ARIADNE: A Resilient IoT Storage Service for Geospatial Computing

Vinayak Gajjewar, Chandra Krintz, Rich Wolski
Department of Computer Science,
University of California, Santa Barbara

***Abstract*—Geospatial applications deployed on edge infrastructures increasingly rely on distributed storage systems to provide resilience to frequent node failures. However, conventional replica placement strategies do not consider geospatial query access patterns, forcing queries to retrieve related data from multiple nodes and increasing network traffic, latency, and energy consumption. In this paper, we present ARIADNE, a distributed storage service that jointly optimizes replica placement for both fault tolerance and query locality. ARIADNE co-locates replicas of data that are likely to be accessed together while dynamically migrating frequently accessed ("hot") data to the most reliable nodes. Experimental results show that ARIADNE reduces network data transfer by up to 40% compared to existing replica placement algorithms while maintaining storage service-level agreements under realistic and frequent node failures. These results demonstrate that query-aware replica placement can substantially improve the efficiency of geospatial analytics on resource-constrained edge platforms.**

I. INTRODUCTION

The Internet of Things (IoT) produces vast streams of data that often must be indexed to, or correlated with, physical space. In these application contexts, each measurement, observation, or record typically corresponds to a specific location (specified by a coordinate system), forming a distributed geospatial dataset that reflects real-world dynamics. Applications that analyze and query such data (e.g., environmental monitoring, precision agriculture, transportation management, or urban analytics) rely on efficiently accessing and aggregating information within spatial regions. Supporting these workloads at the edge, close to where the data originates, is essential for achieving low latency and high availability. However, environments where the edge can become disconnected from reliable infrastructure (such as the cloud), or where machines fail frequently due to fluctuating power availability, are particularly challenging with respect to performance and reliability.

Importantly, the edge devices we consider are not microcontroller-class IoT nodes (e.g., sensor motes or deeply embedded devices), but rather small-form-factor edge computers, such as Raspberry Pi-class systems [1]. These devices provide persistent storage, multi-core CPUs, and full operating systems, enabling them to host distributed services and maintain local indexes, but they remain power- and memory-constrained, failure-prone, and intermittently connected in real-world deployments. This class of edge infrastructure is increasingly common in agricultural, environmental, and industrial settings, yet is poorly served by existing storage systems designed either for microcontrollers or for cloud-scale clusters.

Managing geospatial data in edge IoT deployments is challenging because data is distributed across these devices. Despite frequent device and network failures, the data must remain available for reliable query processing. Existing edge storage systems address this challenge through distributed data replication and fast indexing. However, they treat data as independent objects, ignoring the spatial structure that defines many IoT datasets [2], [3]. Systems that implement geospatial databases and spatial indexing [4], [5], [6] such as SpatialHadoop [7], GeoSpark [8], and GeoMesa [9] provide rich and efficient spatial query support but typically operate in the cloud, relying on abundant and highly reliable compute/memory resources, network connectivity, and centralized control.

This paper addresses the gap between efficient, scalable support for geospatial query workloads and the realities of edge computing environments. Specifically, we present ARIADNE, a distributed storage service for these environments that combines spatial awareness with robust data management. ARIADNE organizes data into spatial blocks that encode geographic coordinates, and places and replicates these blocks on edge devices based on their geospatial proximity and underlying device reliability. ARIADNE (a) co-locates data replicas that are likely to be accessed together on the same machines and (b) re-replicates "hot" data onto the most reliable machines. To enable this, we define a deployment architecture, consistency and synchronization protocols for data blocks, indexing, and node coordination, and a spatial partitioning mechanism for storage block placement and rebalancing.

We evaluate ARIADNE using an experimental cluster composed of Raspberry Pi-class edge nodes, designed to reflect the power, storage, and failure characteristics of real agricultural and industrial IoT deployments where infrastructure (i.e. power, networking, physical security, etc.) is weak or absent. We use the cluster to empirically compare ARIADNE to two recent distributed, IoT storage services. We evaluate system performance along three dimensions. First, we measure the network load generated by queries and the distributed placement of spatial workloads. Second, we assess each system's ability to maintain its query success SLA despite frequent, power-induced device failures. Finally, we evaluate system-wide storage efficiency.

We find that ARIADNE outperforms existing edge storage systems by reducing the amount of data transferred (by 40% overall for the systems and workloads we study) in this setting. Message load is a critical metric for edge systems because it

correlates with power usage (e.g. battery drain due to WiFi usage), intermittent connectivity, and the delay associated with network retries and timeouts due to node failure. By co-locating replicas of spatially related data blocks, ARIADNE increases the likelihood that a spatial query can be resolved efficiently even under high failure rates. Such replication aligns redundancy with both geographic proximity and device reliability, thereby lowering the cost of serving queries while maintaining high availability.

In summary, this paper makes the following contributions.

- We introduce ARIADNE, a spatially aware storage service for IoT deployments that operates entirely at the edge, supporting disconnected and failure-prone environments.
- We design and implement a mechanism that reduces the network load required to replicate and service spatial workloads on failure-prone and heterogeneous devices.
- We evaluate ARIADNE under realistic conditions using a large-scale IoT testbed and demonstrate that ARIADNE is able both improve message parsimony and meet stringent query success SLAs in the presence of frequent node failures.

II. Related Work

Existing approaches to spatial databases and distributed geospatial systems provide abstractions for location-aware data. Popular systems such as PostGIS, Oracle Spatial, and MongoDB GeoJSON offer expressive spatial queries and efficient index structures (R-trees, quadtrees, geohashes) [10], [11], [12]. Distributed frameworks such as GeoMesa, SpatialHadoop, and GeoSpark scale spatial query execution across large cloud clusters by partitioning and replicating spatial data for parallel processing [9], [7], [8], [13]. While these systems offer rich spatial abstractions, they assume reliable datacenter networks and abundant compute and storage resources which are conditions not found in all edge environments.

Locality-aware data placement and replication is also used in cloud and distributed systems to minimize latency or tolerate network failures. Systems including HDFS, Ceph, and Colossus employ zone- or rack-aware replication to reduce correlated failure risk [14], [15], [16]. Fog and edge frameworks such as iFogStor [17] and FogStore [18] extend these ideas to edge-cloud hierarchies, optimizing placement for bandwidth and latency rather than for fault resiliency or data semantics. Additionally, probabilistic graphical models such as dynamic Bayesian networks and Markov models [19], [20], [21] have been used to schedule edge computing tasks in a way that minimizes the risk of unavailability due to correlated failure.

ElfStore [2] and RIoTStore [3] are recent systems that prioritize device reliability in their replica placement decisions. These systems use a two-level coordination/storage model for metadata discovery, and place data block replicas on storage nodes according to anticipated device reliability and available storage capacity. These systems demonstrate that replication can provide durability and low-latency access in unreliable edge environments. However, their placement logic does not consider spatial locality and thus they cannot exploit geographic structure (as ARIADNE does) to improve query performance or messaging efficiency.

A related and complementary class of systems focuses on edge indexing and local data management assuming stable devices. EDIndex and related efforts propose distributed edge indexing mechanisms (e.g., R-tree or hierarchical hash structures) to accelerate local query execution (i.e. caching) and minimize dependence on the cloud [22], [23], [24], [25]. These systems improve performance for spatial queries but do not address device failures, replication, or disconnected operation making them unsuitable for fault-prone, intermittently connected IoT deployments.

Finally, FogStore ([26]) uses a space filling curve (i.e. a geohash) similar to ARIADNE's Hilbert curve partitioner to distribute data across devices. FogStore, however, requires a full Apache Cassandra deployment and does not support differential replication (as ARIADNE does). This design makes it ill-suited for edge environments where devices face strict resource limits and highly dynamic node reliabilities.

III. ARIADNE

ARIADNE is a distributed storage system that combines a novel data and query model, system architecture, consistency model, and replica placement strategy. This design exploits the spatial locality of geospatial query workloads to optimize network usage and failure resilience in edge environments. In this section, we describe the design and intuition behind each of these features.

A. Data and Query Model

ARIADNE represents each data item using a fixed-size data block with a unique, ordered pair of coordinates from a two-dimensional plane (i.e., latitude and longitude). This data model is similar to that of ElfStore [2] and RIoTStore [3]. For IoT, these blocks can contain sensor readings, metadata, or camera images, and include the coordinates associated with each block derive from the physical source location.

Each data block has a Service Level Agreement (SLA) that can be met by replicating that block on one or more devices. This SLA is specified as a number between 0.0 and 1.0 denoting the probability that the data block in question is available (i.e. a query for it succeeds) at any point in time. A data block is considered available at a particular point in time if at least one node on which it is replicated is up when queried. In the empirical evaluation of ARIADNE, we assign an SLA of 0.99 to each data block.

Each ARIADNE request from a client contains a spatial range query that contains a bounding box specified using the XY-coordinates of its top-left and bottom-right corners. ARIADNE returns the subset of data blocks whose XY-coordinates lie within the bounding box of that query. A graphical example of such a spatial range query is shown in Figure 1.

B. System Architecture

ARIADNE uses a two-tier system model composed of *routing nodes* and *storage nodes* to implement a logical separation

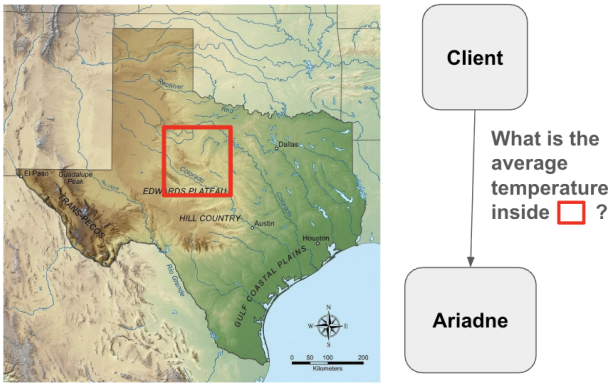

Fig. 1: Clients present queries to ARIADNE using a bounding box and an optional aggregate to be computed over the data in that spatial region. For example, a client may request the average temperature of a particular region in Texas. ARIADNE uses the XY-coordinates of the bounding box to identify data blocks that fulfill the query.

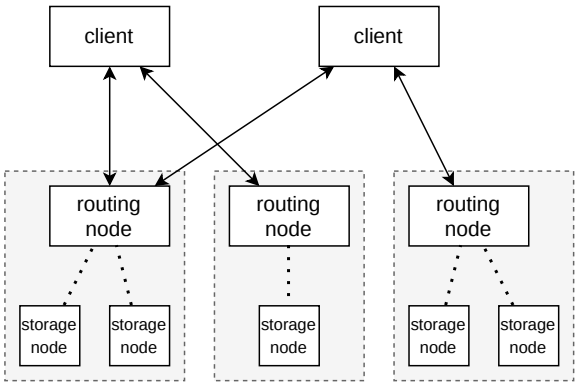

Fig. 2: In ARIADNE, nodes are assigned to either be a storage node or a routing node. Storage nodes are partitioned among routing nodes. Clients can present a spatial query to any available routing node. As a consequence of this architecture, if a routing node is unavailable, all of the storage nodes it manages are unavailable as well.

of control and storage responsibilities. Routing nodes are responsible for maintaining data location metadata, spatial indexing, and request routing, while storage nodes manage the underlying data blocks and serve read and write requests from routing nodes.

ARIADNE statically partitions and assigns storage nodes among the routing nodes. Each storage node relies on its routing node for indexing its locally stored data blocks, and for routing messages to and from clients and other nodes. Figure 2 illustrates this organization graphically.

Central to the design of ARIADNE is the assumption that clients may issue queries to any routing node in the system. Each routing node routes queries to the relevant nodes and handles node unavailability in a manner that is transparent to the client. Upon receiving a query, a routing node consults its local copy of the spatial index to identify the data blocks required to satisfy the query. It then routes requests to the storage nodes (or other routing nodes) that store those blocks. If a node is unavailable at the time that a request is made, the routing node in charge of fielding the client query, attempts to fetch the data blocks from a different node. If all of the nodes that store a relevant data block are unavailable, a response is sent back to the client indicating that the query could not be resolved. All message traffic related to data placement, replica management, and load rebalancing among storage nodes is similarly mediated by the routing nodes, simplifying coordination while limiting the responsibilities of storage nodes.

ARIADNE's hierarchical architecture creates potential single points of failure. If a routing node fails, all connected storage nodes become unavailable as a consequence. To formalize this, we define a *failure region* as the set of storage nodes sharing a single routing node. Section III-D describes how ARIADNE accounts for these correlated failures during replica placement.

C. Differential Replication

ARIADNE replicates data among nodes using an availability probability for each device in a deployment (for both routing nodes and storage nodes). This value is the probability that the device will be available at a given time. For example, a node with an availability of 0.95 is expected to respond to approximately 95 out of 100 requests received; approximately 5 requests are dropped due to an outage. In practice, this probability is calculated by measuring that node's uptime during an earlier window of time (e.g. previous 24 hours), and dividing the total uptime by the window size.

Node reliabilities in IoT deployments can vary significantly, even within the same cluster. To account for this, ARIADNE dynamically adjusts a data block's *replication factor* (the number of distinct storage nodes hosting its replicas) based on the block's query success SLA and the underlying availabilities of the selected nodes. Unlike systems such as Cassandra [27], which use a constant replication factor for all data blocks (designed to cover the worst-case reliability), ARIADNE computes a different replication factor for each data block group based on the reliability of the storage nodes used to store its replicas.

To compute the replication factor for a given data block, ARIADNE assumes that the availability probabilities of routing nodes are independent. Thus, the joint availability probabilities of storage nodes represented by *different* routing nodes are also assumed to be independent. When a data block is added to the system, ARIADNE chooses a candidate set of storage nodes S such that the joint availability probability among the storage nodes is greater than or equal to the target SLA. We define R to be the set of routing nodes that are connected to at least one storage node in S. We define S_r to be the subset of S that is connected to routing node r. Let a_r and a_s denote the availability probabilities of $r \in R$ and $s \in S$, respectively. Given a_r and a_s, ARIADNE computes the

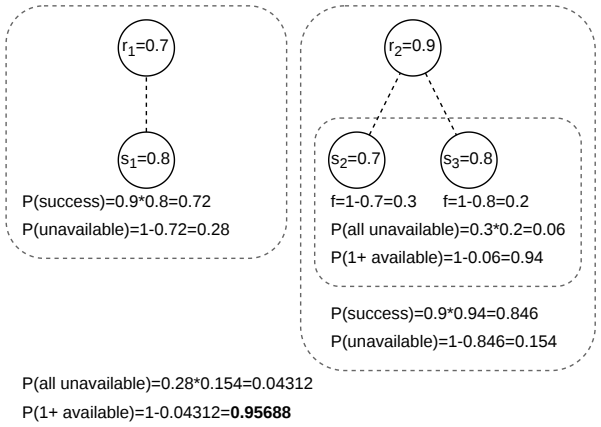

Fig. 3: The availability of a candidate replica set is calculated given the availabilities (f) of the relevant routing nodes and storage nodes. The proposed replica set is $\{s_1, s_2, s_3\}$, where s_1, s_2, and s_3 are storage nodes with s_1 assigned to routing node r_1 and s_2 and s_3 are assigned to r_2. With the availability values for each node given in the figure, the probability that at least one storage node is available for a given routing node is 0.8 and 0.94, respectively, for r_1 and r_2. Thus, the probability that at least one of r_1 and r_2 can service a request is 0.95688.

availability probability of the candidate replica set (a_S) using Equation 1. If a_S is greater than or equal to the block's query success SLA, the candidate replica set satisfies the SLA and the data block is replicated on the storage nodes in S. If a_S is less than the SLA, however, a different candidate replica set must be found. Figure 3 provides an example of computing the availability probability for a candidate replica set.

$$a_S = 1 - \prod_{r \in R} \left[1 - a_r \left(1 - \prod_{s \in S_r} (1 - a_s) \right) \right] \quad (1)$$

We term this process "differential replication" since each data block is replicated on a variable number of nodes depending on the availability characteristics of the selected nodes (and the block's query success SLA). The advantage of this approach is that it is more space efficient than a static approach that uses a constant replication factor when the availability probabilities associated with data nodes are highly variable (as they are in this study). That is, any constant replication factor that is sufficient to meet the query success SLAs using the most unreliable storage nodes is over replicating data on the more reliable nodes, compared to ARIADNE.

Note that this differential replication method assumes that the reliability of each node is known *a priori* and these reliability values do not change over the lifetime of the deployment. Generalizing this scheme to handle the case in which node reliabilities change over time is left to future work.

D. Replica Placement Strategy

ARIADNE's replica placement strategy selects a replica set from the set of candidates identified by differential replication. The goals of this selection process are three-fold:

1) Replicas of a single data block should be dispersed across failure regions.
2) Replicas of data blocks should be placed on the same devices as the data blocks spatially proximate to it.
3) Data blocks that are more likely to be queried for should tend to be replicated on devices with the highest reliabilities.

ARIADNE addresses the first two goals with its initial placement of replicas and the last goal with a replacement process that re-replicates hot data onto the most reliable storage nodes.

ARIADNE's placement strategy attempts to place replicas of a single data block across failure regions, meaning that replicas of a data block will tend to be placed on storage nodes that do not share a common routing node. By dispersing replicas across failure regions, ARIADNE decreases the likelihood that the failure of a single routing node renders all of the replicas of a data block inaccessible.

To maximize query efficiency, ARIADNE's placement strategy places replicas on storage nodes that already contain spatially proximate data. Preserving this spatial locality allows the system to fully leverage its geospatial distributed index during query execution.

ARIADNE optimizes system reliability by placing highly queried data blocks on the most reliable storage devices, thereby minimizing retries and network overhead caused by node unavailability. Because spatial density correlates with access frequency, where data blocks in dense map hotspots face higher query volumes than those in sparser regions, ARIADNE uses spatial density as a proxy for block popularity. Consequently, blocks from dense regions are preferentially replicated across high-reliability nodes. Finally, ARIADNE respects each storage node's maximum capacity, only placing replicas on nodes with sufficient available space.

Initial Placement: ARIADNE determines initial block placement using a consistent hashing variant where the hash function is the Hilbert index of a block's 2D coordinates. Storage nodes are arranged on a ring such that nodes from different failure regions are interleaved (Figure 4). To place a block, ARIADNE calculates its Hilbert index and assigns the first replica to the immediate successor node on the ring. Successive nodes along the ring are then selected until the block's availability satisfies its SLA. Because the Hilbert curve preserves 2D spatial locality, nearby data blocks map to adjacent locations on the ring; simultaneously, the interleaved node order guarantees that consecutive replicas are dispersed across distinct failure regions. Routing nodes are excluded from the hash ring.

Re-replication: The third goal of ARIADNE's replica placement strategy is to minimize fail-over overhead by assigning popular data blocks to the most reliable devices. As mentioned above, ARIADNE uses spatial density as a proxy for block popularity. Because dense map hotspots are not known *a priori*, ARIADNE dynamically re-replicates data blocks at runtime to adapt to evolving density patterns.

ARIADNE triggers re-replication for a node when its storage capacity is filled. This capacity-driven trigger naturally

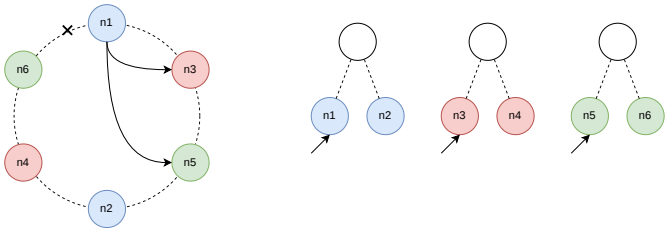

Fig. 4: Hash identifiers are assigned to storage nodes such that successive nodes on the hash ring are interleaved from different failure regions (dispersing replicas across distinct failure regions). Replicas are placed on successive nodes (following the Hilbert index x of the block) along the ring until the target SLA is met.

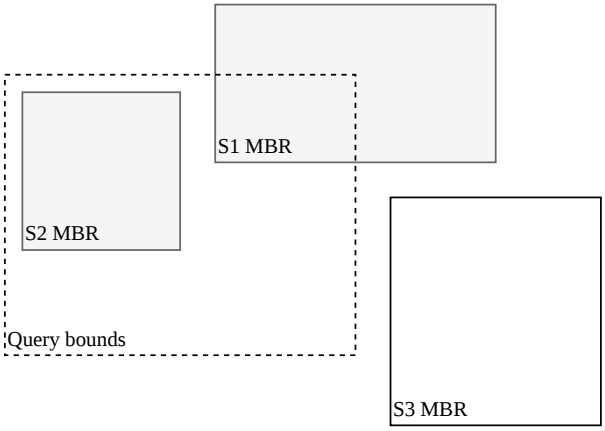

Fig. 5: ARIADNE's MBR index reduces network load for spatial range queries. S1, S2, and S3 are storage nodes with data blocks. Minimum bounding rectangles (MBRs) for the blocks are stored for each storage nodes. In this example, the query is pushed to S1 and S2 while S3 is skipped because its MBR does not intersect with the query. Note that a query can be pushed to a node that does not actually contain any relevant data.

prioritizes dense, high-demand regions. Because ARIADNE's placement policy colocates spatially proximate data, nodes assigned to hotspots exhaust their storage faster than those storing sparse areas, ensuring that "popular" blocks are re-replicated first.

When re-replication is triggered, all of the data blocks stored on that node are re-replicated onto other devices, i.e. those with sufficient storage capacity and high reliability values. If there are no other storage nodes with sufficient remaining capacity, re-replication fails and an error response is returned to the client application.

E. Data Indexing

ARIADNE can be configured to resolve queries using either a fine-grained full index or a coarse-grained Minimum Bounding Rectangle (MBR) index. The fine-grained index has the advantage of making queries more efficient to resolve, but the downside is that keeping the index consistent across all the routing nodes requires a greater network load. On the other hand, the MBR index is more coarse-grained, which means that it does not resolve queries as efficiently as the fine-grained index, but it requires less network load to keep the index consistent.

Regardless of which type of index is used, ARIADNE keeps the index globally consistent among routing nodes using the Scuttlebutt anti-entropy protocol. Every N milliseconds, where N is a configurable parameter of the system, each routing node contacts another, randomly-selected routing node and exchanges index information with it. After enough iterations, the state that each routing node maintains converges to a global value. Smaller values of N increase network load but decrease the time required for convergence, whereas larger values of N decrease network load but take more time to converge.

Minimum Bounding Rectangle (MBR) Index: When ARIADNE is configured to use the MBR index, the state that each routing node shares with its peers is the set of MBRs for the data blocks that each of its storage nodes stores, as shown in Figure 5. When a routing node receives a query from a client, it checks this state and forwards the query to the routing nodes whose storage nodes' MBRs overlap with the bounds of the query. Each storage node returns the data blocks that are contained by the bounds of the query. One consequence of this scheme is that multiple replicas of the same data block may be transmitted over the network if they are replicated on storage nodes belonging to different routing nodes. The advantage of this scheme is that the index is lightweight; each storage node's MBR can be uniquely identified by the coordinates of its top-left and bottom-right points.

Fine-grained Full Index: When ARIADNE is configured to use the fine-grained index, each routing node maintains the list of block IDs that each other routing node is in charge of, as well as their XY-coordinates. This is so that when a query comes in, each routing node has the ability to decompose that query into a set of requests for separate block IDs, which it can then forward to the relevant routing nodes. Only when a request for a data block fails does the routing node servicing the query contact that data block's backup replicas. If a node stores multiple data blocks relevant to a particular query, requests are batched to minimize communication overhead. The advantage of this method is that the amount of data transferred over the network while servicing a query is minimized because no duplicates of the same data block are transmitted. This efficiency comes at the cost of a greater network load required for keeping the index consistent because the fine-grained full index is much larger than the lightweight MBR index.

F. Data Consistency and Synchronization

To propagate block update operations to all replicas, as well as to handle the case where clients perform concurrent operations on the same data block, storage nodes that store replicas of the same data block synchronize their replicas through the anti-entropy protocol known as Scuttlebutt recon-

ciliation [28]. In our scheme, each distinct replica of a data block is associated with a version vector (a form of logical timestamp), and each routing node periodically selects another peer and exchanges version vectors for data blocks that they have in common. Version vectors are a mapping from replica IDs to logical timestamps, and thus the size of a replica's version vector is proportional to the replication factor of the data block. Version vectors can only provide a partial ordering between operations, so in the event that the version vectors for two replicas of a data block are logically concurrent, we use physical timestamp (e.g. coordinated using NTP [29]) as a tie breaker to determine which data block is "newer" and thus should overwrite the older one.

ARIADNE includes a parameter to specify the duration of the time between consistency messages, which we call the synchronization window. A larger synchronization window will lead to fewer messages being exchanged between routing nodes, but will result in updates being propagated more slowly through the system. Smaller synchronization windows will increase the speed with which updates are propagated, but require more messages to be exchanged between routing nodes. For this work, we assume that users will choose a synchronization window size that approximately matches the expected rate of updates to the system.

ARIADNE also uses Scuttlebutt to synchronize the distributed index (MBR or full index) across routing nodes. When a data block is replicated on a set of storage nodes, if the MBR index is used, the MBRs of those storage nodes may need to expand to include that new data block. When this occurs, the MBR index maintained by each routing node must be modified to reflect this change to ensure correctness. The full index, in contrast, must be updated whenever a new data block is replicated. ARIADNE also includes a tuning parameter that controls the frequency with which index synchronization messages are exchanged.

G. Implementation

To evaluate the efficacy of our design, we implemented a prototype of ARIADNE in C++. ARIADNE uses ZeroMQ [30] sockets for communication and the Protocol Buffers library [31] to serialize and deserialize messages. There are three main components of the ARIADNE codebase:

- **[ariadne_client]** End users interact with ARIADNE through the client library, which abstracts away the details of versioning and consistency.
- **[ariadne_routing_node]** An ARIADNE deployment requires there to be some number of routing nodes listening on predefined endpoints, which serve as an interface between clients and the storage nodes that replicate data blocks. For routing nodes, the endpoints of the storage nodes that it supervises and those of the other routing nodes are specified as command line arguments.
- **[ariadne_storage_node]** When an ARIADNE storage node joins, the endpoint of its routing node is specified as a command line argument.

Metric	Value
Mean	0.577
Median	0.567
SD	0.287

TABLE I: Statistical properties of real-world device availabilities as measured over a period of 24 hours. A device's availability is calculated as the fraction of time during which that device experienced no under-voltage faults.

IV. EXPERIMENTAL METHODOLOGY

We compare ARIADNE against two similar edge storage services: ElfStore and RIoTStore. We re-implement them from their respective publications and source code repositories [2], [3]. We picked these systems to compare against because they both also perform differential replication to meet block-based, query success SLAs on heterogeneous devices. We use the Jackpine spatial database benchmark [32] and the failure characteristics from a live IoT deployment to evaluate these systems under realistic conditions.

For each system, we measure SLA violations, inter-machine network load, and storage efficiency. Note that in an IoT context, the network load between machines is often highly correlated with energy usage (and therefore battery drain), so minimizing network load is a key deployment engineering consideration. Lastly, because IoT devices are failure prone, we measure how well each system is able to maintain a query success SLA of 0.99 in the presence of frequent device failures.

Routing nodes initiate a Scuttlebutt gossip at a fixed interval of 10ms to ensure that no stale data is returned to a query (for both ARIADNE and RIoTStore protocols). In order to reflect a resource-constrained deployment, we constrain each storage node's storage capacity to store a max of 1,600 data blocks, each of which is 2KB in size.

A. Experimental Apparatus

We deploy each system (ARIADNE, RIoTStore, and ElfStore) on a topology of 256 storage nodes which we divide evenly among 16 routing nodes. The devices are 4-core (1.4GHz), 1GB RAM, Raspberry Pi 3B+ single board computers [33] connected via a cascaded 1GB switched Ethernet network. This cluster is an experimental system designed to expose the feasibility of deploying low-power, large-scale computing in industrial or agricultural settings where power, physical shelter, and network infrastructure is weak or non-existent.

Each device shares a USB power distribution unit (PDU) with 41 other devices that is powered by a 120 VAC circuit. Because the inexpensive PDUs were not designed for ruggedized high-performance computing duty cycles, the machines often report under-voltage faults that cause unexpected CPU throttle events, and will "throttle cap" intermittently due to power fluctuations that could damage machine circuitry.

For the purposes of this study, we treat undervoltage events (measured using `vcgencmd`) that result in a CPU throttling as the onset of a failure duration. We treat the restoration of full CPU capability following a failure as the return of that machine

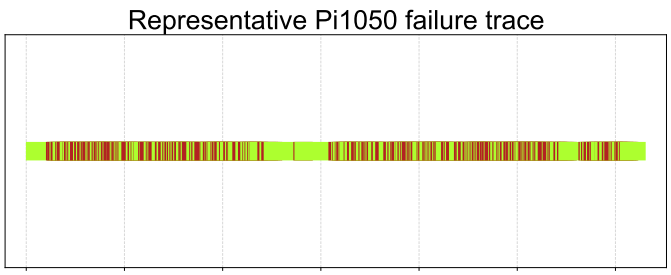

Fig. 6: A representative failure trace from a device in the Raspberry Pi cluster over a 24 hour period. Yellow regions denote uptime and red regions denote periods of unavailability due to undervoltage throttling.

to service. Table I shows the statistical summary of device availability (measured as a fraction of the 24-hour period) for all 272 devices, and Figure 6 depicts a representative failure trace from one of the devices over a 24-hour period. The mean device availability is 57.7%, signifying that, on average, a given device is available only 57.7% of the time. This failure trace dataset is more representative of failure-prone IoT deployments compared to others discussed the literature [34]. We refer to this experimental trace as the Pi1050 dataset.

To account for time-varying failure behavior at individual nodes, each experimental run uses voltage measurements from the preceding 24-hour period to estimate per-node availability. These measurements are collected at approximately one-minute intervals. For each node, we compute an availability probability as the fraction of the observation period during which the node does not experience an undervoltage condition. If an undervoltage event occurs on any node while serving a query, our experimental methodology determines that nodes cannot send or receive any messages to/from other nodes for the duration of the undervoltage event.

To ensure a fair comparison between ARIADNE, RIoTStore, and ElfStore, we record a dataset of failure traces collected over the course of 24 hours and replay the same set of failure traces for each system, to eliminate a source of potential discrepancy. This means that, for a single trial, we assign each node the same failure trace when evaluating ARIADNE, RIoTStore, and ElfStore. The failure traces are sampled without replacement from our dataset.

B. Spatial Application Workload Traces

To implement realistic data access patterns, we generate query traces using the Jackpine spatial database benchmark [32]. Jackpine models how end users and applications interact with spatial data, using spatial joins, analysis functions, and queries based on real-world spatial applications. The benchmark includes workload traces representing geocoding, map browsing, and land management, as well as flood risk and toxic spill analysis, among other analyses.

In this study, we use the US Census Bureau's TIGER [35] dataset for the state of Texas as a dataset against which Jackpine generates query traces. This dataset contains 13,441 points of interest (POI), each one represented by a point with a given latitude and longitude, as well as additional information about the location. In our experiments, we store each POI as a separate data block of size 2KB.

Statistic	ARIADNE	RIoTStore	ElfStore
mean	0.9946	0.9944	0.8575
median	0.9949	0.9945	0.8705
min	0.9900	0.9901	0.6507
max	0.9993	0.9989	0.9895

TABLE II: Query SLA for ARIADNE , RIoTStore, and ElfStore using a query success SLA of 0.99 per block. Both ARIADNE and RIoTStore are able to meet the SLA, while ElfStore does not. Note that for all three systems, the choice of index used does not impact query SLA.

A spatial range query from the Jackpine map-navigation benchmark consists of a bounding box having a randomly chosen boundary surrounding a POI followed by 5 randomly chosen transformations applied in sequence. Each transformation either shifts the box by 10KM in a random direction (preserving its size), halves the size of the bounding box (preserving its center point), or doubles the size of the bounding box (preserving its center point). The result of each initial bounding box selection and series of transformations results in a final bounding box that circumscribes one or more POIs (each represented as a data block). The data blocks containing the final POIs are sent to the client. Note that these queries exhibit strong spatial locality, meaning that POIs that are geographically close together have a higher probability of being accessed in the same query.

V. EVALUATION

Through a quantitative analysis of the performance characteristics of ARIADNE, we endeavor to answer the following questions.

- Is ARIADNE able to meet query success SLAs in the face of frequent failures?
- Does ARIADNE reduce the amount of data transferred between nodes while servicing geospatial workloads?
- Does ARIADNE reduce the number of messages transferred between nodes while servicing geospatial workloads?
- Under what conditions does ARIADNE reduce the networking overhead required to service geospatial workloads?

A. Meeting Data Service Level Agreements (SLAs)

First, we investigate whether the differential replication mechanisms in the three competitive approaches (ARIADNE, RIoTStore, and ElfStore) meet stringent query success SLA target (0.99) on failure-prone devices. To quantify the actual availability of a data block, we divide the fraction of times at least one replica was available divided by the number of times it was requested by a client. If the SLA is met, we expect to see that this fraction at or above the block's query success SLA value (0.99).

Table II shows the results of this analysis. ARIADNE and RIoTStore meet SLA target across statistics while ElfStore does not. The ElfStore mean availability is 0.8575 with a minimum of 0.6507 (both below the SLA target). ElfStore misses the SLA because it under-replicates blocks as a result of its assumption that routing nodes never fail and there are no correlated failures between routing and storage nodes.

We also find that the type of index used (which we evaluate further in Section III-E) does not impact the ability of ARIADNE or RIoTStore meet the SLA. In contrast, with either index type, ElfStore fails to meet the target SLA. Because of this, we exclude ElfStore from subsequent evaluations to focus on the comparison of ARIADNE and RIoTStore. Importantly, this result establishes that ARIADNE's differential replication mechanism is able to meet stringent SLAs on failure-prone devices.

B. Network Load for Different Workloads

We next evaluate the impact of different differential replication strategies on network load. As detailed above, the RIoTStore strategy attempts to optimize storage use (while meeting an SLA) while ARIADNE optimizes for spatial locality in its query operations (while meeting an SLA). We define network load as the total number of bytes transferred among all nodes across the deployment for data placement and spatial query operations. Network load reflects the cost of the messages employed for block placement, query resolution, index updating, and (in the case of ARIADNE only) re-replication.

We compare ARIADNE and RIoTStore using different workload mixes. Within a workload, block placement operations are "writes" and spatial Jackpine queries are "reads". The write fraction is the number of writes over the total number of operations (reads + writes). For each workload mix, we keep the total number of operations constant at 1 million, and vary the write fraction. For example, at 5%, we perform 50,000 writes (block placements) and 950,000 reads (Jackpine queries). As mentioned above, we omit ElfStore from further comparison because it fails to meet query success SLAs.

We first evaluate the impact of using the full index with each replication strategy. Figure 7 shows the total number of gigabytes (GB) transferred for 1 million operations for increasing write fractions, when ARIADNE (blue) and RIoTStore (orange) use the full index. The graph shows that, regardless of workload mix, both systems transfer a similar number of total bytes end-to-end. For example, with a write fraction of 0.05, ARIADNE with the full index sends an average of 659 GB and RIoTStore with a full index sends an average of 648.67 GB.

Using a full index, only a single data block must be retrieved. Since block transfer dominates, the choice of replica placement strategy has a negligible impact on total bytes transferred. Moreover, increasing the write fraction (placements versus queries) increases the end-to-end bytes transferred. This trend occurs for the full index because writes are costlier than reads. That is, for each block placed, the routing nodes must disseminate the entire list of block IDs to other nodes. With

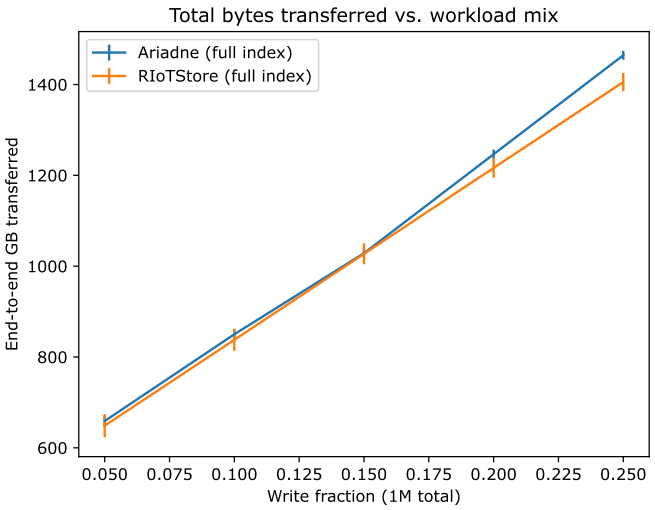

Fig. 7: Total bytes transferred for ARIADNE and RIoTStore with a full index. Regardless of workload mix, ARIADNE and RIoTStore utilize the network similarly when using a full index.

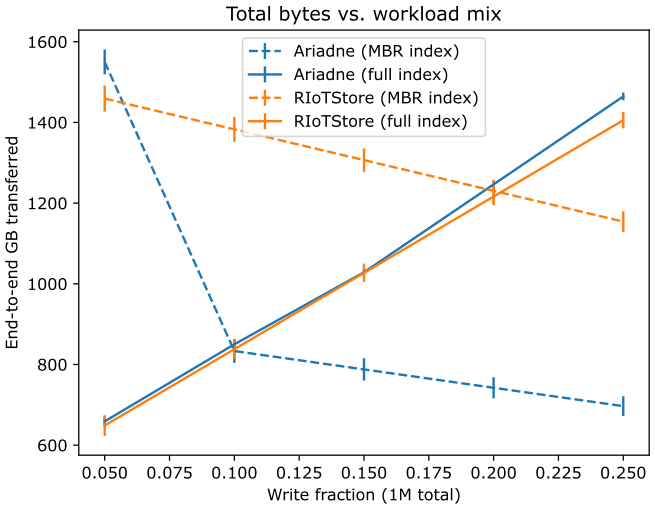

Fig. 8: Total bytes transferred for different workload mixes for ARIADNE and RIoTStore. MBR indices are dashed and full indices are solid lines. When the fraction of write operations is below 0.11, both ARIADNE and RIoTStore with an MBR index minimize total bytes transferred. When the fraction of writes is above 0.11, only ARIADNE with the MBR index minimizes bytes transferred.

the full index, ARIADNE and RIoTStore are equivalent from the point of view of total bytes transferred. We can also conclude that ARIADNE's re-replication mechanism does not significantly increase the amount of data transferred.

We next evaluate the impact of using a coarse-grained MBR index. We use the same experimental setup as above for the full index (1 million operations and increasing write fractions). Figure 8 shows the same data from Figure 7, with dashed lines

added. The dashed lines represent use of the coarse-grained MBR index for ARIADNE (blue) and RIoTStore (orange).

This result shows that there is a "crossover point" at a write fraction of approximately 0.11. More precisely, the crossover point is the write fraction w at which the total bytes transferred for the MBR index (B_{MBR}) and the full index (B_{Full}) are equivalent. Let q represent the average bytes required to service a query and p represent the average bytes required to place a block and update the index. The total bytes transferred can be expressed as:

$$B_{\text{total}} = q(1-w) + pw \tag{2}$$

where $w \in [0, 1]$ represents the fraction of total operations that are writes (block placements). By setting $B_{\text{MBR}} = B_{\text{Full}}$, we can analytically determine the critical write fraction w^* that identifies the workload mix where the indexing mechanisms reach parity in terms of data efficiency.

Below the crossover write fraction of 0.11, the full index (ARIADNE or RIoTStore) results in the fewest bytes transferred end-to-end. Above this fraction, ARIADNE with the MBR index transfers the fewest bytes. For a write fraction of 0.15, for example, using the MBR index, ARIADNE sends an average of 787.79 GB and RIoTStore sends an average of 1306.65 GB. That is, ARIADNE reduces network load for this workload mix by over 500 GB or 39.7%.

As write fraction increases, the network load for MBR index use decreases monotonically. This indicates that block placement operations are less expensive than queries using the MBR index. This is in contrast to the full index which shows an increase in network load as the write fraction increases. For the full index, each routing node is updated when a write occurs, whereas for the MBR index, such updates only occur when the MBR for a storage node expands.

Note that at a workload mix of 0.05, ARIADNE with the MBR index exhibits high network load compared to RIoTStore. In this case ARIADNE's re-replication mechanism is never triggered (so placement is never optimized for spatial locality using workload behavior). In this case, all queries are resolved using the initial (unoptimized) placement.

From these results, we can conclude that ARIADNE with the MBR index minimizes bytes transferred for workload mixes above a write fraction of 0.11. For write fractions below this value, use of the full index minimizes bytes transferred. Note that this "threshold" value of 0.11 is dependent upon both the characteristics of the dataset and the workload being serviced by the system. Accordingly, as part of future work, we are investigating extensions to ARIADNE that permit the system to intelligently switch between index types as workload characteristics change.

C. Messages Transferred For Different Workloads

We have established that ARIADNE reduces total bytes transferred, but in some IoT regimes, the total number of messages transmitted may also be important (such as when devices use significant energy for radio communication). We thus also report the total number of messages transferred between nodes of the system. The experimental setup for these results is the same as for the previous figures. We perform a total of 1,000,000 operations with increasing write fractions and measure the total messages transferred among all of the nodes of the system. Figure 9 shows the total number of messages transferred (1×10^8) for these operations as write fraction increases. We show message count for ARIADNE in blue and for RIoTStore in orange. We use dashed lines for the MBR index and solid lines for the full index.

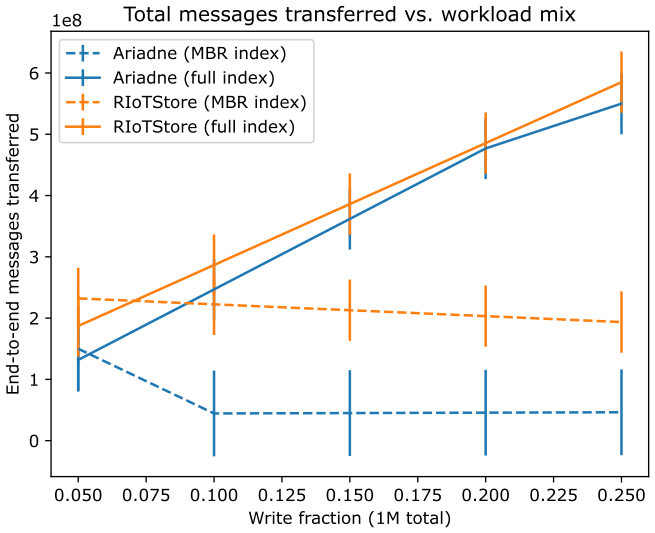

Fig. 9: Total messages transferred. for different workload mixes for ARIADNE and RIoTStore. MBR indices are dashed and full indices are solid lines. When the fraction of write operations performed is above 0.09, ARIADNE with the MBR index minimizes total messages transferred.

For different workload mixes, ARIADNE using the MBR index minimizes the number of messages sent versus RIoTStore using either index. Below a write fraction of 0.05, ARIADNE, using the full index performs slightly better than ARIADNE using the MBR index for the reasons outlined above (no rebalancing is triggered). As in the earlier results, both ARIADNE and RIoTStore with the MBR index lines trend downward, indicating that write operations (block placements) are less expensive than reads (spatial queries) for the MBR index (because it avoids full index updates). Note also that ARIADNE with the full index transfers fewer messages than RIoTStore with the full index. This is because ARIADNE's spatial locality reduces the average number of devices that must be contacted to resolve a query. This result shows that ARIADNE reduces messages sent while maintaining the SLA, which is useful in some IoT contexts.

D. Storage Efficiency

Next, we evaluate ARIADNE's storage efficiency. ARIADNE achieves reduced network load and message count versus RIoTStore, by trading off additional storage space. Thus, we

System	Total storage mean (MB)	Total storage SD (MB)
ARIADNE	808.75	17.24
RIoTStore	722.01	10.84

TABLE III: Total storage capacity in MB (mean and standard deviation) consumed across all nodes required to meet the 0.99 query success SLA for ARIADNE and RIoTStore. We present the mean and standard deviation of this value over 30 trials. The data shows that, due to ARIADNE's spatial locality-preserving replica placement, ARIADNE requires 12% more storage capacity across all nodes than RIoTStore to meet the target SLA.

Write Frac.	Re-replicate (GB)	No re-replicate (GB)	% Increase
0.05	1537	1537	0%
0.10	833	1456	74.8%
0.15	788	1376	74.6%
0.20	742	1296	74.7%
0.25	697	1216	74.5%

TABLE IV: ARIADNE network load (in GB on average) for multiple randomized initial placements with and without re-replication using different workload mixes (write fractions) using the Jackpine benchmark. The data shows that re-replication is necessary for minimizing end-to-end bytes transferred.

report the total storage space consumed by replicated data blocks across storage nodes in Table III. For this experiment, we place 86,400 data blocks using each system. The table shows average and standard deviation across 30 trials, each using a different set of failure traces.

For these experiments, ARIADNE requires 12% more storage space on average than RIoTStore. ARIADNE requires 808.75 MB (with a standard deviation of 17 MB), on average, and RIoTStore requires 722.01 MB (with a standard deviation of 10 MB) on average. RIoTStore optimizes storage capacity by placing data on the most reliable nodes first. This minimizes total replication size at the cost of spatial locality lost (which manifests as message count and load). This effect is also shown in the replication factor which is the number of replicas placed per block on average by each strategy. The median replication factor across blocks for ARIADNE in this study is 5.0, while for RIoTStore it is 4.0.

ARIADNE attempts to balance storage use with spatial locality. To do so, ARIADNE in some cases must place data on less-reliable nodes which increases the replication factor and total amount of storage used. However, as the results in the earlier subsections show, ARIADNE trades off this cost in additional storage with an overall reduction in network load and messages transmitted for geospatial queries for the benchmarks and deployments we consider.

E. Ablation Study

Next, we perform an ablation study to quantify the contribution of ARIADNE's key design components. Specifically, we disable individual optimizations and measure network load. Table IV reports the average network load in GB for different workload mixes (write fractions) for ARIADNE (using the MBR index) with and without re-replication. For write fractions of 0.1 and higher, disabling re-replication significantly increases the total bytes transferred because the system cannot correct sub-optimal replica placements. Below 0.1, ARIADNE does not trigger re-replication and thus the network load for both configurations is the same. ARIADNE re-replication reduces network load by 74.8% for the 0.1 write fraction.

	ARIADNE	RIoTStore	ElfStore
MB per query (Jackpine)	1.69	4.18	4.09
SD	0.52	0.05	0.14
MB per query (random)	4.27	4.00	4.12
SD	0.05	0.19	0.07

TABLE V: Average bytes transferred for both Jackpine queries (strong spatial locality) and randomly-generated queries that do not exhibit spatial locality. Each system is configured to use the lightweight MBR index to service queries. The data shows that when the workload does not exhibit spatial locality, the network efficiency advantage that ARIADNE has over RIoTStore and ElfStore vanishes.

F. Sensitivity Study

To evaluate ARIADNE's sensitivity to spatial locality, we modify the Jackpine benchmark to replace its bounding-box queries with uniformly random block requests. Doing so removes all spatial access patterns from the workload. We then execute 1,000,000 queries and measure the average network load required to service each query. As shown in Table V, using the original Jackpine workload, ARIADNE incurs the lowest network load, averaging 1.69 MB per query compared to 4.18 MB for RIoTStore and 4.09 MB for ElfStore. When spatial locality is removed, however, ARIADNE's network load increases to 4.27 MB per query, exceeding both baselines, while the performance of RIoTStore and ElfStore remains largely unchanged. These results demonstrate that ARIADNE's replica placement strategy derives its benefit from exploiting spatial locality. When that assumption no longer holds, its advantage disappears, highlighting an opportunity for future work on workloads with more diverse access patterns.

VI. CONCLUSIONS

In this paper, we present ARIADNE, a distributed storage service optimized for complex geospatial workloads on edge devices. ARIADNE uses Hilbert indices to co-locate replicas of spatially-proximate data blocks and re-replicates dense data blocks onto the most reliable devices in order to make geospatial access patterns as network-efficient as possible while also maintaining stringent SLAs on failure-prone devices. We evaluate ARIADNE against the Jackpine spatial database benchmark and the Pi1050 failure trace dataset against a set of representative baseline systems. Our evaluation shows that ARIADNE reduces bytes transferred by as much as 40% while maintaining stringent SLAs on failure-prone storage nodes.

We thank the IC2E reviewers for their constructive feedback. This work is supported in part by an Oracle gift, NSF

awards CNS-2444318 and CNS-2107101, and DoE award DE-SC0025541. The ARIADNE source code repository is available at https://github.com/MAYHEM-Lab/ariadne.

REFERENCES

[1] "Rasberry Pi," https://www.raspberrypi.org, [Online; accessed 7-Feb-2026].

[2] S. K. Monga, S. K. Ramachandra, and Y. Simmhan, "Elfstore: A resilient data storage service for federated edge and fog resources," in *IEEE International conference on web services*, 2019.

[3] V. Gajjewar, C. Krintz, and R. Wolski, "RIoTstore: Resilient Data Storage for Spatial IoT Applications," in *IEEE Annual Congress on Artificial Intelligence of Things (AIoT)*, 2025.

[4] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, and R. Montanari, "Locality-preserving spatial partitioning for geo big data analytics in main memory frameworks," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 2020, pp. 1–6.

[5] M. Werner, "Spatial data locality in scalable and fault-tolerant distributed spatial computing systems," in *Proceedings of the 7th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '18)*, 2018, pp. 47–56, emphasizes preserving spatial data locality to reduce network shuffling and to support fault-tolerant spatial workloads.

[6] R. Simmonds, P. Watson, and J. Halliday, "Antares: A scalable, real-time, fault tolerant data store for spatial analysis," in *2015 IEEE World Congress on Services*, 2015.

[7] A. Eldawy and M. Mokbel, "Spatialhadoop: A mapreduce-framework for spatial data," in *IEEE International Conference on Data Engineering (ICDE '15)*, 2015, pp. 1352–1363.

[8] S. Yu *et al.*, "Geospark: A cluster computing framework for processing large-scale spatial data," in *ACM SIGSPATIAL 2015*, 2015.

[9] C. Hughes *et al.*, "Geomesa: A distributed spatial database," in *Big Data 2015*, 2015.

[10] P. Ramsey, "The many spatial indexes of postgis," Blog post, Crunchy Data, May 2021. [Online]. Available: https://www.crunchydata.com/blog/the-many-spatial-indexes-of-postgis

[11] O. Corporation, "Indexing and querying spatial data," Oracle Corporation, Tech. Rep., 2000. [Online]. Available: https://docs.oracle.com/html/A88805_01/sdo_inde.htm

[12] M. M. Alam *et al.*, "A survey on spatio-temporal data analytics systems," *arXiv preprint arXiv:2103.09883*, 2021. [Online]. Available: https://arxiv.org/pdf/2103.09883

[13] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 1071–1085.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[15] S. A. Weil and other authors, "Ceph: A scalable, high-performance distributed file system," *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 307–320, 2006.

[16] D. Hildebrand and D. Serenyi, "Colossus under the hood: a peek into google's scalable storage system," https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[17] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "ifogstor: An iot data placement strategy for fog infrastructure," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, 2017.

[18] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, 2018. [Online]. Available: https://doi.org/10.1145/3210284.3210297

[19] A. Aral and I. Brandić, "Learning spatiotemporal failure dependencies for resilient edge computing services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1578–1590, 2020.

[20] S. Pallewatta, V. Kostakos, and R. Buyya, "Reliability-aware proactive placement of microservices-based iot applications in fog computing environments," *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 11 326–11 341, 2024.

[21] J. Liang, B. Ma, Z. Feng, and J. Huang, "Reliability-aware task processing and offloading for data-intensive applications in edge computing," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4668–4680, 2023.

[22] Q. He, S. Tan, F. Chen, X. Xu, L. Qi, X. Hei, H. Jin, and Y. Yang, "Edindex: Enabling fast data queries in edge storage systems," in *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2023.

[23] T. C. Rese, A. Kapp, and D. Bermbach, "Evaluating the impact of spatial features of mobility data and index choice on database performance," in *2025 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2025, pp. 1–12.

[24] M. Malensek, S. L. Pallickara, and S. Pallickara, "Expressive query support for multidimensional data in distributed hash tables," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*. IEEE, 2012, pp. 31–38.

[25] T. Hara, N. Murakami, and S. Nishio, "Replica allocation for correlated data items in ad hoc sensor networks," *ACM Sigmod Record*, vol. 33, no. 1, pp. 38–43, 2004.

[26] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, 2018, pp. 148–159.

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

[28] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, "Efficient reconciliation and flow control for anti-entropy protocols," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, 2008.

[29] D. L. Mills, "Network time protocol (ntp)," Tech. Rep., 1985.

[30] "ZeroMQ - Distributed Messaging," 2024, "http://zeromq.org/" [Online; accessed 7-Feb-2026].

[31] "Protocol Buffers. Google's Data Interchange Format," "http://code.google.com/p/protobuf" [On-line accessed 7-Feb-2026].

[32] S. Ray, B. Simion, and A. D. Brown, "Jackpine: A benchmark to evaluate spatial database performance," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 1139–1150.

[33] R. Pi, "Raspberry pi model 3b+," 2024, https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/; Accessed Oct 21, 2024.

[34] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed," *Journal of High Performance Computing Applications*, vol. 20, no. 4, 2006.

[35] R. W. Marx, "The tiger system: automating the geographic structure of the united states census," *Government publications review*, vol. 13, no. 2, pp. 181–201, 1986.