FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access

Harshit Gupta
Georgia Institute of Technology
Atlanta, Georgia
harshitg@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, Georgia
rama@gatech.edu

ABSTRACT

We design Fogstore, a key-value store for event-based systems, that exploits the concept of relevance to guarantee low-latency access to relevant data with strong consistency guarantees, while providing tolerance from geographically correlated failures. Distributed event-based processing pipelines are envisioned to utilize the resources of densely geo-distributed infrastructures for low-latency responses - enabling real-time applications. Increasing complexity of such applications results in higher dependence on state, which has driven the incorporation of state-management as a core functionality of contemporary stream processing engines *a la* Apache Flink and Samza. Processing components executing under the same context (like location) often produce information that may be relevant to others, thereby necessitating shared state and an out-of-band globally-accessible data-store. Efficient access to application state is critical for overall performance, thus centralized data-stores are not a viable option due to the high-latency of network traversals. On the other hand, a highly geo-distributed datastore with low-latency implemented with current key-value stores would necessitate degrading client expectation of consistency as per the PACELC theorem. In this paper we exploit the notion of contextual relevance of events (data) in situation-awareness applications - and offer differential consistency guarantees for clients based on their context. We highlight important systems concerns that may arise with a highly geo-distributed system and show how Fogstore's design tackles them. We present, in detail, a prototype implementation of Fogstore's mechanisms on Apache Cassandra and a performance evaluation. Our evaluations show that Fogstore is able to achieve the throughput of eventually consistent configurations while serving data with strong consistency to the contextually relevant clients.

CCS CONCEPTS

• **Information systems → Key-value stores**; Data replication tools; *Distributed storage*;

KEYWORDS

Distributed key-value stores, edge computing, latency-consistency trade-off, context awareness

ACM Reference Format:
Harshit Gupta and Umakishore Ramachandran. 2018. FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3210284.3210297

DEBS '18, June 25–29, 2018, Hamilton, New Zealand

ACM ISBN 978-1-4503-5782-1/18/06…$15.00
https://doi.org/10.1145/3210284.3210297

1 INTRODUCTION

Situation-awareness applications (e.g., large-scale video surveillance) continuously sense the environment and generate actionable knowledge in the form of events (e.g. detection of a suspicious vehicle). The actors involved in these applications are typically machines, or Internet-of-Things devices (e.g., CCTV cameras), requiring a sense-process-actuate control loop to work at machine-perception speeds. A cloud-based execution model for situation-awareness applications fails to deliver on this requirement due to the latency for streaming the sensor data to the remote Cloud datacenters, and the heavy network load that can ensue on the backhaul network connecting the sensors to the Cloud. In order to alleviate these performance issues, recently there is a trend of shifting computational resources closer to the sources and sinks of data, i.e., towards the edge of the network - called *fog/edge* computing[5]. Owing to the benefits of edge computing, situation-awareness applications - modeled as stream processing applications - are scheduled over highly geo-distributed infrastructure [8]. The events generated by situation-awareness applications, to deliver more value, are tagged with spatio-temportal contextual attributes, so as to enable querying over space and time. For various classes of applications, like smart surveillance and connected cars, the set of recently generated events and location-tagged attributes (e.g., state of a traffic light) acts as the application state - that guides the application actions in the future. Hence, system support primitives for *saving and retrieving events* that constitute the application state are critical in a service platform meant for geo-distributed applications. Contemporary stream processing platforms like Foglets [17], Apache Flink [7] and Samza [14] provide primitives for accessing and modifying the application state. Quite often, multiple application components on different edge nodes may want to share the application state – e.g., situation-awareness applications may have multiple processes working on events pertaining to the same geographical area; this would require moving the state out of memory into an out-of-core external store [12] - a design choice that is also supported by Apache Flink.

Keeping such application state information on Cloud-based data stores would defeat the purpose of placing the application components in geo-distributed fog/edge nodes since saving/retrieving state is in the critical path of the application execution and should incur as little latency as possible. Hence, the application state needs to be stored in a geo-distributed manner as well [9], leveraging the same edge nodes as those used for placing the computational components. Access to the application state in this geo-distributed setting warrants the same measures for throughput and fault-tolerance as in cloud-datacenters, namely, *replication and load-balancing*. Most importantly, replica placement has to be done taking into account the low-latency access requirement of applications. However, fog computing systems are susceptible to geographically correlated failures [10]. An eventually consistent data-store with geo-replication would satisfy both these requirements by placing replicas in a widely geo-distributed fashion with some replicas in proximity for low-latency access. However, many stream-based situation-awareness applications require strong consistency guarantees on the application state [3]. In fact, researchers at Google observe that coping with eventual consistency in the application layer requires significant development time and often leads to complicated and error-prone mechanisms [18]. Hence strong consistency should be provided by the datastore layer itself. Providing strong consistency (e.g, quorum-based consistency maintenance) on a geo-replicated data-store would lead to high latency. Hence tolerance from geo-correlated failures and low-latency are conflicting objectives when strong consistency is a requirement.

In this paper, we tackle the problem of providing both the conflicting but valid objectives of fault-tolerance and low-latency while satisfying consistency guarantees in geo-distributed key-value stores. The key insight about geo-distributed applications is the dependence of relevance of a certain data-item on the client's context. For instance, in the smart city domain, information related to a particular city may be relevant to clients who are in close proximity to that city. Using this context-sensitive characteristic of applications, we design a replication strategy that guarantees strong consistency for relevant data replicas and eventual consistency for the replicas intended primarily to ward off geo-correlated failures. The strategy is to place replicas close to the relevant clients (for low latency) and also in geographically distant locations (for fault-tolerance).

FogStore[1], which embodies the design principles for achieving fault-tolerance and low-latency for strongly consistent access to application state makes the following contributions:

- Develop a notion of *relevance* for situation-awareness applications, which is formalized as *Context-of-Interest* (CoI) - which determines the degree of consistency at which queried state should be reported;
- Propose a location-aware replica placement strategy to guarantee low latency and tolerance from geographically correlated failures and a quorum selection policy that guarantees strongly consistent operations for *relevant* data;
- Implement the proposed policies in Cassandra, a popular key-value store in the Cloud software ecosystem;

[1]A short paper describing the idea and preliminary results has been puslished in proceedings of the 1st Fog World Congress 2017.

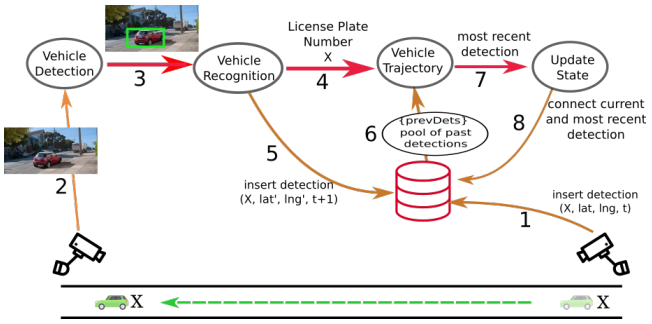


Figure 1: Schematic of multi-camera tracking of suspicious vehicles. The numbered arrows suggest the temporal order of flow of information.

- Conduct emulation-based experimental studies to quantify the performance of FogStore.

The rest of the paper is organized as follows. Section 2 outlines a use-case from the smart surveillance domain that serves as the driving application for Fogstore. Section 3 provides the necessary motivation and conceptual background for this paper. Section 4 presents the architectural elements of Fogstore while Section 5 discusses the specific implementation details on Cassandra. Section 6 showcases the performance of Fogstore in different settings. We conclude in Section 7 with directions for future work.

2 MOTIVATING USE-CASE

To highlight the necessity of a system like FogStore, we present a motivating use-case that poses strict latency and staleness requirements on the data-store. Consider a distributed camera network that may be deployed on urban roadways, feeding real-time video streams for multi-camera tracking of suspicious vehicles.

The application can be logically partitioned into components, as per the MobileFog programming model [11], each performing a specific function and having well-defined input-output characteristics. A schematic of the application is presented in Figure 1. Upon detection of a vehicle in a video frame, the application extracts the identity of the vehicle by reading the license plate to get the unique identifier of the vehicle. It then inserts the detection of that vehicle along with location and time into the set of detections. The application also maintains the complete trajectory information of the vehicle, in that, for each detection of a vehicle, it saves the location and time when the car was detected before that. To achieve this, for each detection the application retrieves the set of previous detections that took place within 5 KM and 10 minutes from the current detection. It looks for the most recent detection from them and adds the identifier of that detection to the current detection's *prevDetection* field.

It is evident from the schematic that access to application state lies in the critical path of application execution, hence making correct execution of the application contingent on fast access to state. For instance, if the insert of a vehicle detection is slow, the vehicle may be detected by an adjoining camera and not mark the former detection as the previous one - hence missing that detection from the trajectory. It is worth noting that the presented application has

Algorithm 1 Vehicle tracking algorithm

```
1: procedure onDetectVehicle(V, X, Y, T)
2:     K ← INSERT INTO vehicle_detections (V, X, Y, T)
3:     prevDets ← SELECT * FROM vehicle_detections
   WHERE (x, y) within 5 km &t WITHIN 10 min ORDER BY t
4:     prevDet ← k.V : k has most recent time in prevDets
5:     UPDATE K in vehicle_detections SET K.prevDet = prevDet
```

a dependence on contextually relevant events, specifically previous detections within a 5 km radius and at a maximum of 10 minutes before the current detection. Events that don't fall under this space-time filter may be past detections of the car but are typically not relevant for generating a fine-grained trajectory. Furthermore, retrieving the entire set of previous detections of the particular car and searching for the most recent detection could be slow.

3 BACKGROUND

3.1 Fog/edge computing

Fog/edge computing is defined as a non-trivial extension of cloud computing with highly geo-distributed virtualized resources placed in close proximity to end-users and devices. The main driving force behind this dense geo-distribution is a result of demands for lower response times, by applications like AR/VR and connected vehicles, and for reducing backhaul bandwidth consumption by data-intensive applications like city-scale smart surveillance. Although the infrastructure model is radically different from the way data-centers are constructed, the convergent vision for fog computing is to provide the same set of platform services so that application development on the Fog is compatible with that on the cloud. Application state management is one such platform services that is provided by popular execution runtimes (e.g., Apache Flink , Samza and Foglets) through a key-value store as the state backend.

Confais et al. [9] evaluate the performance of several off-the-shelf object stores (Cassandra, Rados and Inter-Planetary File System) in the fog computing environment using the Grid'5000 testbed, and rank them based on performance metrics like data access latency and network communication overhead. Their evaluation also takes qualitative metrics like mobility-support and fault-tolerance into account. They identify IPFS as the best off-the-shelf software for building fog computing object-storage systems. In this paper, however, we choose Cassandra to be the base system - firstly due to its easy extensibility and flexibility and secondly because IPFS treats each object as immutable, which is not necessarily the abstraction that applications expect when they interact with the datastore to read/write their state. Nevertheless, the metrics that they identify as crucial for data-stores in fog computing serve as guidelines when designing Fogstore.

3.2 Replication and consistency in Dynamo-style databases

Key-value stores built on the Dynamo-style model offer fine-grained tuning options for consistency per-operation. Figure 2 shows how Cassandra (a popular Dynamo-style key-value store) distributes data among the cluster nodes. The data-item's key is hashed to determine the nodes responsible for storing it (replicas). The client making read/update requests can specify a consistency level for that specific operation, which determines the number of nodes that need to execute that operation before the client is acknowledged of its completion. For example, an update operation with consistency level of *TWO* would update the copy of the data-item on 2 of the replica nodes and acknowledges that the update was successful. The update is propagated to the rest of replicas in an eventual manner. The choice of this consistency level plays a crucial role in tuning the tradeoff between latency and consistency. Eventually consistent implementations use a consistency level of *ONE* while strongly-consistency implementations use consistency level of *QUORUM*. In a highly geo-distributed datastore, synchronizing with all replicas for every operation can be extremely slow, given that the replicas may be stored on nodes with a high network round-trip-time from the client/coordinator.

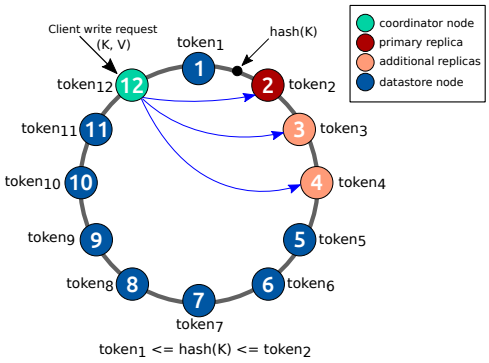


Figure 2: Demonstration of write-request path for a 12-node Cassandra cluster on keyspace with Replication Factor (RF) = 3

3.3 PACELC theorem : tradeoff between latency and consistency

The PACELC theorem extends the CAP theorm and presents conditional statements, saying that under network partitions a database system has to trade-off between Availability and Consistency (similar to CAP theorem) but under partition-free conditions it has to trade-off between Latency and Consistency. There has been a considerable amount of research exploring the latency-consistency trade-off - especially after the advent of Dynamo-style database systems that allow a per-operation tuning of consistency. McKenzie et al. [13] show how the latency-consistency trade-off can be made further finer-grained - than the granularity built into off-the-shelf Dynamo-style systems - and provide probabilistic guarantees on the consistency and latency of query results, which are useful for applications that may be able to tolerate some amount of inconsistency for lower-latency. Rahman et al. [15] go a step further by making this fine-grained tuning adaptive based on network conditions and optimize either latency or consistency while providing a guaranteed SLA for the other. Machine learning techniques have also proven to be effective in predicting the client-centric consistency level that satisfies the latency and staleness thresholds [19]. However

the above systems are meant for datastores running in cloud datacenters with clients being applications co-located in datacenters themselves. The notion of proximity of replicas from the clients has the granularity of a datacenter i.e., data is considered local if it is located in the same datacenter as the client. This notion of proximity breaks when placing replicas in such a densely geo-distributed and heterogeneous infrastructure as fog/edge computing, where the granularity of proximity needs to be more fine-grained and independent of constructs like datacenters and racks.

A rather recent proposal is GPlacer [22], which performs placement of replicas across multiple datacenters to minimize latencies of consistent transactions. Their focus is on partitioned fully-replicated databases and they statically choose datacenters to place replicas on for particular partitions. However, their replica placement heuristics takes on the order of 100s of seconds for an infrastructure size comparable to model fog computing deployments. The replication strategy is in the critical logic of Cassandra (which we use for implementation) and cannot tolerate slow implementations. Furthermore, their fault-tolerance model assumes that multiple datacenters cannot fail in a correlated manner - which is not the case in fog computing infrastructure.

3.4 Context-of-Interest

Applications built for the Internet of Things, and those for situation-awareness use-cases in particular, have a strong notion of locality-awareness. For example, in the case of publish-subscribe systems, the publishers and subscribers are often located geographically close to each other, because the events they are concerned with pertain to the local geographical area. This property was exploited by Teranishi et al. [20] with the Locality-aware publish-subscribe system. In general, geo-distributed applications possess the notion of *contextual relevance*, in that a piece of data is more relevant in a certain context than another. The notion of context is highly application-specific. For instance, in the use-case mentioned in Section 2, a vehicle detection is relevant at a higher degree in a context around the event in space and time, that is, within 5 kms from the event's location and within 10 minutes of the event generation. This is because of the nature of the application - to build an accurate trajectory of a vehicle, the previous detection has to be in proximity to the current detection. Similarly, smart cars accessing the state of a traffic light (red/green), where the cars located in the same city (proximity), would require consistent access to traffic light state to prevent collisions. Clients not in the same city would be able to live with data that may be stale, that is, eventual consistency may be good enough for them. Often the data generated by Internet-of-Things applications are used for for big-data analysis, which are offline batch processing tasks and don't require highly consistent data.

In the context of key-value stores, the notion of relevance translates to consistency and staleness as follows : *all relevant items must be available in a consistent manner with minimum staleness.* In other words, for a data-item I all entities for whom I is contextually relevant must see updates to I in the same order (serializability) and with as low staleness as possible (real-timeliness).

4 FOGSTORE ARCHITECTURE

4.1 Context of Interest

Here we formally define the notion of Context of Interest (CoI). Fogstore allows the architect of the database to specify a generic (possibly conservative) *region of relevance* (also called Context of Interest) around each data item such that operations originating from that region are executed in a strongly consistent manner [2]. In this paper, the notion of region is articulated as a circle in geographical space with a radius that determines the size of the region, represented by the parameter *CoISize*. For example, a certain application may require all clients within a 10 kilometre radius of the data item to expect strong consistency.

Fogstore would compare the location of a client to the location of the queried data-item and would provide a strongly consistent result only when the client's location lies within the CoI of the data-item. Clients beyond the region of relevance are typically not using the data for critical operations and hence can tolerate inconsistent/stale data to the same extent as an eventually consistent database.

4.2 Data model

To be able to leverage the context-aware optimizations of FogStore, data-items need to be annotated with additional contextual information, described as follows

- Each data item needs to possess spatial and temporal information, that is, location in terms of latitude-longitude and timestamp
- Each data item has a special column called *part_key* which serves as the key used to partition data across nodes in the distributed datastore. This key is composed of contextual information that would drive mechanisms of replica placement and quorum formation, as described in detail in subsequent subsections. One major concern in having data partitioning on the basis of location is to load-balance the data-items across datastore nodes located in the region for better throughput, which we handle by creating a hybrid partitioning key containing both location information and consistent hash of the data-item's key (described in detail in Section 5).

```
{
  "vehicle_id": "A54 3527",
  "location": {
    "latitude": "33.42553",
    "longitude": "-84.74456",
  },
  "timestamp": "1520123197",
  "part_key": "djgw0315209685"
  "uuid": "8de5b3dc"
}
```

Listing 1: A sample data-item (spatio-temporal event) in the vehicle-tracking detection set. The primary key of the table is a combination of *part_key*, *vehicle_id* and *uuid* fields, so that multiple detections of the same vehicle by different cameras can be differentiated.

[2] We refer to one-copy serializability as strong consistency guarantee in this paper.

4.3 Differential consistency for replicas

We would like to revisit the two fundamental requirements of situation-awareness applications:

- Clients sharing context with the data-item require low-latency access to strongly consistent data
- Placement of all replicas in proximity may lead to complete data loss under geographically-correlated failures

These requirements lead us the design decision of having two classes of replicas, which is also illustrated in Figure 3.

(1) **In-CoI replicas** : Placed on nodes located at low network delay from the clients, typically within/around the CoI. Their maximum distance from the data-item's location is determined by a parameter *InCoIDist*. This parameter determines the location of InCoI (strongly consistent) replicas and hence has implications on the latency of query operations. These replicas are kept consistent by enforcing that all read and write quorums include a majority of these replicas. These replicas are meant to provide both latency and consistency to users that are contextually relevant.

(2) **Out-CoI replicas** : The purpose of these replicas is to provide tolerance from geographically-correlated failures. They are placed on datastore nodes which are a minimum distance away from the data-item's location, a parameter called *OutCoIDist*. This parameter determines the location of Out-CoI replicas, and hence affects the geographical separation between InCoI and OutCoI replicas - and thereby the fault-tolerance. These replicas are kept eventually consistent by propagating updates asynchronously, and hence are never included in the read/write quorum operations originating from the CoI. These replicas could also serve as the source of data for big-data analysis of information generated at the edge, by using tools like Kafka Connect [2] that can use key-value stores as data sources for large-scale analytics.

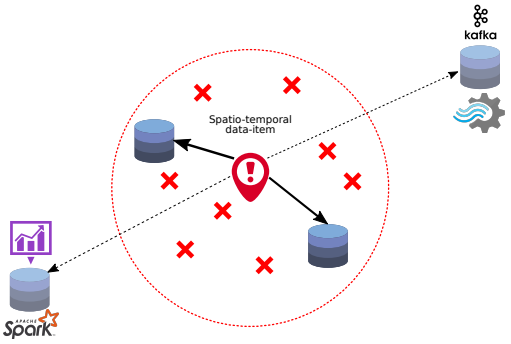


Figure 3: Illustration of the two types of replicas maintained by Fogstore along with the typical use-cases that both these types serve. The dotted circle around the spatio-temporal data-item denotes the Context-of-Interest for the data-item. We direct all reads/updates to that data-item originating within the CoI (represented by red crosses) to the InCoI replicas (which are shown inside the circle). The other (OutCoI) replicas are kept eventually consistent and serve the use-cases dealing with batch processing and monitoring.

It is worth noting that the concept of differential consistency for replicas based on their context is not a novel concept proposed in this paper. Apache Cassandra itself provides a consistency level called LOCAL_QUORUM [1] that makes sure that any operation selects a quorum that consists of the quorum-set of replicas on the local datacenter on which the request-handler (coordinator) node is located. This is done to avoid the high latency of inter-datacenter traversal. The updates are propagated to rest of the replicas (in other datacenters) in an eventual manner. However, this concept is easy to adopt in the context of cloud-based data-stores, which have well-defined notion of datacenters such that the network latency across datacenters is at least an order of magnitude higher than inter-datacenter latency. Highly geo-distributed data-stores based on fog computing infrastructures cannot make use of such a concept, especially when the concept of contextual locality is based on node location rather than simpler properties like subnet of IP address.

4.4 Handling skews in event workload

Situation-awareness application sense activities in the environment and transform them into events, that act as the input workload for application state data-stores. The variation of input workload traffic with location is, hence, highly contingent on the distribution of activity in a region. For example, even within a city, there may be surveillance cameras deployed only in a certain portion of the city (e.g. Downtown), which would create a skew in terms of the number of events pertaining to those areas compared to rest of the city.

We aim to store spatio-temporal events in proximity to the location that those events pertain to, which may lead to non-uniformity in the traffic served by datastore nodes, with those located in busy regions of the city handling more traffic than those in regions with lesser activity. A location-agnostic load-balanced distribution of data-items would ensure that all datastore nodes receive uniform traffic, which would, however, defeat the purpose of Fogstore. The skew-tolerant load-balancing proposed in this paper ensures that the load is uniformly spread across all the nodes in proximity to the data-items. This notion of proximity is tunable, and can be set to match the degree of skew in workloads.

Hence the major issues that the implementation of Fogstore should resolve are :

- Placement of replicas based on the data-item's context, so as to have replicas both in proximity serving the queries from relevant clients within the CoI with low-latency and also at a significant geographical-separation from the CoI to provide tolerance from geo-correlated failures
- Providing a transparent consistency interface to clients by determining the per-query consistency level based on contextual information of the client and queried item. This tuning of consistency-level is done by choosing the quorum of replicas in a way so as to deliver strongly consistent information to relevant clients and possibly stale information to clients outside the context-of-interest
- Avoid the formation of hotspots in data-partitioning due to inherent skews in the input workload

5 IMPLEMENTATION

To demonstrate the performance improvements by employing the proposed replica placement and contextual consistency model, we implement a prototype of Fogstore by extending Apache Cassandra, the details of which are presented in the following subsections.

5.1 Context-of-Interest aware data distribution

Fogstore uses the same mechanism to perform data-distribution across nodes as Cassandra, however, the policy used takes into account the context of data-item and datastore nodes.

5.1.1 Construction of token ring. Cassandra performs data distribution and replication by transforming the partition-key of a data-item into the token-space. The primary replica is selected based on the token of the data-item and that of datastore nodes, while further replicas are selected based on custom replication policies, for example placing atleast 2 replicas on each datacenter. Typical datacenter-based deployments of Cassandra use consistent hasing [4] to transform partition-key into token-space so as to achieve uniform load-balancing across cluster nodes.

In order to perform location-aware data distribution we use the spatial encoding (e.g., Geohash or Hilbert's curve [16]) of a data-item's location field to compose the partition-key. Datastore nodes are placed on the token-ring at a position equal to the spatial encoding of their location. In order to ensure that a data-item is placed on a node whose spatial-encoding is similar to the data-item's, it is important that tokens be ordered with respect to their spatial encodings for replica selection. Hence we don't use the popular consistent hashing algorithm for generating the token, but rather use the ByteOrderedPartitioner which simply translates a partition-key into a byte-sequence, and hence preserves the order of tokens.

The distribution of spatio-temporal event traffic is not expected to be uniform, with much more activity in densely populated regions and lesser activity in sparsely populated regions. Forming a node's token solely based on the spatial encoding would lead to partitions in the hash ring not being uniform in the number of tokens contained in them. This can lead to uneven distribution (poorload-balancing) of key-value pairs across datastore nodes. Consistent hashing forms one extreme of data partitioning that achieves best load balancing, but does not take into account spatial locality of replicas, while just using spatial encoding forms the other extreme which guarantees spatial locality but not load balancing. Hence, the two objectives of proximal data placement and load balancing prompts us to come up with a hybrid data partitioning scheme. We construct the partition-key PK of a data-item i as shown in Figure 4.

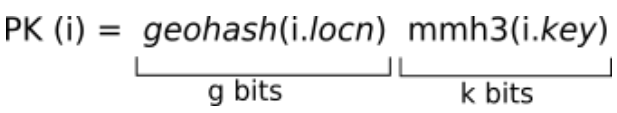


Figure 4: Illustration showing the inclusion of location-specific information in data-item's token to enforce proximal placement and the hash of key for even distribution.

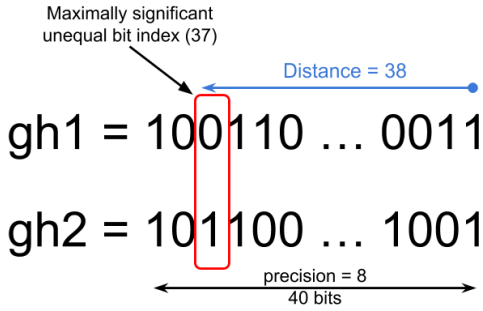


Figure 5: An illustration of distance between two spatial encodings, example being specific to Geohash.

The number of bits (g) of geohash included in the partition-key determines the granularity of location information that is taken into account when partitioning data-items. A smaller value of g would give lesser emphasis on spatial locality and hence would distribute key-value pairs across a large area - which may be good for applications that don't require very low latency. On the other hand, a higher value of g would preserve spatial proximity of replicas to a higher degree but would be less even in terms of load distribution. This split between keys is flexible and can be set according to the geo-distribution of fog nodes in a cluster and the point of tradeoff between data proximity and evenness of data distribution required by the application.

5.1.2 Notion of distance in spatial encoding. We define a notion of distance between two spatial encodings which is core to choosing the right nodes to place replicas on. Two spatial codes $d1$ and $d2$, have a distance of d if and only if $d1$ and $d2$ have the d^{th} bit as the maximally significant bit that differs in them. This is clarified in Figure 5. This notion of distance can be applied to any spatial encoding technique used to calculate tokens. It preserves the closeness of locations, that is, if two locations are closeby in terms of the proposed encoding distance, they are also closeby in terms of geographical distance. The converse, however, is not true, that is locations that are close in geographical distance may have spatial encodings that are far apart in terms of encoding distance. This property is evaluated in Section 6 when determining the location of OutCoI replicas.

5.1.3 Replica selection based on Context-of-Interest. Selection of replicas for a given data-item is done based on the spatial encoding distance of a node's token and the data-item's token. The replica selection alogrithm takes the following two parameters :

- *InCoIDist* : the maximum spatial encoding distance threshold for placing InCoI replicas.
- *OutCoIDist* : the minimum spatial encoding distance threshold for placing OutCoI replicas.

The replica selection policy for InCoI replicas is presented in Algorithm 2. The procedure $findInCoIReplicas$ takes, as input, the hash ring H, token of concerned item $itemToken$, $inCoIDist$ and number of replicas $nReplicas$ and returns a list of nodes that would host the InCoI replicas. The first node token in the token ring higher than the item's token is used as the starting point for iteration to find the

InCoI replicas (it_S). Each potential node that is within CoI's distance threshold and has a token with key portion lexicographically higher than item's token is a suitable candidate. Note that the comparison of token's key portion is solely for load balancing purposes. A fixed number of such replicas are selected. If the required number of replicas are not found after a complete traversal of the ring, the *inCoIDist* distance threshold is incremented by a small amount and the search is repeated. This increase in the threshold (which is specified by application developer) may harm the expected latency, but we choose to do so over declaring that no suitable replicas could be found.

A similar search is performed for getting the list of OutCoI replicas, the difference being that the spatial encoding distance between item's token and node's token now needs to be greater than the CoI distance threshold.

Algorithm 2 Replica selection algorithm

```
procedure FINDREPLICAS(H, itemToken, it_S, inCoiDist, N)
    replicas ← {}
    for it ∈ iterate(H, it_S) do
        if geohashDist(it.geohash, itemToken.geohash)
            ≤ inCoiDist
            &  it.key ≥ itemToken.key
            &  it not already chosen
        then
            replicas = replicas ∪ {it}

        if |replicas| == N
            break

    return replicas
procedure FINDINCOIREPLICAS(H, itemToken, inCoiDist, nReplicas)
    it_S ← H.find(itemToken)
    nFound ← 0
    inCoiReplicas ← {}
    while |inCoiReplicas| < nReplicas do
        R ← findReplicas(H, itemToken, it_S, inCoiDist,
                         nReplicas − nFound)
        inCoiReplicas ← inCoiReplicas ∪ R
        nFound ← nFound + |R|
        inCoiDist + +
    return inCoiReplicas
```

5.2 Context-of-Interest aware consistency

Selection of replicas constituting a quorum based on the context of interest of data item queried for is done based on the token of the coordinator node [3]. We use a parameter *CoISize*, which represents the size of the CoI in terms of spatial encoding distance. If the coordinator's token is less than *CoISize* units distant from the item's token, it lies within the CoI of the queried data item, and a quorum of InCoI replicas are selected to sychronously execute the operation. This ensures that all operations within the CoI will be highly consistent. Read operations on coordinators that are outside the CoI of queried data item are returned the version from whichever replica responds the fastest, thus providing eventual consistency. Write operations, on the other hand, need to update a quorum of the InCoI replicas before returning so that InCoI replicas don't enter an inconsistent state. Enforcing quorum for operations inside the CoI is fast as the quorum members are located in close network proximity from the client/coordinator node.

5.3 Optimizations for efficient range queries

Typical deployments of Cassandra key-value store use consistent hashing (Murmur3 hash) on the partition-key to partition the key-value pairs across nodes. Since the Murmur3 hash function does not preserve the order between input values, Cassandra does not allow range queries on the partition key. However, Fogstore uses the ByteOrderedPartitioner which preserves the order between partition-keys, thus making it possible to issue range queries on them. A typical range query has a structure as shown in Listing 2.

```
SELECT * FROM tracking_ks.detections
WHERE token(part_key)>=token(<min_part_key>) AND
token(part_key)<=token(<max_part_key>) AND
key='<object_key>'
```

Listing 2: Typical form of a range query that Fogstore handles. The minimum and maximum limits of partition key range queried for is calculated based on the bounding-box of locations that forms the range query. Note that we omit timestamp based filtering for simplicity, however that can be incorporated in the query provided a secondary index is built on the timestamp field.

For efficient execution of this query, a secondary index on the *key* column is created so that events within a particular partition can be queried in lesser time. Upon receiving such a query, the coordinator node splits the range into individual partitions and issues concurrent read requests to the replicas responsible for those partitions.

6 EVALUATING FOGSTORE

We use the distributed network emulator MaxiNet [21] to create the infrastructure setup for evaluation experiments. MaxiNet is an extension of the popular network emulator MiniNet, and allows the user to package an application as a Docker container and deploy it on a set of nodes. Latencies between the nodes are calculated using the geographical distance between the nodes based on the online tool WAN Latency Estimator[4].

6.1 Comparison of FogStore against typical replication policies

The first step towards proving the efficacy of the context-aware policies of Fogstore is to evaluate its performance against typical replication policies - quorum-based (strict) and eventual[5]. For this experiment, we build a representative infrastructure topology of datastore nodes - 4 inside Atlanta region and 4 more as remote datacenters. Yahoo Cloud Serving Benchmark (YCSB) is used to

[3]The underlying assumption is that client and coordinator node are in relative proximity

[4]http://wintelguy.com/wanlat.html

[5]The strict and eventual systems use Cassandra's SimpleStrategy policy for replication.

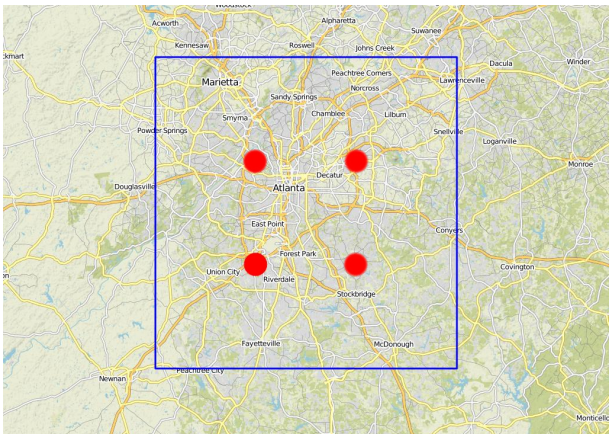


Figure 6: Map showing the locations of fog nodes in Atlanta region and the bounding box from where events are generated. The nodes shown in the figure are the ones that store the InCoI (consistent) replicas in Fogstore. The evaluation also consists of nodes located at remote locations : Houston, San Francisco, Chicago and Seattle.

generate spatio-temporal workloads of applications running in the Atlanta area, by having 4 client nodes running YCSB with 4 threads each. Each YCSB client is collocated with one of the datastore nodes in Atlanta area and mimics a situation-awareness application component making queries to the spatio-temporal state. To cover a wide variety of workloads, we experiments with workloads that have varying read-to-update ratios (20%, 50% and 80% reads) and varying distribution of selecting keys for operations (hotspot, latest and Zipfian). We consider mutable data-items to measure the behaviour of evaluated systems on consistency. We set the replication factor of eventually consistent and quorum-based store to 3. For the replication policy of Fogstore, we set number of InCoI replicas to 2 and OutCoI replicas to 1. Also, the read and write quorum for queries by clients inside the CoI are set to 2 and 1 respectively.

To be able to perform these experiments and collect the required metrics, we had to modify the core workload executor of YCSB. Here we describe the major modifications to YCSB here :

- We associate each version of an entity with a given key with a timestamp, corresponding to the wall-clock time when the YCSB client starts updating or inserting that version. Each entry in the table possesses the timestamp associated with the current version. We also record the time when an update/insert finishes. Since the experiment is done as an emulation on a single machine, the clocks of all datastore nodes and YCSB clients (Docker containers) use the time of the underlying physical machine.
- We associate each key generated by YCSB to a unique geolocation, since the spatial attributes of a data-item is central to the consistency model of Fogstore. The challenge here is to create a deterministic mapping between the key domain and the geolocation domain - so that this mapping stays the same across all the threads on all YCSB clients. Each key selected by the request distribution of YCSB is based on an integer value, which we convert to a sequence of bits. This bit sequence is then decoded (exactly same as Geohash decoding) to generate the latitude and longitude of the request.
- To bring about the notion of locality in access patterns in reads and updates, we ensure that for every data-item requested (for read/update), the client is within the context-of-interest of the data-item. This is done so that for all queries, the InCoI replicas would be chosen from the set of 4 local nodes. We use the location of the coordinator node to approximate the location of the client. Hence the parameter CoISize is set to 36 so as to ensure all possible events have the coordinator node inside their CoI.
- We use the latency aware load balancing policy to avoid choosing a coordinator that is too far away in terms of network latency - effectively defeating the purpose of Fogstore.

	20% reads	50% reads	80% reads
Latest	0.17	0.6	0.11
Hotspot	0.06	0.06	0.02
Zipfian	0.03	0.02	0.01

Table 1: Percentage of reads returning a version that was not most recently written. The percentage of inconsistent reads has been reported for workloads with varying proportion of read requests and key-selection distribution.

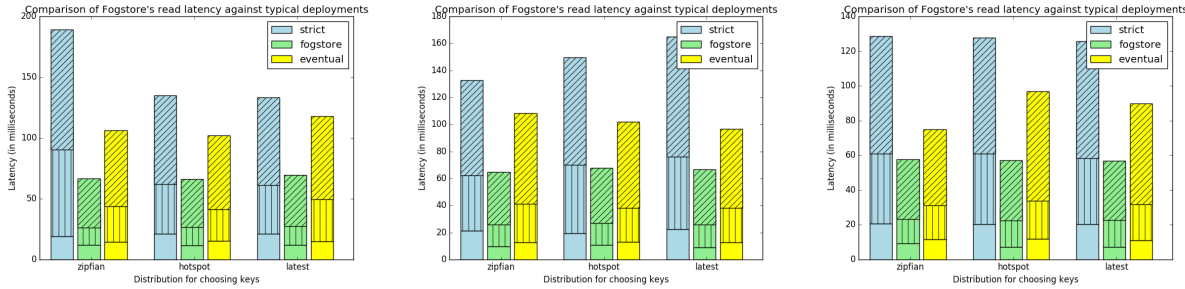
The metrics we are interested in are :

- Latency of read/update operations
- Throughput of read/update operations
- Client-centric degree of consistency

Applications that require strong consistency for data-accesses would use the aforementioned quorum-based (strict) datastore. The client's expectation from a datastore guaranteeing strong consistency is that a read always returns the most recent version whose update was successful. We analyze the trace of YCSB clients and for every read compare the version returned to the version that was written by the most recent successful update, and call it a violation if these versions don't match. We analyze the YCSB client trace against a quorum-based store and, following the expectation, don't find any violations as the read and write quorums overlap. This means that the application does not have to implement special logic for handling inconsistent/stale data reads.

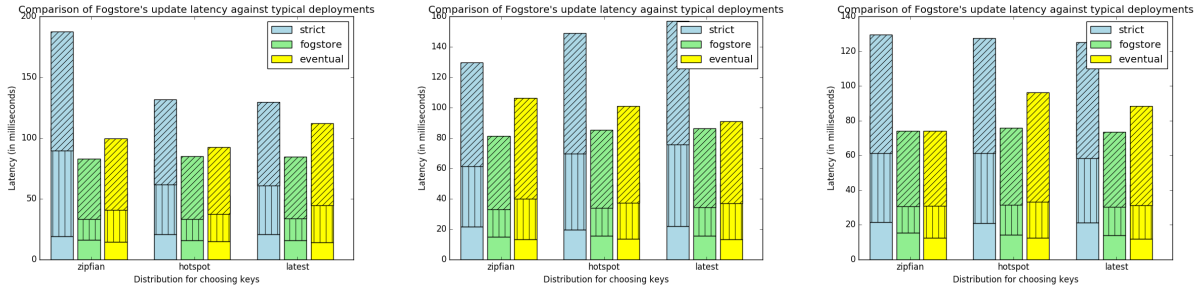
This reduction in programming effort due to strong consistency guarantees from the datastore comes at a price on performance, as now each operation has to wait to complete on a quorum of nodes, which may have high network latency between them. This hypothesis is validated by the variation of read and update latencies shown in Figures 7 and 8 and the aggregate operation throughput shown in Figure 9. The applications in this paper's context are heavily dependent on the low-latency execution of read and update operations, and hence guaranteeing performance of paramount importance.

For the sake of performance, the applications at hand would use an eventually consistent datastore, and wait for each operation to complete on only one replica before acknowledging the user.



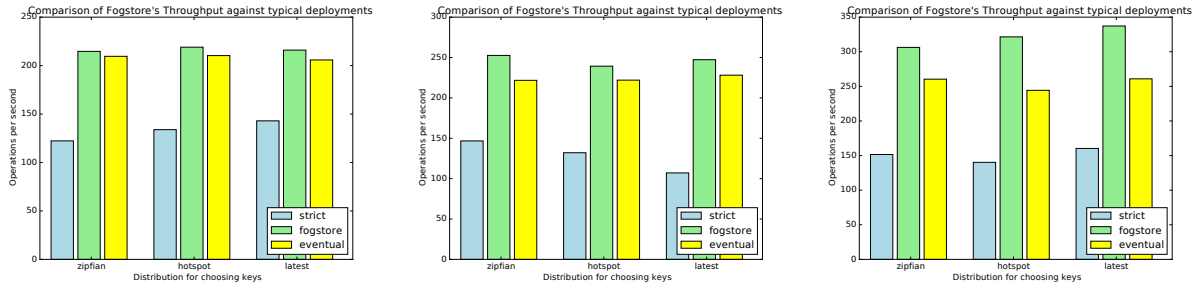
(a) 20% reads **(b) 50% reads** **(c) 80% reads**

Figure 7: Read latencies for varying distribution of reads. The solid, vertically dashed and obliquely dashed portions of the bar denote 50^{th}, 95^{th} and 99^{th} percentiles respectively.



(a) 20% reads **(b) 50% reads** **(c) 80% reads**

Figure 8: Update latencies for varying distribution of reads. The remaining operations are updates. The solid, vertically dashed and obliquely dashed portions of the bar denote 50^{th}, 95^{th} and 99^{th} percentiles respectively.



(a) 20% reads **(b) 50% reads** **(c) 80% reads**

Figure 9: Throughput (ops/sec) for varying distribution of reads

The experimental results show that the eventually consistent datastore is able to outperform the quorum-based store significantly in terms of operation latencies and throughput (Figures 7, 8 and 9). However, since clients may initiate reads before previous updates to those data-items have been propagated to all replicas, there are mismatches between the version returned by read operations and the most recently written version. Table 1 shows the percentage of reads that undergo such consistency violations. Note that due to the limitations of the emulation platform, we only have 16 client threads in these experiments, and increasing the concurrency would lead to more reads that violate strong consistency. Hence the improvement in performance comes at the cost of programming effort to handle inconsistent reads from the datastore.

To counter the performance limitations of quorum-based and consistency violations of an eventually consistent datastore, we run the same client workloads against Fogstore, which is fundamentally Cassandra extended with the context-aware policies for replication and quorum selection. Since the read and write quorums are limited

to replicas placed close to the data-item and overlap, there are no consistency violations i.e., it offers strong consistency. On the performance side, we observe that read and update latencies are even better than that of the eventually consistent store. We reason that this is because of the location-agnostic replica placement in plain Cassandra, where all the 3 replicas of a data-item may be located on remote nodes, leading to high operation latency even with eventual consistency. Fogstore is, therefore, able to achieve a better throughput than the eventual store. In fact, since the read quorum is set to 1 and write quorum to 2, the throughput for a read-heavy workload beats the eventual datastore by a higher margin as the number of replicas to block for is just one.

The context-aware optimizations of Fogstore allow it to obtain the performance of an eventually-consistent store and, at the same time, provide consistency guarantees similar to a quorum-based database. This enables the design of systems that are dependent both on high throughput and low programming effort of dealing with inconsistent operation results.

6.2 Performance of range queries

Applications processing spatio-temporal data have a high performance dependence on efficiency of range-queries. Fogstore performs data-distribution based on location, which would lead range queries to span across a number of datastore nodes. In the following set of experiments we analyze the performance of Fogstore for delivering results of range queries and the impact of range filter size. We compare the performance of range queries on Fogstore against a baseline eventually consistent database setup - which uses the data-item's *type* to partition items across datastore nodes. Since events are not mutable, we are not dependent on consistency guarantees.

For this paper, we assume that range queries request events of a certain type T in a circular geographical area of radius R km centered at location L and can be expressed as the tuple (L, R, T). To support efficient range queries, we built a wrapper that transforms the tuple (L, R, T) into a number of contiguous subset of geohashes, such that they cover the area covered by the range (similar to [6]). For each of these subsets we trigger a *child* concurrent sub-query to Fogstore that returns events with a location falling under the subset of geohashes. The range query is said to be completed when all the *children* sub-queries are complete. For this set of experiments, we build a datastore cluster same as the previous experiment, with 4 nodes around Atlanta and 4 at remote locations (see Figure 6). A precision of 4 is chosen for encoding locations into Geohash, both for data-nodes' tokens and partition-key of data-items. We load the database with 4000 events of 10 different types from the area marked by the blue outline. For each range query (L, R, T), L is sampled from the bounding box in Figure 6 while T is sampled from the set of 10 event types. The value of R is varied and the impact of its variation on range query completion time is reported in Figure 10.

As is evident from Figure 10, the location-based distribution of data done by Fogstore is able to achieve comparable performance to the baseline eventually consistent datastore with the increase in radius of range queries. Increasing the range-query radius increases the number of replicas (datastore nodes) that would store the events

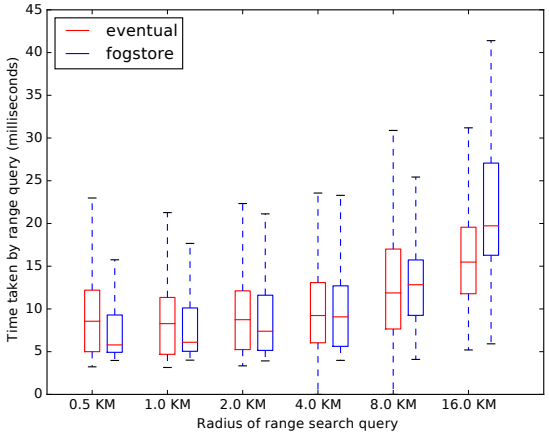


Figure 10: Comparison of latency of range queries with varying range-radius for Fogstore's data partitioning and eventually consistent Cassandra deployment. All statistics are aggregated over 5000 range queries.

queried for, which increases the overhead on Fogstore's coordinator node to split the original sub-query further down into read requests that are sent to individual replicas. This factor, however, does not impact the baseline datastore, since the events are partitioned based on the item type field, which is fixed for a particular range-query, meaning each replica assigned to that item type would have all the data-items of that type. Furthermore, an increase in range-query radius also leads to increase in the number of events returned, which also impacts query-completion time, both for Fogstore and the baseline.

6.3 Load balancing across data-store nodes

One of the supposed limitations of partitioning data-items based on spatial encoding using Cassandra's hash ring mechanism is that skews in application workload can lead to some datastore nodes storing significantly more data-items than others. This is because of the fact that we want to preserve spatial proximity of replica placement.

In order to perform large-scale tests of load balancing, we design a simulation environment which mimics the replica placement approach of Fogstore [6]. We focus on the region around Atlanta, as shown in Figure 6, and place 64 datastore nodes within that region in a uniformly-spaced manner. To simulate spatial skew in data access pattern, we generate 80% of the data-items from a small area (0.0625 times the full region) around Downtown while the rest of the region generates 20% of data-items. For every data-item we set the number of InCoI replicas to 2, and configure the CoI distance so as to have all the 64 fog nodes candidates for hosting the InCoI replicas. Furthermore, we also envision having multiple datastore nodes deployed at a particular resource location, akin to a mini-datacenter. The data-partitioning policy should be able to

[6]The emulation platform used in previous experiments is not large enough to emulate a very large number of nodes

utilize all the available capacity, both at the same location as well as across multiple locations. The metric of interest is the number of data-items that each node is assigned. A data partitioning policy that does not take load-balancing into account would lead to nodes close to the hotspot region storing much more data-items than those away from it, resulting in a high variance in the aforementioned metric. We report the measured metric in Figure 11.

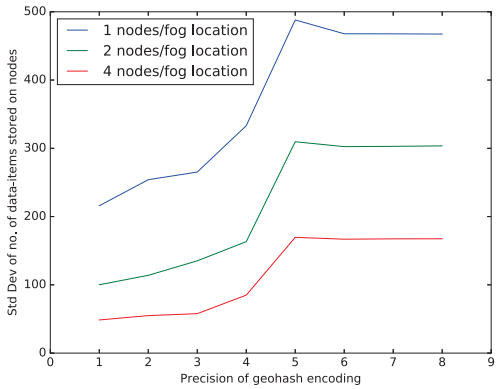


Figure 11: Standard deviation of the number of data-items stored by each datastore node (lower number denotes better load balancing). We vary the precision used for Geohash encoding as well as the number of datastore nodes on each resource location. The numbers are averaged over 10 simulation runs.

As described in Section 5, the precision of encoding a data-node's location for generating its token is crucial for determining the degree of spatial load balancing. A smaller value of precision leads to a large number of nodes having the same location-specific part of the token, leading to more widespread load balancing. A larger precision leads to higher spatial proximity and, hence, less widespread load balancing. We are able to see the above effect of Geohash encoding precision on the degree of load balancing (as show in Figure 11). Interestingly, a Geohash encoding precision of 5 and above encodes the location of all datastore nodes to be distinct, thus leading to no changes in load balancing metric for higher precision.

Furthermore, an increase in the number of datastore nodes on each resource location also improves the load balancing metric, as the data-partitioning policy is able to seamlessly distribute data-items across them as well. Hence with a proper precision of Geohash encoding for token formation and enough resources near regions with higher activity, we can obtain both spatial proximity of replicas and good load balancing.

6.4 Evaluation of fault-tolerance

Fogstore's data distribution policy takes tolerance to geographically correlated failures into account. Contemporary cloud-based databases distribute replicas across multiple datacenters, so that they are at a sufficiently high distance from each other to not be affected by correlated failures like earthquakes or massive power outages. Fogstore, however due to lack of physical constructs like datacenters and regions, performs wide-area geo-distribution using the location of data-items and data-store nodes, by guaranteeing a certain minimum distance in their spatial encodings. However, since spatial encodings tend to translate higher (two) dimensional attributes into one-dimension, high distance in terms of encoding of two locations does not consistently translate to high geographical (actual) distances between the two locations.

In the following set of experiments we examine the distribution of the geographical distance between Out-of-CoI replica nodes and the data-item. Since the In-CoI replicas are kept close to the data-item's location, placing the Out-of-CoI replicas significantly far away from the data-item's location would imply geographical separation between the In-CoI and Out-CoI replicas - thus achieving the aforementioned tolerance to correlated failures. We choose a few candidate large-scale resource topologies that are likely of being representative of how such geo-distributed infrastructures could be realized, which are described below :

(1) Capitals : Each capital of mainland USA's states is considered a central location of resource capacity, with a fixed number of fog nodes scattered around a certain radius (50 km) from the capital city's centre.
(2) AT&T : Locations of core routers of AT&T's backbone network inside the USA are considered central locations of resource capacity, with a fixed number of fog nodes scattered around a certain radius (50 km) from the peering points location.

Accesses are generated from within a radius of 100 km from each central resource capacity location (an example of which is shown in Figure 12). We note the location where the Out-CoI replicas

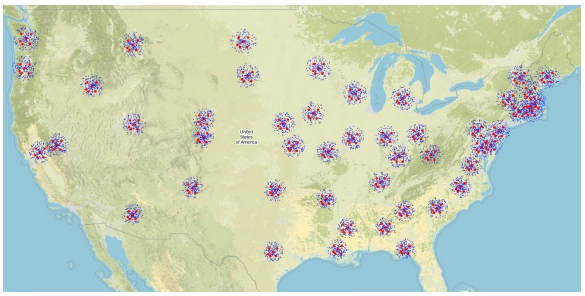


Figure 12: Geographical distribution of data-store resources (red dots) and accesses (blue dots) for a reference topology with USA state capitals as resource capacity locations.

are placed and the distance of that node from the location of the data-item for the aforementioned reference topologies, as shown in Figure 13. For both the candidate topologies, a maximum encoding distance threshold of 34 can allow a median separation of atleast 2500 KM, which is a significantly large distance for geographically-correlated failures, like natural disasters or massive power outages, to affect.

7 CONCLUSIONS AND FUTURE WORK

Through FogStore, we have presented the design of context-aware replica placement and quorum selection policies for a key-value

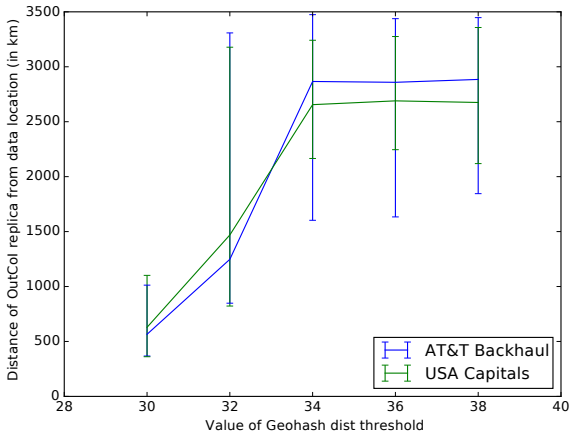


Figure 13: Distribution of distance between OutCol replicas and data-items' locations for various reference topologies. The precision of Geohash encoding for generating the token has been taken as 8.

store built for highly geo-distributed (edge computing) infrastructures. Replica placement is done taking the low-latency requirement of data-access into account, while also making the placement tolerant to geographically correlated failures. It, creates two types of replicas, ones which are located in proximity to the clients for low-latency and ones in remote location for fault-tolerance. The quorum selection policy leverages the fact that the context of clients determine the degree of consistency expected when querying the state of a situation-awareness application. Hence queries from clients in the vicinity of the queried item, which require consistent data access, have a majority of the proximal replicas in quorum, while those from remote clients are offered eventual consistency. We show the performance of the proposed policies by implementing them on Apache Cassandra and using YCSB to stress-test the system. Evaluations show that the proposed policies are able to achieve a throughput and latency comparable to eventually consistent systems, while still guaranteeing serializability guarantees on relevant data-items to clients.

As future work, we plan to extend the idea of contextual-relevance of data to include temporal-relevance as well, and use these concepts to design a holistic data management platform for geo-distributed Internet-of-Things data producers and consumers. Such a management platform should take into account the limited storage capacity on the geo-distributed resources. Furthermore, due to the wide geo-distribution and the large-scale of edge computing infrastructure, guaranteeing one-hop data-access by maintaining a Dynamo-style global hash-ring may lead to high network traffic [9]. The network traffic could be aggravated when the storage nodes show high churn. An avenue for exploration is trading network latency to some extent for a more scalable data distribution (*a la* peer-to-peer systems).

ACKNOWLEDGEMENTS

This work was funded in part by an NSF Award (NSF-CPS-1446801) and a grant from Microsoft Corp. We thank members of Georgia Tech's Embedded Pervasive Lab and the anonymous reviewers for helping to improve the presentation.

REFERENCES

[1] DataStax configuring data consistency in apache cassandra. https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_config_consistency_c.html. Accessed: 2018-02-19.
[2] Kafka connect. https://docs.confluent.io/current/connect/intro.html. Accessed: 2018-03-07.
[3] Lorenzo Affetti. Consistent stream processing: Doctoral symposium. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 355–358, New York, NY, USA, 2017. ACM.
[4] Austin Appleby. Murmurhash 2.0, 2008.
[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.
[6] Mohamed Ben Brahim, Wassim Drira, Fethi Filali, and Noureddine Hamdi. Spatial data extension for cassandra nosql database. *Journal of Big Data*, 3(1):11, 2016.
[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[8] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. On qos-aware scheduling of data stream applications over fog computing infrastructures. In *Computers and Communication (ISCC), 2015 IEEE Symposium on*, pages 271–276. IEEE, 2015.
[9] Bastien Confais, Adrien Lebre, and Benoît Parrein. Performance analysis of object store systems in a fog and edge computing infrastructure. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXIII*, pages 40–79. Springer, 2017.
[10] Rodrigo S Couto, Stefano Secci, Miguel Elias M Campista, and Luís Henrique MK Costa. Latency versus survivability in geo-distributed data center design. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 1102–1107. IEEE, 2014.
[11] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.
[12] Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu, and María S Pérez-Hernández. Exploring shared state in key-value store for window-based multi-pattern streaming analytics. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1044–1052. IEEE Press, 2017.
[13] Marlon McKenzie, Hua Fan, and Wojciech Golab. Fine-tuning the consistency-latency trade-off in quorum-replicated distributed storage systems. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1708–1717. IEEE, 2015.
[14] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.
[15] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4):20, 2017.
[16] Hans Sagan. HilbertâĂŹs space-filling curve. In *Space-filling curves*, pages 9–30. Springer, 1994.
[17] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269. ACM, 2016.
[18] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.
[19] Subhajit Sidhanta, Wojciech Golab, Supratik Mukhopadhyay, and Saikat Basu. Adaptable sla-aware consistency tuning for quorum-replicated datastores. *IEEE Transactions on Big Data*, 3(3):248–261, 2017.
[20] Yuuichi Teranishi, Ryohei Banno, and Toyokazu Akiyama. Scalable and locality-aware distributed topic-based pub/sub messaging for iot. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–7. IEEE, 2015.
[21] Philip Wette, Martin Draxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.
[22] Victor Zakhary, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Global-scale placement of transactional data stores. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT*, pages 26–29, 2018.