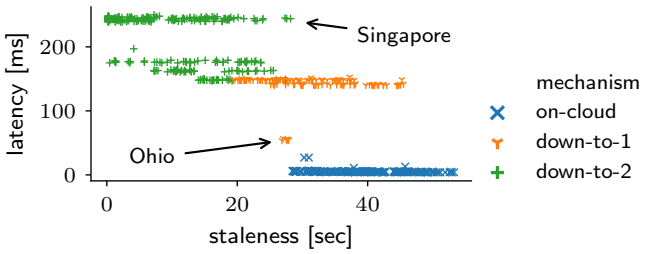


Fig. 14. Staleness vs latency of the answer for each query in the real-world experiment. For clarity, we only show a sample of the queries. The varied latency from cloud to the different nodes is reflected in the different latency-staleness clusters in the figure.

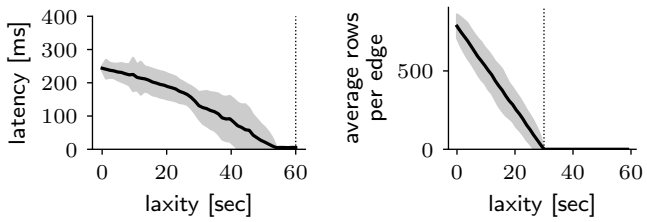
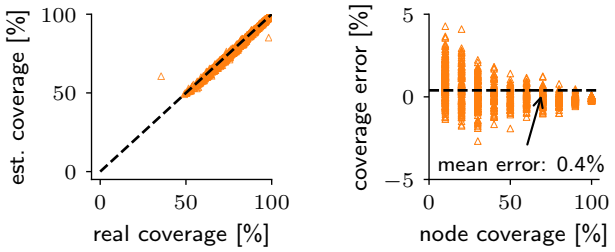


Fig. 15. Mean latency (left) and average work per edge (right) as a function of laxity. Shaded areas show standard deviation. Dotted lines on the right show push period f and on the left the time it takes data to be pushed from edge to the cloud $(D-1)*f$.



(a) Vs. actual coverage.

(b) Vs. node coverage.

Fig. 16. Accuracy of coverage estimation for all queries in the real-world experiment. For clarity we only show a sample of the data.

F. Real World Experiment

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 13 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 of 1 million tweets from New York City, Toronto, London, Paris, Singapore, and Tokyo over a one week period from December 2019 using Twint [50]. We speed time up for this experiment 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 hashtags per tweet in the past 10 minutes. For this experiment we do not add any artificial network delay or jitter.

Figure 14 shows the latecy/staleness tradeoff for queries in the twitter experiment. While the overall shape of the curve remains similar to those seen in Figure 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 15 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 16 confirms that coverage estimation remains very accurate in the real-world, with a mean error of 0.4%.

VI. Related Work

There are 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 [15] 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 [51] 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 [52], [53] or reduce bandwidth using lossy data transformations such as resampling [54]. 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 [55], Apache Impala [56] or Facebook's Presto [57] process queries on top of heterogeneous data stores such as Hive [58], PostgreSQL, Hadoop, MySQL and Apache Kafka [59] 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 [35], [60], [61] 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 [16], [17]. Recent research extends the stream process paradigm to the edge computing settings, as generic stream processing frameworks or bespoke applications [16], [62]–[68]. 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 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 [69]–[74]. In these studies, the network itself is the database [69] and to extract information from the network, various methods [72], [74] are proposed to aggregate and propagate data resulting from a query to a single base station. TAG [74] and TinyDB [72] 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 [75] and Microsoft Cosmos DB [76] 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 tradeoff between freshness, accuracy, and performance in continuous (streaming) queries was investigated by Heintz et al. [77], [78]. 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. [27] 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. [28] propose the t-freshness property which considers when operations begin rather than end, and use it to derive CAP-style impossibility results for the tradeoff 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.

VII. Conclusions

We proposed Feather: a geo-distributed, hierarchical, eventually-consistent tabular data store 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 show it provides a user-controlled tradeoff between latency, staleness, bandwidth, and load on edge nodes.

We plan to extend Feather in two directions. First, we want to support more types of applications by allowing data to flow "downstream" on-demand, and by improving the implementation for non-disjoint keys. Second, we want to investigate dynamic control policies for the latency/staleness tradeoff. By tuning the laxity parameter dynamically, we can better adapt to changes in data distribution and query patterns.

Acknowledgment

This work was supported by Huawei Technologies Canada Co., Ltd and Natural Sciences and Engineering Research Council of Canada (NSERC) Grant # CRDPJ 543885-19.

REFERENCES

[1] D. A. Chekired, L. Khoukhi, and H. T. Mouftah, "Industrial iot data scheduling based on hierarchical fog computing: a key for enabling smart factory," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4590–4602, 2018.

[2] B. Siregar, A. B. A. Nasution, and F. Fahmi, "Integrated pollution monitoring system for smart city," in *2016 International Conference on ICT For Smart Society (ICISS)*. IEEE, 2016, pp. 49–52.

[3] N. K. Giang, V. C. Leung, and R. 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*, 2016, pp. 91–98.

[4] N. Mohamed, J. Al-Jaroodi, I. Jawhar, S. Lazarova-Molnar, and S. Mahmoud, "Smartcityware: A service-oriented middleware for cloud and fog enabled smart city services," *IEEE Access*, vol. 5, pp. 17 576–17 588, 2017.

[5] Y. Saleem, N. Crespi, M. H. Rehmani, and R. Copeland, "Internet of things-aided smart grid: technologies, architectures, applications, prototypes, and future research directions," *IEEE Access*, vol. 7, pp. 62 962–63 003, 2019.

[6] R. Morello, C. De Capua, G. Fulco, and S. C. 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*, vol. 17, no. 23, pp. 7828–7837, 12 2017.

[7] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, "Computation offloading and resource allocation for low-power IoT edge devices," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, 2016, pp. 7–12.

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[9] J. Ren, H. Guo, C. Xu, and Y. Zhang, "Serving at the edge: A scalable IoT architecture based on transparent computing," *IEEE Network*, vol. 31, no. 5, pp. 96–105, 2017.

[10] H. Wang, J. Gong, Y. Zhuang, H. Shen, and J. 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)*. IEEE, 2017, pp. 1213–1222.

[11] R. K. Barik, H. Dubey, and K. 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)*. IEEE, 2017, pp. 477–481.

[12] A. Trivedi, L. Wang, H. Bal, and A. Iosup, "Sharing and caring of data at the edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, Jun. 2020.

[13] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[14] S. H. Mortazavi, B. Balasubramanian, E. de Lara, and S. P. Narayanan, "Toward session consistency for the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[15] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A distributed multi-resolution time-series datastore," in *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 2013, pp. 288–297.

[16] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable streaming for the mobile edge," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '19. New York, NY, USA: ACM, 2019, p. 153–158.

[17] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, "A survey on geographically distributed big-data processing using MapReduce," *IEEE Transactions on Big Data*, vol. 5, pp. 60–80, 07 2017.

[18] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

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

[20] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, and Q. Zhang, "Edge computing in IoT-based manufacturing," *IEEE Communications Magazine*, vol. 56, pp. 103–109, 09 2018.

[21] M. Chen, Y. Hao, K. Lin, Z. Yuan, and L. Hu, "Label-less learning for traffic control in an edge network," *IEEE Network*, vol. 32, no. 6, pp. 8–14, 2018.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual consistency," *Communications of the ACM*, vol. 57, no. 5, pp. 61–68, May 2014.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[24] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[25] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "Wanalytics: Analytics for a geo-distributed data-intensive world." in *CIDR*, 2015.

[26] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: Toward a distributed data store for fog computing," in *2017 IEEE Fog World Congress (FWC)*. IEEE, 2017, pp. 1–6.

[27] W. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '11, 2011, p. 197–206.

[28] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya, "Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, Jan. 2017.

[29] S. H. Mortazavi, M. Salehe, B. Balasubramanian, E. de Lara, and S. PuzhavakathNarayanan, "Sessionstore: A session-aware datastore for the edge," in *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, 2020, pp. 59–68.

[30] H. Gupta, Z. Xu, and U. 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.

[31] I. Psaras, O. Ascigil, S. Rene, G. Pavlou, A. Afanasyev, and L. Zhang, "Mobile data repositories at the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[32] B. Confais, A. Lebre, and B. 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*. Springer, 2017, pp. 40–79.

[33] F. Gao, Z. Huang, S. Wang, and Z. 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)*, 2016, pp. 2645–2649.

[34] P. Volgyesi, A. Dubey, T. Krentz, I. Madari, M. Metelko, and G. Karsai, "Time synchronization services for low-cost fog computing applications," in *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, ser. RSP '17. Association for Computing Machinery, 2017, p. 57–63.

[35] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[36] S. Krishnan and J. L. U. Gonzalez, "Google cloud dataflow," in *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 255–275.

[37] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with LSTM," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99.*, vol. 2, 1999, pp. 850–855.

[38] B. Doucoure, K. Agbossou, and A. Cardenas, "Time series prediction using artificial wavelet neural network and multi-resolution analysis: Application to wind speed data," *Renewable Energy*, vol. 92, pp. 202–211, 2016.

[39] C. Liu, S. C. Hoi, P. Zhao, and J. Sun, "Online ARIMA algorithms for time series prediction," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[40] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[41] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[42] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. Association for Computing Machinery, 2016, p. 101–114.

[43] D. M. Kane, J. Nelson, and D. 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*, ser. PODS '10, 2010, p. 41–52.

[44] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, Oct. 2008.

[45] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, 2014, pp. 75–88.

[46] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[47] W. Almesberger, "Linux traffic control-implementation overview," EPFL ICA, Tech. Rep., 1998.

[48] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, 2005, pp. 18–23.

[49] N. Taxi and L. Commission, "New york city trip record data," https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2020.

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

[51] Y. Yang, Q. Cao, and H. Jiang, "EdgeDB: An efficient time-series database for edge computing," *IEEE Access*, vol. 7, pp. 142 295–142 307, 2019.

[52] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[53] M. Jang, H. Lee, K. Schwan, and K. Bhardwaj, "SOUL: An edge-cloud system for mobile applications in a sensor-rich world," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 155–167.

[54] P. Lou, L. Shi, X. Zhang, Z. Xiao, and J. Yan, "A data-driven adaptive sampling method based on edge computing," *Sensors*, vol. 20, no. 8, p. 2174, 2020.

[55] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015, pp. 1383–1394.

[56] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. Wanderman, and M. M. Yoder, "Impala: A modern, open-source SQL engine for Hadoop," in *Proceedings of the 7th biennial conference on innovative data systems research*, 2015.

[57] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte *et al.*, "Presto: SQL on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1802–1813.

[58] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," in *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE, 2010, pp. 996–1005.

[59] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[60] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ Twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[61] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache Spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[62] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "SpanEdge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 168–178.

[63] M. D. de Assunção, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1 – 17, 2018.

[64] E. G. Renart, J. Diaz-Montes, and M. Parashar, "Data-driven stream processing at the edge," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 31–40.

[65] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 377–392.

[66] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "LAVEA: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17, 2017.

[67] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15, 2015, pp. 426–438.

[68] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[69] R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, "The sensor network as a database," Citeseer, Tech. Rep., 2002.

[70] C. Intanagonwiwat, R. Govindan, and D. 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*. ACM, 2000, pp. 56–67.

[71] X. Li, Y. J. Kim, R. Govindan, and W. Hong, "Multi-dimensional range queries in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, 2003, pp. 63–75.

[72] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[73] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM Sigmod record*, vol. 31, no. 3, pp. 9–18, 2002.

[74] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[75] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

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

[77] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing timeliness and cost in geo-distributed streaming analytics," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 232–245, 2020.

[78] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 361–373.