Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service

Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog
Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam ,Tim Rath
Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, Akshat Vig
dynamodb-paper@amazon.com
Amazon Web Services

Abstract

Amazon DynamoDB is a NoSQL cloud database service that provides consistent performance at any scale. Hundreds of thousands of customers rely on DynamoDB for its fundamental properties: consistent performance, availability, durability, and a fully managed serverless experience. In 2021, during the 66-hour Amazon Prime Day shopping event, Amazon systems - including Alexa, the Amazon.com sites, and Amazon fulfillment centers, made trillions of API calls to DynamoDB, peaking at 89.2 million requests per second, while experiencing high availability with single-digit millisecond performance. Since the launch of DynamoDB in 2012, its design and implementation have evolved in response to our experiences operating it. The system has successfully dealt with issues related to fairness, traffic imbalance across partitions, monitoring, and automated system operations without impacting availability or performance. Reliability is essential, as even the slightest disruption can significantly impact customers. This paper presents our experience operating DynamoDB at a massive scale and how the architecture continues to evolve to meet the ever-increasing demands of customer workloads.

1 Introduction

Amazon DynamoDB is a NoSQL cloud database service that supports fast and predictable performance at any scale. DynamoDB is a foundational AWS service that serves hundreds of thousands of customers using a massive number of servers located in data centers around the world. DynamoDB powers multiple high-traffic Amazon properties and systems including Alexa, the Amazon.com sites, and all Amazon fulfillment centers. Moreover, many AWS services such as AWS Lambda, AWS Lake Formation, and Amazon SageMaker are built on DynamoDB, as well as hundreds of thousands of customer applications.

These applications and services have demanding operational requirements with respect to performance, reliability, durability, efficiency, and scale. The users of DynamoDB rely on its ability to serve requests with consistent low latency. For DynamoDB customers, consistent performance at any scale is often more important than median request service times because unexpectedly high latency requests can amplify through higher layers of applications that depend on DynamoDB and lead to a bad customer experience. The goal of the design of DynamoDB is to complete *all* requests with low single-digit millisecond latencies. In addition, the large and diverse set of customers who use DynamoDB rely on an ever-expanding feature set as shown in Figure 1. As DynamoDB has evolved over the last ten years, a key challenge has been adding features without impacting operational requirements. To benefit customers and application developers, DynamoDB uniquely integrates the following six fundamental system properties:

DynamoDB is a fully managed cloud service. Using the DynamoDB API, applications create tables and read and write data without regard for where those tables are stored or how they're managed. DynamoDB frees developers from the burden of patching software, managing hardware, configuring a distributed database cluster, and managing ongoing cluster operations. DynamoDB handles resource provisioning, automatically recovers from failures, encrypts data, manages software upgrades, performs backups, and accomplishes other tasks required of a fully-managed service.

DynamoDB employs a multi-tenant architecture. DynamoDB stores data from different customers on the same physical machines to ensure high utilization of resources, enabling us to pass the cost savings to our customers. Resource reservations, tight provisioning, and monitored usage provide isolation between the workloads of co-resident tables.

DynamoDB achieves boundless scale for tables. There are no predefined limits for the amount of data each table can store. Tables grow elastically to meet the demand of the customers' applications. DynamoDB is designed to scale the resources dedicated to a table from several servers to many thousands as needed. DynamoDB spreads an application's data across more servers as the amount of data storage and the demand for throughput requirements grow.

DynamoDB provides predictable performance. The simple

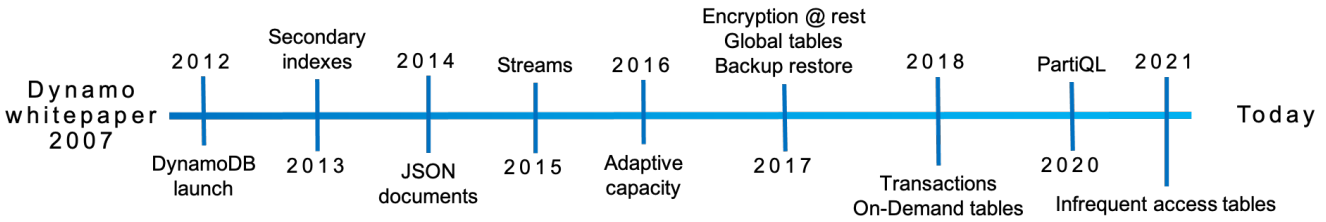

Figure 1: DynamoDB timeline

DynamoDB API with *GetItem* and *PutItem* operations allows it to respond to requests with consistent low latency. An application running in the same AWS Region as its data will typically see average service-side latencies in the low single-digit millisecond range for a 1 KB item. Most importantly, DynamoDB latencies are predictable. Even as tables grow from a few megabytes to hundreds of terabytes, latencies remain stable due to the distributed nature of data placement and request routing algorithms in DynamoDB. DynamoDB handles any level of traffic through horizontal scaling and automatically partitions and re-partitions data to meet an application's I/O performance requirements.

DynamoDB is highly available. DynamoDB replicates data across multiple data centers—called Availability Zones in AWS—and automatically re-replicates in the case of disk or node failures to meet stringent availability and durability requirements. Customers can also create global tables that are geo-replicated across selected Regions for disaster recovery and provide low latency access from anywhere. DynamoDB offers an availability SLA of 99.99 for regular tables and 99.999 for global tables (where DynamoDB replicates across tables across multiple AWS Regions).

DynamoDB supports flexible use cases. DynamoDB doesn't force developers into a particular data model or consistency model. DynamoDB tables don't have a fixed schema but instead allow each data item to contain any number of attributes with varying types, including multi-valued attributes. Tables use a key-value or document data model. Developers can request strong or eventual consistency when reading items from a table.

In this paper, we describe how DynamoDB evolved as a distributed database service to meet the needs of its customers without losing its key aspect of providing a single-tenant experience to every customer using a multi-tenant architecture. The paper explains the challenges faced by the system and how the service evolved to handle those challenges while connecting the required changes to a common theme of durability, availability, scalability, and predictable performance.

The paper captures the following lessons that we have learnt over the years

- Adapting to customers' traffic patterns to reshape the physical partitioning scheme of the database tables improves customer experience.
- Performing continuous verification of data-at-rest is a reliable way to protect against both hardware failures and software bugs in order to meet high durability goals.
- Maintaining high availability as a system evolves requires careful operational discipline and tooling. Mechanisms such as formal proofs of complex algorithms, game days (chaos and load tests), upgrade/downgrade tests, and deployment safety provides the freedom to safely adjust and experiment with the code without the fear of compromising correctness.
- Designing systems for predictability over absolute efficiency improves system stability. While components such as caches can improve performance, do not allow them to hide the work that would be performed in their absence, ensuring that the system is always provisioned to handle the unexpected.

The structure of this paper is as follows: Section 2 expands on the history of DynamoDB and explains its origins, which derive from the original Dynamo system. Section 3 presents the architectural overview of DynamoDB. Section 4 covers the journey of DynamoDB from provisioned to on-demand tables. Section 5 covers how DynamoDB ensures strong durability. Section 6 describes the availability challenges faced and how these challenges were handled. Section 7 provides some experimental results based on the Yahoo! Cloud Serving Benchmark (YCSB) benchmarks, and Section 8 concludes the paper.

2 History

The design of DynamoDB was motivated by our experiences with its predecessor Dynamo [9], which was the first NoSQL database system developed at Amazon. Dynamo was created in response to the need for a highly scalable, available, and durable key-value database for shopping cart data. In the early years, Amazon learned that providing applications with direct access to traditional enterprise database instances led to scaling bottlenecks such as connection management, interference between concurrent workloads, and operational problems with tasks such as schema upgrades. Thus, a service-oriented architecture was adopted to encapsulate an application's data

behind service-level APIs that allowed sufficient decoupling to address tasks like reconfiguration without having to disrupt clients.

High availability is a critical property of a database service as any downtime can impact customers that depend on the data. Another critical requirement for Dynamo was predictable performance so that applications could provide a consistent experience to their users. To achieve these goals, Amazon had to start from first principles when designing Dynamo. The adoption of Dynamo widened to serve several use cases within Amazon because it was the only database service that provided high reliability at scale. However, Dynamo still carried the operational complexity of self-managed large database systems. Dynamo was a single-tenant system and teams were responsible for managing their own Dynamo installations. Teams had to become experts on various parts of the database service and the resulting operational complexity became a barrier to adoption.

During this period, Amazon launched new services (notably Amazon S3 and Amazon SimpleDB) that focused on a managed and elastic experience in order to remove this operational burden. Amazon engineers preferred to use these services instead of managing their own systems like Dynamo, even though the functionality of Dynamo was often better aligned with their applications' needs. Managed elastic services freed developers from administrating databases and allowed them to focus on their applications.

The first database-as-a-service from Amazon was SimpleDB [1], a fully managed elastic NoSQL database service. SimpleDB provided multi-datacenter replication, high availability, and high durability without the need for customers to set up, configure, or patch their database. Like Dynamo, SimpleDB also had a very simple table interface with a restricted query set that served as a building block for many developers. While SimpleDB was successful and powered many applications, it had some limitations. One limitation was that tables had a small capacity in terms of storage (10GB) and of request throughput. Another limitation was the unpredictable query and write latencies, which stemmed from the fact that all table attributes were indexed, and the index needed to be updated with every write. These limitations created a new kind of operational burden for developers. They had to divide data between multiple tables to meet their application's storage and throughput requirements.

We realized that the goal of removing the limitations of SimpleDB and providing a scalable NoSQL database service with predictable performance could not be met with the SimpleDB APIs. We concluded that a better solution would combine the best parts of the original Dynamo design (incremental scalability and predictable high performance) with the best parts of SimpleDB (ease of administration of a cloud service, consistency, and a table-based data model that is richer than a pure key-value store). These architectural discussions culminated in Amazon DynamoDB, a public service launched in 2012 that shared most of the name of the previous Dynamo system but little of its architecture. Amazon DynamoDB was the result of everything we'd learned from building large-scale, non-relational databases for Amazon.com and has evolved based on our experiences building highly scalable and reliable cloud computing services at AWS.

Operation	Description
`PutItem`	Inserts a new item, or replaces an old item with a new item.
`UpdateItem`	Updates an existing item, or adds a new item to the table if it doesn't already exist.
`DeleteItem`	The DeleteItem operation deletes a single item from the table by the primary key.
`GetItem`	The GetItem operation returns a set of attributes for the item with the given primary key.

Table 1: DynamoDB CRUD APIs for items

3 Architecture

A DynamoDB table is a collection of items, and each item is a collection of attributes. Each item is uniquely identified by a primary key. The schema of the primary key is specified at the table creation time. The primary key schema contains a partition key or a partition and sort key (a composite primary key). The partition key's value is always used as an input to an internal hash function. The output from the hash function and the sort key value (if present) determines where the item will be stored. Multiple items can have the same partition key value in a table with a composite primary key. However, those items must have different sort key values.

DynamoDB also supports secondary indexes to provide enhanced querying capability. A table can have one or more secondary indexes. A secondary index allows querying the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB provides a simple interface to store or retrieve items from a table or an index. Table 1 contains the primary operations available to clients for reading and writing items in DynamoDB tables. Any operation that inserts, updates, or deletes an item can be specified with a condition that must be satisfied in order for the operation to succeed. DynamoDB supports ACID transactions enabling applications to update multiple items while ensuring atomicity, consistency, isolation, and durability (ACID) across items without compromising the scalability, availability, and performance characteristics of DynamoDB tables.

A DynamoDB table is divided into multiple partitions to handle the throughput and storage requirements of the table. Each partition of the table hosts a disjoint and contiguous part of the table's key-range. Each partition has multiple replicas distributed across different Availability Zones for high avail-

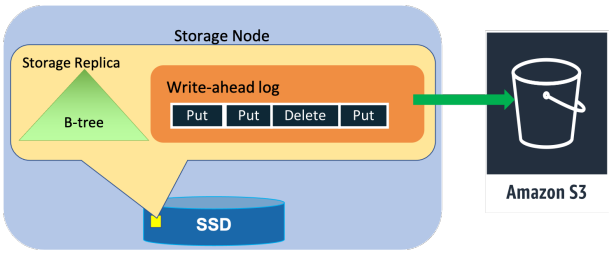

Figure 2: Storage replica on a storage node

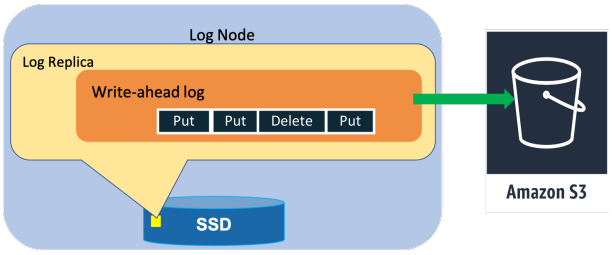

Figure 3: Log replica on a log node

ability and durability. The replicas for a partition form a replication group. The replication group uses Multi-Paxos [14] for leader election and consensus. Any replica can trigger a round of the election. Once elected leader, a replica can maintain leadership as long as it periodically renews its leadership lease.

Only the leader replica can serve write and strongly consistent read requests. Upon receiving a write request, the leader of the replication group for the key being written generates a write-ahead log record and sends it to its peer (replicas). A write is acknowledged to the application once a quorum of peers persists the log record to their local write-ahead logs. DynamoDB supports strongly and eventually consistent reads. Any replica of the replication group can serve eventually consistent reads. The leader of the group extends its leadership using a lease mechanism. If the leader of the group is failure detected (considered unhealthy or unavailable) by any of its peers, the peer can propose a new round of the election to elect itself as the new leader. The new leader won't serve any writes or consistent reads until the previous leader's lease expires.

A replication group consists of storage replicas that contain both the write-ahead logs and the B-tree that stores the key-value data as shown in Figure 2. To improve availability and durability, a replication group can also contain replicas that only persist recent write-ahead log entries as shown in Figure 3. These replicas are called `log replicas`. Log replicas are akin to acceptors in Paxos. Log replicas do not store key-value data. Section 5 and 6 discusses how `log replicas` help DynamoDB improve its availability and durability.

DynamoDB consists of tens of microservices. Some of the core services in DynamoDB are the metadata service, the

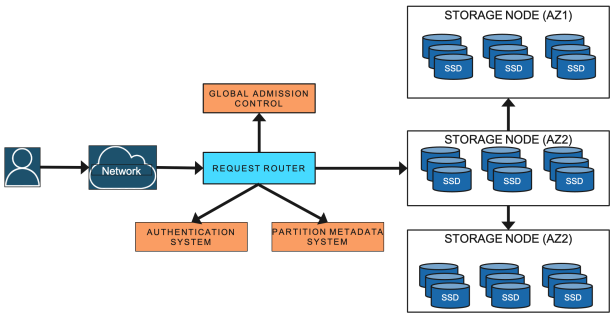

Figure 4: DynamoDB architecture

request routing service, the storage nodes, and the autoadmin service, as shown in Figure 4. The metadata service stores routing information about the tables, indexes, and replication groups for keys for a given table or index. The request routing service is responsible for authorizing, authenticating, and routing each request to the appropriate server. For example, all read and update requests are routed to the storage nodes hosting the customer data. The request routers look up the routing information from the metadata service. All resource creation, update, and data definition requests are routed to the autoadmin service. The storage service is responsible for storing customer data on a fleet of storage nodes. Each of the storage nodes hosts many replicas of different partitions.

The autoadmin service is built to be the central nervous system of DynamoDB. It is responsible for fleet health, partition health, scaling of tables, and execution of all control plane requests. The service continuously monitors the health of all the partitions and replaces any replicas deemed unhealthy (slow or not responsive or being hosted on bad hardware). The service also performs health checks of all core components of DynamoDB and replaces any hardware that is failing or has failed. For example, if the autoadmin service detects a storage node to be unhealthy, it kicks off a recovery process that replaces the replicas hosted on that node to bring the system back to a stable state.

Other DynamoDB services not shown in in Figure 4 support features such as point-in-time restore, on-demand backups, update streams, global admission control, global tables, global secondary indices, and transactions.

4 Journey from provisioned to on-demand

When DynamoDB launched, we introduced an internal abstraction, partitions, as a way to dynamically scale both the capacity and performance of tables. In the original DynamoDB release, customers explicitly specified the throughput that a table required in terms of read capacity units (RCUs) and write capacity units (WCUs). For items up to 4 KB in size, one RCU can perform one strongly consistent read request per second. For items up to 1 KB in size, one WCU can

perform one standard write request per second. RCUs and WCUs collectively are called provisioned throughput. The original system split a table into partitions that allow its contents to be spread across multiple storage nodes and mapped to both the available space and performance on those nodes. As the demands from a table changed (because it grew in size or because the load increased), partitions could be further split and migrated to allow the table to scale elastically. Partition abstraction proved to be really valuable and continues to be central to the design of DynamoDB. However this early version tightly coupled the assignment of both capacity and performance to individual partitions, which led to challenges.

DynamoDB uses admission control to ensure that storage nodes don't become overloaded, to avoid interference between co-resident table partitions, and to enforce the throughput limits requested by customers. Admission control in DynamoDB has evolved over the past decade. Admission control was the shared responsibility of all storage nodes for a table. Storage nodes independently performed admission control based on the allocations of their locally stored partitions. Given that a storage node hosts partitions from multiple tables, the allocated throughput of each partition was used to isolate the workloads. DynamoDB enforced a cap on the maximum throughput that could be allocated to a single partition, and ensured that the total throughput of all the partitions hosted by a storage node is less than or equal to the maximum allowed throughput on the node as determined by the physical characteristics of its storage drives.

The throughput allocated to partitions was adjusted when the overall table's throughput was changed or its partitions were split into child partitions. When a partition was split for size, the allocated throughput of the parent partition was equally divided among the child partitions. When a partition was split for throughput, the new partitions were allocated throughput based on the table's provisioned throughput. For example, assume that a partition can accommodate a maximum provisioned throughput of 1000 WCUs. When a table is created with 3200 WCUs, DynamoDB created four partitions that each would be allocated 800 WCUs. If the table's provisioned throughput was increased to 3600 WCUs, then each partition's capacity would increase to 900 WCUs. If the table's provisioned throughput was increased to 6000 WCUs, then the partitions would be split to create eight child partitions, and each partition would be allocated 750 WCUs. If the table's capacity was decreased to 5000 WCUs, then each partition's capacity would be decreased to 675 WCUs.

The uniform distribution of throughput across partitions is based on the assumptions that an application uniformly accesses keys in a table and the splitting a partition for size equally splits the performance. However, we discovered that application workloads frequently have non-uniform access patterns both over time and over key ranges. When the request rate within a table is non-uniform, splitting a partition and dividing performance allocation proportionately can result in the *hot* portion of the partition having less available performance than it did before the split. Since throughput was allocated statically and enforced at a partition level, these non-uniform workloads occasionally resulted in an application's reads and writes being rejected, called throttling, even though the total provisioned throughput of the table was sufficient to meet its needs.

Two most commonly faced challenges by the applications were: *hot partitions* and *throughput dilution*. Hot partitions arose in applications that had traffic going consistently towards a few items of their tables. The hot items could belong to a stable set of partitions or could hop around to different partitions over time. Throughput dilution was common for tables where partitions were split for size. Splitting a partition for size would cause the throughput of the partition to be divided equally among the newly created child partitions, and hence the per partition throughput would decrease.

In both cases, from the customer's perspective, throttling caused their application to experience periods of unavailability even though the service was behaving as expected. Customers who experienced throttling would work around it by increasing a table's provisioned throughput and not use all the capacity. That is, tables would be over-provisioned. While this allowed them to achieve the performance they needed, it was a poor experience because it was difficult to estimate the right level of performance provisioning for their tables.

4.1 Initial improvements to admission control

As we mentioned at the start of this section, hot partitions and throughput dilution stemmed from tightly coupling a rigid performance allocation to each partition, and dividing that allocation as partitions split. We liked that enforcing allocations at an individual partition level avoided the need for the complexities of distributed admission control, but it became clear these controls weren't sufficient. Shortly after launching, DynamoDB introduced two improvements, bursting and adaptive capacity, to address these concerns.

4.1.1 Bursting

The key observation that partitions had non-uniform access also led us to observe that not all partitions hosted by a storage node used their allocated throughput simultaneously. Hence, to absorb temporal spikes in workloads at a partition level, DynamoDB introduced the concept of bursting. The idea behind bursting was to let applications tap into the unused capacity at a partition level on a best effort basis to absorb short-lived spikes. DynamoDB retained a portion of a partition's unused capacity for later bursts of throughput usage for up to 300 seconds and utilized it when consumed capacity exceeded the provisioned capacity of the partition. The unused capacity is called *burst* capacity.

DynamoDB still maintained workload isolation by ensuring

that a partition could only burst if there was unused throughput at the node level. The capacity was managed on the storage node using multiple token buckets: two for each partition (allocated and burst) and one for the node. These buckets provided admission control. When a read or write request arrived on a storage node, if there were tokens in the partition's allocated token bucket, then the request was admitted and tokens were deducted from the partition and node level bucket. Once a partition had exhausted all the provisioned tokens, requests were allowed to burst only when tokens were available both in the burst token bucket and the node level token bucket. Read requests were accepted based on the local token buckets. Write requests using burst capacity required an additional check on the node-level token bucket of other member replicas of the partition. The leader replica of the partition periodically collected information about each of the members node-level capacity. In section 4.3 we explain how we increased a node's ability to burst.

4.1.2 Adaptive capacity

DynamoDB launched adaptive capacity to better absorb long-lived spikes that cannot be absorbed by the burst capacity. Adaptive capacity allowed DynamoDB to better absorb workloads that had heavily skewed access patterns across partitions. Adaptive capacity actively monitored the provisioned and consumed capacity of all the tables. If a table experienced throttling and the table level throughput was not exceeded, then it would automatically increase (boost) the allocated throughput of the partitions of the table using a proportional control algorithm. If the table was consuming more than its provisioned capacity then capacity of the partitions which received the boost would be decreased. The autoadmin system ensured that partitions receiving boost were relocated to an appropriate node that had the capacity to serve the increased throughput, however like bursting, adaptive capacity was also best-effort but eliminated over 99.99% of the throttling due to skewed access pattern.

4.2 Global admission control

Even though DynamoDB had substantially reduced the throughput problem for non-uniform access using bursting and adaptive capacity, both the solutions had limitations. Bursting was only helpful for short-lived spikes in traffic and it was dependent on the node having throughput to support bursting. Adaptive capacity was reactive and kicked in only after throttling had been observed. This meant that the application using the table had already experienced brief period of unavailability. The salient takeaway from bursting and adaptive capacity was that we had tightly coupled partition level capacity to admission control. Admission control was distributed and performed at a partition level. DynamoDB realized it would going to be beneficial to remove admission control from the partition and let the partition burst *always* while providing workload isolation.

To solve the problem of admission control, DynamoDB replaced adaptive capacity with global admission control (GAC). GAC builds on the same idea of token buckets. The GAC service centrally tracks the total consumption of the table capacity in terms of tokens. Each request router maintains a local token bucket to make admission decisions and communicates with GAC to replenish tokens at regular intervals (in the order of few seconds). GAC maintains an ephemeral state computed on the fly from client requests. Each GAC server can be stopped and restarted without any impact on the overall operation of the service. Each GAC server can track one or more token buckets configured independently. All the GAC servers are part of an independent hash ring. Request routers manage several time-limited tokens locally. When a request from the application arrives, the request router deducts tokens. Eventually, the request router will run out of tokens because of consumption or expiry. When the request router runs of tokens, it requests more tokens from GAC. The GAC instance uses the information provided by the client to estimate the global token consumption and vends tokens available for the next time unit to the client's share of overall tokens. Thus, it ensures that non-uniform workloads that send traffic to only a subset of items can execute up to the maximum partition capacity.

In addition to the global admission control scheme, the partition-level token buckets were retained for defense-in-depth. The capacity of these token buckets is then capped to ensure that one application doesn't consume all or a significant share of the resources on the storage nodes.

4.3 Balancing consumed capacity

Letting partitions always burst required DynamoDB to manage burst capacity effectively. DynamoDB runs on a variety of hardware instance types. These instance types vary by throughput and storage capabilities. The latest generation of storage nodes hosts thousands of partition replicas. The partitions hosted on a single storage node could be wholly unrelated and belong to different tables. Hosting replicas from multiple tables on a storage node, where each table could be from a different customer and have varied traffic patterns involves defining an allocation scheme that decides which replicas can safely co-exist without violating critical properties such as availability, predictable performance, security, and elasticity.

Colocation was a straightforward problem with provisioned throughput tables. Colocation was more manageable in the provisioned mode because of static partitions. Static partitions made the allocation scheme reasonably simple. In the case of provisioned tables without bursting and adaptive capacity, allocation involved finding storage nodes that could accommodate a partition based on its allocated capacity. Partitions

were never allowed to take more traffic than their allocated capacity and, hence there were no noisy neighbors. All partitions on a storage node did not utilize their total capacity at a given instance. Bursting when trying to react to the changing workload meant that the storage node might go above its prescribed capacity and thus made the colocation of tenants a more complex challenge. Thus, the system packed storage nodes with a set of replicas greater than the node's overall provisioned capacity. DynamoDB implemented a system to proactively balance the partitions allocated across the storage nodes based on throughput consumption and storage to mitigate availability risks caused by tightly packed replicas. Each storage node independently monitors the overall throughput and data size of all its hosted replicas. In case the throughput is beyond a threshold percentage of the maximum capacity of the node, it reports to the autoadmin service a list of candidate partition replicas to move from the current node. The autoadmin finds a new storage node for the partition in the same or another Availability Zone that doesn't have a replica of this partition.

4.4 Splitting for consumption

Even with GAC and the ability for partitions to burst always, tables could experience throttling if their traffic was skewed to a specific set of items. To address this problem, DynamoDB automatically scales out partitions based on the throughput consumed. Once the consumed throughput of a partition crosses a certain threshold, the partition is split for consumption. The split point in the key range is chosen based on key distribution the partition has observed. The observed key distribution serves as a proxy for the application's access pattern and is more effective than splitting the key range in the middle. Partition splits usually complete in the order of minutes. There are still class of workloads that cannot benefit from split for consumption. For example, a partition receiving high traffic to a single item or a partition where the key range is accessed sequentially will not benefit from split. DynamoDB detects such access patterns and avoids splitting the partition.

4.5 On-demand provisioning

Many applications that migrated to DynamoDB previously ran on-premises or on self-hosted databases. In either scenario, the application developer had to provision servers. DynamoDB provides a simplified serverless operational model and a new model for provisioning - read and write capacity units. Because the concept of capacity units was new to customers, some found it challenging to forecast the provisioned throughput. As mentioned in the beginning of this section, customers either over provisioned, which resulted in low utilization or under provisioned which resulted in throttles. To improve the customer experience for spiky workloads, we launched on-demand tables. On-demand tables remove the burden from our customers of figuring out the right provisioning for tables. DynamoDB provisions the on-demand tables based on the consumed capacity by collecting the signal of reads and writes and instantly accommodates up to double the previous peak traffic on the table. If an application needs more than double the previous peak on table, DynamoDB automatically allocates more capacity as the traffic volume increases to ensure that the workload does not experience throttling. On-demand scales a table by splitting partitions for consumption. The split decision algorithm is based on traffic. GAC allows DynamoDB to monitor and protect the system from one application consuming all the resources. The ability to balance based on consumed capacity effectively means partitions of on-demand tables can be placed intelligently so as to not run into node level limits.

5 Durability and correctness

Data should never be lost after it has been committed. In practice, data loss can occur because of hardware failures, software bugs, or hardware bugs. DynamoDB is designed for high durability by having mechanisms to prevent, detect, and correct any potential data losses.

5.1 Hardware failures

As with most database management systems, the write-ahead logs [15] in DynamoDB are central for providing durability and crash recovery. Write ahead logs are stored in all three replicas of a partition. For higher durability, the write ahead logs are periodically archived to S3, an object store that is designed for 11 nines of durability. Each replica still contains the most recent write-ahead logs that are usually waiting to be archived. The unarchived logs are typically a few hundred megabytes in size. In a large service, hardware failures such as memory and disk failures are common. When a node fails, all replication groups hosted on the node are down to two copies. The process of healing a storage replica can take several minutes because the repair process involves copying the B-tree and write-ahead logs. Upon detecting an unhealthy storage replica, the leader of a replication group adds a log replica to ensure there is no impact on durability. Adding a log replica takes only a few seconds because the system has to copy only the recent write-ahead logs from a healthy replica to the new replica without the B-tree. Thus, quick healing of impacted replication groups using log replicas ensures high durability of most recent writes.

5.2 Silent data errors

Some hardware failures can cause incorrect data to be stored [5, 7]. In our experience, these errors can happen because of the storage media, CPU, or memory [5]. Unfortunately, it's

very difficult to detect these and they can happen anywhere in the system. DynamoDB makes extensive use of checksums to detect silent errors. By maintaining checksums within every log entry, message, and log file, DynamoDB validates data integrity for every data transfer between two nodes. These checksums serve as guardrails to prevent errors from spreading to the rest of the system. For example, a checksum is computed for every message between nodes or components and is verified because these messages can go through various layers of transformations before they reach their destination. Without such checks, any of the layers could introduce a silent error.

Every log file that is archived to S3 has a manifest that contains information about the log, such as a table, partition and start and end markers for the data stored in the log file. The agent responsible for archiving log files to S3 performs various checks before uploading the data. These include and are not limited to verification of every log entry to ensure that it belongs to the correct table and partition, verification of checksums to detect any silent errors, and verification that the log file doesn't have any holes in the sequence numbers. Once all the checks are passed, the log file and its manifest are archived. Log archival agents run on all three replicas of the replication group. If one of the agents finds that a log file is already archived, the agent downloads the uploaded file to verify the integrity of the data by comparing it with its local write-ahead log. Every log file and manifest file are uploaded to S3 with a content checksum. The content checksum is checked by S3 as part of the put operation, which guards against any errors during data transit to S3.

5.3 Continuous verification

DynamoDB also continuously verifies data at rest. Our goal is to detect any silent data errors or bit rot in the system. An example of such a continuous verification system is the *scrub* process. The goal of scrub is to detect errors that we had not anticipated, such as bit rot. The scrub process runs and verifies two things: all three copies of the replicas in a replication group have the same data, and the data of the live replicas matches with a copy of a replica built offline using the archived write-ahead log entries. The process of building a replica using archived logs is explained in section 5.5 below. The verification is done by computing the checksum of the live replica and matching that with a snapshot of one generated from the log entries archived in S3. The scrub mechanism acts as a defense in depth to detect divergences between the live storage replicas with the replicas built using the history of logs from the inception of the table. These comprehensive checks have been very beneficial in providing confidence in the running system. A similar technique of continuous verification is used to verify replicas of global tables. Over the years, we have learned that continuous verification of data-at-rest is the most reliable method of protecting against hardware failures, silent data corruption, and even software bugs.

5.4 Software bugs

DynamoDB is a distributed key-value store that's built on a complex substrate. High complexity increases the probability of human error in design, code, and operations. Errors in the system could cause loss or corruption of data, or violate other interface contracts that our customers depend on. We use formal methods [16] extensively to ensure the correctness of our replication protocols. The core replication protocol was specified using TLA+ [12, 13]. When new features that affect the replication protocol are added, they are incorporated into the specification and model checked. Model checking has allowed us to catch subtle bugs that could have led to durability and correctness issues before the code went into production. Other services such as S3 [6] have also found model-checking useful in similar scenarios.

We also employ extensive failure injection testing and stress testing to ensure the correctness of every piece of software deployed. In addition to testing and verifying the replication protocol of the data plane, formal methods have also been used to verify the correctness of our control plane and features such as distributed transactions.

5.5 Backups and restores

In addition to guarding against physical media corruption, DynamoDB also supports backup and restore to protect against any logical corruption due to a bug in a customer's application. Backups or restores don't affect performance or availability of the table as they are built using the write-ahead logs that are archived in S3. The backups are consistent across multiple partitions up to the nearest second. The backups are full copies of DynamoDB tables and are stored in an Amazon S3 bucket. Data from a backup can be restored to a new DynamoDB table at any time.

DynamoDB also supports point-in-time restore. Using point-in-time restore, customers can restore the contents of a table that existed at any time in the previous 35 days to a different DynamoDB table in the same region. For tables with the point-in-time restore enabled, DynamoDB creates periodic snapshots of the partitions that belong to the table and uploads them to S3. The periodicity at which a partition is snapshotted is decided based on the amount of write-ahead logs accumulated for the partition. The snapshots, in conjunction to write-ahead logs, are used to do point-in-time restore. When a point-in-time restore is requested for a table, DynamoDB identifies the closest snapshots to the requested time for all the partitions of the tables, applies the logs up to the timestamp in the restore request, creates a snapshot of the table, and restores it.

6 Availability

To achieve high availability, DynamoDB tables are distributed and replicated across multiple Availability Zones (AZ) in a Region. DynamoDB regularly tests resilience to node, rack, and AZ failures. For example, to test the availability and durability of the overall service, power-off tests are exercised. Using realistic simulated traffic, random nodes are powered off using a job scheduler. At the end of all the power-off tests, the test tools verify that the data stored in the database is logically valid and not corrupted. This section expands on some of the challenges solved in the last decade to ensure high availability.

6.1 Write and consistent read availability

A partition's write availability depends on its ability to have a healthy leader and a healthy write quorum. A healthy write quorum in the case of DynamoDB consists of two out of the three replicas from different AZs. A partition remains available as long as there are enough healthy replicas for a write quorum and a leader. A partition will become unavailable for writes if the number of replicas needed to achieve the minimum quorum are unavailable. If one of the replicas is unresponsive, the leader adds a log replica to the group. Adding a log replica is the fastest way to ensure that the write quorum of the group is always met. This minimizes disruption to write availability due to an unhealthy write quorum. The leader replica serves consistent reads. Introducing log replicas was a big change to the system, and the formally proven implementation of Paxos provided us the confidence to safely tweak and experiment with the system to achieve higher availability. We have been able to run millions of Paxos groups in a Region with log replicas. Eventually consistent reads can be served by any of the replicas. In case a leader replica fails, other replicas detect its failure and elect a new leader to minimize disruptions to the availability of consistent reads.

6.2 Failure detection

A newly elected leader will have to wait for the expiry of the old leader's lease before serving any traffic. While this only takes a couple of seconds, the elected leader cannot accept any new writes or consistent read traffic during that period, thus disrupting availability. One of the critical components for a highly available system is failure detection for the leader. Failure detection must be quick and robust to minimize disruptions. False positives in failure detection can lead to more disruptions in availability. Failure detection works well for failure scenarios where every replica of the group loses connection to the leader. However, nodes can experience gray network failures. Gray network failures can happen because of communication issues between a leader and follower, issues with outbound or inbound communication of a node, or front-end routers facing communication issues with the leader even though the leader and followers can communicate with each other. Gray failures can disrupt availability because there might be a false positive in failure detection or no failure detection. For example, a replica that isn't receiving heartbeats from a leader will try to elect a new leader. As mentioned in the section above, this can disrupt availability. To solve the availability problem caused by gray failures, a follower that wants to trigger a failover sends a message to other replicas in the replication group asking if they can communicate with the leader. If replicas respond with a healthy leader message, the follower drops its attempt to trigger a leader election. This change in the failure detection algorithm used by DynamoDB significantly minimized the number of false positives in the system, and hence the number of spurious leader elections.

6.3 Measuring availability

DynamoDB is designed for 99.999 percent availability for global tables and 99.99 percent availability for Regional tables. Availability is calculated for each 5-minute interval as the percentage of requests processed by DynamoDB that succeed. To ensure these goals are being met, DynamoDB continuously monitors availability at service and table levels. The tracked availability data is used to analyze customer perceived availability trends and trigger alarms if customers see errors above a certain threshold. These alarms are called customer-facing alarms (CFA). The goal of these alarms is to report any availability-related problems and proactively mitigate the problem either automatically or through operator intervention. In addition to real-time tracking, the system runs daily jobs that trigger aggregation to calculate aggregate availability metrics per customer. The results of the aggregation are uploaded to S3 for regular analysis of availability trends.

DynamoDB also measures and alarms on availability observed on the client-side. There are two sets of clients used to measure the user-perceived availability. The first set of clients are internal Amazon services using DynamoDB as the data store. These services share the availability metrics for DynamoDB API calls as observed by their software. The second set of clients is our DynamoDB canary applications. These applications are run from every AZ in the Region, and they talk to DynamoDB through every public endpoint. Real application traffic allows us to reason about DynamoDB availability and latencies as seen by our customers and catch gray failures [10, 11]. They are a good representation of what our customers might be experiencing both as long and short-term trends.

6.4 Deployments

Unlike a traditional relational database, DynamoDB takes care of deployments without the need for maintenance win-

dows and without impacting the performance and availability that customers experience. Software deployments are done for various reasons, including new features, bug fixes, and performance optimizations. Often deployments involve updating numerous services. DynamoDB pushes software updates at a regular cadence. A deployment takes the software from one state to another state. The new software being deployed goes through a full development and test cycle to build confidence in the correctness of the code. Over the years, across multiple deployments, DynamoDB has learned that it's not just the end state and the start state that matter; there could be times when the newly deployed software doesn't work and needs a rollback. The rolled-back state might be different from the initial state of the software. The rollback procedure is often missed in testing and can lead to customer impact. DynamoDB runs a suite of upgrade and downgrade tests at a component level before every deployment. Then, the software is rolled back on purpose and tested by running functional tests. DynamoDB has found this process valuable for catching issues that otherwise would make it hard to rollback if needed.

Deploying software on a single node is quite different from deploying software to multiple nodes. The deployments are not atomic in a distributed system, and, at any given time, there will be software running the old code on some nodes and new code on other parts of the fleet. The additional challenge with distributed deployments is that the new software might introduce a new type of message or change the protocol in a way that old software in the system doesn't understand. DynamoDB handles these kinds of changes with read-write deployments. Read-write deployment is completed as a multi-step process. The first step is to deploy the software to read the new message format or protocol. Once all the nodes can handle the new message, the software is updated to send new messages. New messages are enabled with software deployment as well. Read-write deployments ensure that both types of messages can coexist in the system. Even in the case of rollbacks, the system can understand both old and new messages.

All the deployments are done on a small set of nodes before pushing them to the entire fleet of nodes. The strategy reduces the potential impact of faulty deployments. DynamoDB sets alarm thresholds on availability metrics (mentioned in section 6.3). If error rates or latency exceed the threshold values during deployments, the system triggers automatic rollbacks. Software deployments to storage nodes trigger leader failovers that are designed to avoid any impact to availability. The leader replicas relinquish leadership and hence the group's new leader doesn't have to wait for the old leader's lease to expire.

6.5 Dependencies on external services

To ensure high availability, all the services that DynamoDB depends on in the request path should be more highly available than DynamoDB. Alternatively, DynamoDB should be able to continue to operate even when the services on which it depends are impaired. Examples of services DynamoDB depends on for the request path include AWS Identity and Access Management Services (IAM) [2], and AWS Key Management Service (AWS KMS) [3] for tables encrypted using customer keys. DynamoDB uses IAM and AWS KMS to authenticate every customer request. While these services are highly available, DynamoDB is designed to operate when these services are unavailable without sacrificing any of the security properties that these systems provide.

In the case of IAM and AWS KMS, DynamoDB employs a statically stable design [18], where the overall system keeps working even when a dependency becomes impaired. Perhaps the system doesn't see any updated information that its dependency was supposed to have delivered. However, everything before the dependency became impaired continues to work despite the impaired dependency. DynamoDB caches result from IAM and AWS KMS in the request routers that perform the authentication of every request. DynamoDB periodically refreshes the cached results asynchronously. If IAM or KMS were to become unavailable, the routers will continue to use the cached results for pre-determined extended period. Clients that send operations to request routers that don't have the cached results will see an impact. However, we have seen a minimal impact in practice when AWS KMS or IAM is impaired. Moreover, caches improve response times by removing the need to do an off-box call, which is especially valuable when the system is under high load.

6.6 Metadata availability

One of the most important pieces of metadata the request routers needs is the mapping between a table's primary keys and storage nodes. At launch, DynamoDB stored the metadata in DynamoDB itself. This routing information consisted of all the partitions for a table, the key range of each partition, and the storage nodes hosting the partition. When a router received a request for a table it had not seen before, it downloaded the routing information for the entire table and cached it locally. Since the configuration information about partition replicas rarely changes, the cache hit rate was approximately 99.75 percent. The downside is that caching introduces bimodal behavior. In the case of a cold start where request routers have empty caches, every DynamoDB request would result in a metadata lookup, and so the service had to scale to serve requests at the same rate as DynamoDB. This effect has been observed in practice when new capacity is added to the request router fleet. Occasionally the metadata service traffic would spike up to 75 percent. Thus, introducing new

request routers impacted the performance and could make the system unstable. In addition, an ineffective cache can cause cascading failures to other parts of the system as the source of data falls over from too much direct load [4].

DynamoDB wanted to remove and significantly reduce the reliance on the local cache for request routers and other metadata clients without impacting the latency of the customer requests. When servicing a request, the router needs only information about the partition hosting the key for the request. Therefore, it was wasteful to get the routing information for the entire table, especially for large tables with many partitions. To mitigate against metadata scaling and availability risks in a cost-effective fashion, DynamoDB built an in-memory distributed datastore called *MemDS*. MemDS stores all the metadata in memory and replicates it across the MemDS fleet. MemDS scales horizontally to handle the entire incoming request rate of DynamoDB. The data is highly compressed. The MemDS process on a node encapsulates a Perkle data structure, a hybrid of a Patricia tree [17] and a Merkle tree. The Perkle tree allows keys and associated values to be inserted for subsequent lookup using the full key or a key prefix. Additionally, as keys are stored in sorted order, range queries such as *lessThan*, *greaterThan*, and *between* are also supported. The MemDS Perkle tree additionally supports two special lookup operations: `floor` and `ceiling`. The `floor` operation accepts a key and returns a stored entry from the Perkle whose key is less than or equal to the given key. The `ceiling` operation is similar but returns the entry whose key is greater than or equal to the given key.

A new partition map cache was deployed on each request router host to avoid the bi-modality of the original request router caches. In the new cache, a cache hit also results in an asynchronous call to MemDS to refresh the cache. Thus, the new cache ensures the MemDS fleet is always serving a constant volume of traffic regardless of cache hit ratio. The constant traffic to the MemDS fleet increases the load on the metadata fleet compared to the conventional caches where the traffic to the backend is determined by cache hit ratio, but prevents cascading failures to other parts of the system when the caches become ineffective.

DynamoDB storage nodes are the authoritative source of partition membership data. Partition membership updates are pushed from storage nodes to MemDS. Each partition membership update is propagated to all MemDS nodes. If the partition membership provided by MemDS is stale, then the incorrectly contacted storage node either responds with the latest membership if known or responds with an error code that triggers another MemDS lookup by the request router.

7 Micro benchmarks

To show that scale doesn't affect the latencies observed by applications, we ran YCSB [8] workloads of types A (50 percent reads and 50 percent updates) and B (95 percent reads and 5 percent updates). Both benchmarks used a uniform key distribution and items of size 900 bytes. The workloads were run against production DynamoDB in the North Virginia region. The workloads were scaled from 100 thousand total operations per second to 1 million total operations per second. Figure 5 shows the read latencies of both workloads at 50^{th} and 99^{th} percentiles. The purpose of the graph is to show, even at different throughput, DynamoDB read latencies show very little variance and remain identical even as the throughput of the workload is increased. The read throughput of the Workload B is twice that Workload A and still the latencies show very little variance. Figure 6 shows the writes latencies of both workloads at 50^{th} and 99^{th} percentiles. Like the read latencies, the write latencies remain constant no matter the throughput of the workload. In case of YCSB, workload A drives a higher throughput than workload B, but the write latency profile for both workloads are similar.

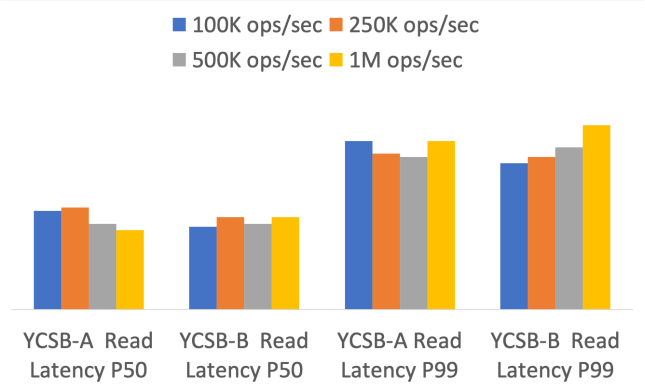

Figure 5: Summary of YCSB read latencies

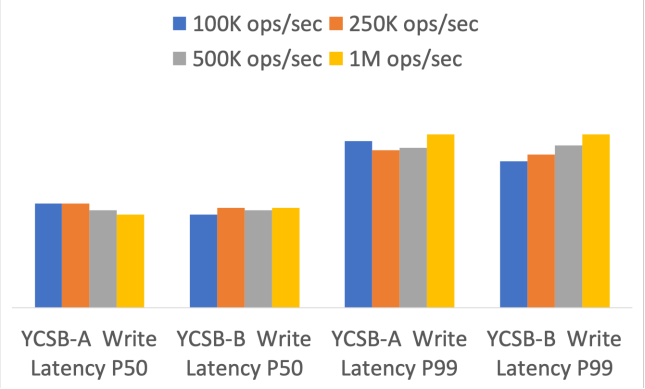

Figure 6: Summary of YCSB write latencies

8 Conclusion

DynamoDB has pioneered the space of cloud-native NoSQL databases. It is a critical component of thousands of applica-

tions used daily for shopping, food, transportation, banking, entertainment, and so much more. Developers rely on its ability to scale data workloads while providing steady performance, high availability, and low operational complexity. For more than 10 years, DynamoDB has maintained these key properties and extended its appeal to application developers with game-changing features such as on-demand capacity, point-in-time backup and restore, multi-Region replication, and atomic transactions.

9 Acknowledgements

DynamoDB has benefited greatly from its customers whose continuous feedback pushed us to innovate on their behalf. We have been lucky to have an amazing team working with us on this journey. We thank Shawn Bice, Rande Blackman, Marc Brooker, Lewis Bruck, Andrew Certain, Raju Gulabani, James Hamilton, Long Huang, Yossi Levanoni, David Lutz, Maximiliano Maccanti, Rama Pokkunuri, Tony Petrossian, Jim Scharf, Khawaja Shams, Stefano Stefani, Subu Subramanian, Allan Vermuellen, Wei Xiao and the entire DynamoDB team for the impactful contributions during the course of this evolution. Many people have helped to improve this paper. We thank the anonymous reviewers who helped shape the paper. Special thanks to Darcy Jayne, Kiran Reddy, and Andy Warfield for going above and beyond to help.

References

[1] Amazon SimpleDB: Simple Database Service. https://aws.amazon.com/simpledb/.

[2] AWS Identity and Account Management Service. https://aws.amazon.com/iam/.

[3] AWS Key Management Service. https://aws.amazon.com/kms/.

[4] Summary of the amazon dynamodb service disruption and related impacts in the us-east region. 2015. https://aws.amazon.com/message/5467D2/.

[5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):1–28, 2008.

[6] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[7] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak. Silent data corruption—myth or reality? In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 108–109. IEEE, 2008.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.

[10] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 545–557, 2020.

[11] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[12] L. Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002.

[13] L. Lamport. The pluscal algorithm language. In *International Colloquium on Theoretical Aspects of Computing*, pages 36–60. Springer, 2009.

[14] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[15] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[16] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[17] K. Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99. Citeseer, 1991.

[18] B. Weiss and M. Furr. Static stability using availability zones. https://aws.amazon.com/builders-library/static-stability-using-availability-zones/.