CS60203: Design Optimization of Computing Systems *November 17, 2024*

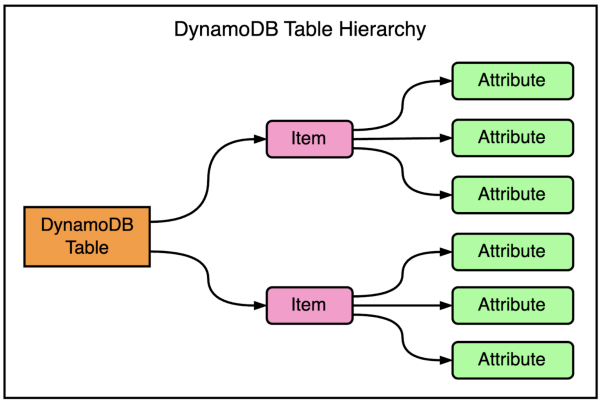
DynamoDB

Instructor: Mainack Mondal *Scribe-By: M Arka Rutvik*

Contents

1 DynamoDB Architecture

1.1 DynamoDB Tables



- Each item is uniquely identified by a primary key, and the schema of this key is specified at the time of table creation.
- The primary key's schema contains a partition key or can be a composite key (consisting of a partition and sort key).
- The partition key helps determine where the item will be physically stored.
- DynamoDB also supports secondary indexes to query data in a table using an alternate key.
- A particular table can have one or more secondary indexes.

1.2 Interface

The below table shows the primary operations that can be used by clients to read and write items in a DynamoDB table.

DynamoDB Interface

Operation	Functionality
PutItem	Insert a new item or replace an existing item with a new item
UpdateItem	Updates an existing item or adds a new item to the table if it doesn't exist
DeleteItem	Delete a single item from the table based on the primary key
GetItem	Returns a set of attributes for the item for the given primary key

2 Partitioning and Replication

A DynamoDB table is divided into multiple partitions to:

- Handle more throughput as requests increase.
- Store more data as the table grows.

Each partition of the table hosts a part of the table's key range. For example, if there are 100 keys and 5 partitions, each partition can hold 20 keys.

How does this ensure high availability?

Each partition has multiple replicas distributed across nodes (Availability Zones) .Together, these replicas form a replication group and improve the partition's availability and durability.

- A replication group consists of storage replicas that contain both the write-ahead logs and the B-tree that stores the key value data.
- A group can contain replicas that only store write-ahead log entries and not the key-value data. These replicas are known as log replicas.

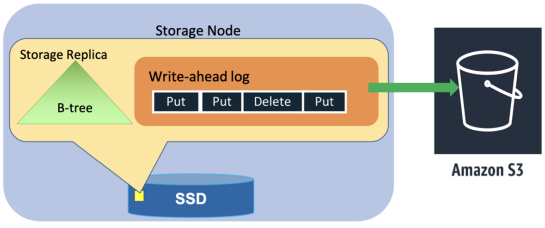


Figure 2: Storage replica on a storage node

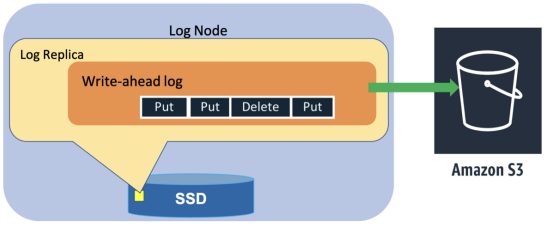


Figure 3: Log replica on a log node

But whenever you replicate data across multiple nodes, guaranteeing a consensus becomes a big issue. How do we ensure that each partition has the same value for a particular key ?

Amazon solved this with the help of **Multi-Paxos**.

The replication group uses Multi-Paxos for consensus and leader election. The leader is a key player within the replication group:

- The leader serves all write requests
- On receiving a write request, the leader of the group generates a write-ahead log record and sends it to the other replicas.
- A write is acknowledged to the application once a *quorum of replicas stores the log record to their local write-ahead logs.
- The leader also serves strongly consistent read requests. On the other hand, any other replica can serve eventually consistent reads.

But what happens if the leader goes down?

The leader of a replication group maintains its leadership using a lease mechanism. If the leader of the group fails and this failure is detected by any of the other replicas, the replica can propose a new round of the election to elect itself as the new leader.

3 DynamoDB Request Flow

The below diagram shows the request flow on a high level.

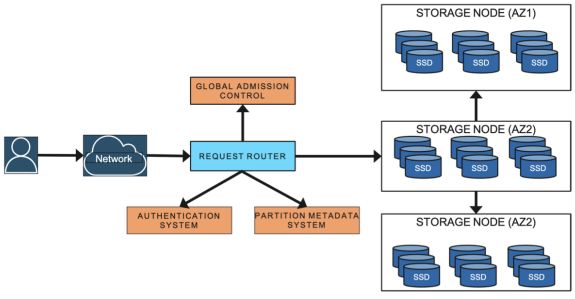


Figure 4: DynamoDB architecture

Let's understand how it works in a step-by-step manner

- Requests arrive at the request router service. This service is responsible for routing each request to the appropriate storage node. However, it needs to call other services to make the routing decision.
- The request router first checks whether the request is valid by calling the authentication service. The authentication service is hooked to the AWS IAM and helps determine whether the operation being performed on a given table is authorized.
- Next, the request router fetches the routing information from the metadata service. The metadata service stores routing information about the tables, indexes, and replication groups for keys of a given table or index.

- The request router also checks the global admission control to make sure that the request doesn't exceed the resource limit for the table.
- The request router calls the storage service to store the data on a fleet of storage nodes. Each storage node hosts many replicas belonging to different partitions.

4 Hot Partitions and Throughput Dilution

In the initial release, DynamoDB allowed customers to explicitly specify the throughput requirements for a table in terms of read capacity units (RCUs) and write capacity units (WCUs). As the demand from a table changed (based on size and load), it could be split into partitions.

For example, let's say a partition has a maximum throughput of 1000 WCUs. When a table is created with 3200 WCUs, DynamoDB creates 4 partitions with each partition allocated 800 WCUs.

All of this was controlled by the admission control system to make sure that storage nodes don't become overloaded. However, this approach assumed a uniform distribution of throughput across all partitions, resulting in some problems.

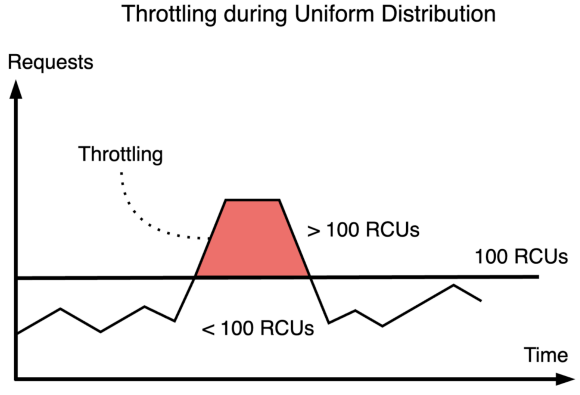
Two consequences because of this approach were hot partitions and throughput dilation.

Hot partitions

- Hot partitions arose in applications that had non-uniform access patterns. In other words, more traffic consistently went to a few items on the tables rather than an even distribution.

Throughput Dilutions

- Throughput dilution was common for tables where partitions were split for size. Splitting a partition for size would result in the throughput of the partition being equally divided among the child partitions. This would decrease the per-partition throughput.
- The static allocation of throughput at a partition level can cause reads and writes to be rejected if that partition receives a high number of requests. The partition's throughput limit was exceeded even though the total provisioned throughput of the table was sufficient. Such a condition is known as **throttling**.



From a customer's perspective, throttling creates periods of unavailability even though the service behaved as expected. To solve this, the customer will try to increase the table's provisioned throughput but not be able to use all that capacity effectively.

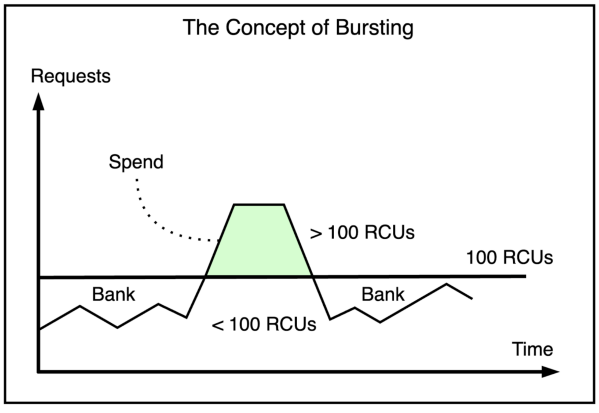
How to solve this issue?

Amazon came up with two ideas to solve this issue. **Bursting** and **Global Admission Control**

4.1 Bursting

The idea behind bursting was to let applications tap into the unused capacity at a partition level to absorb short-lived spikes for up to 300 seconds. The unused capacity is called the burst capacity.

It's the same as storing money in the bank from your salary each month to buy a new car with all those savings. The below diagram shows this concept.



4.2 Global Admission Control

Bursting took care of short-lived spikes. However, long-lived spikes were still a problem in cases that had heavily skewed access patterns across partitions.

DynamoDB developers implemented an adaptive capacity system that monitored the provisioned and consumed capacity of all tables. In case of throttling where the table level throughput wasn't exceeded, it would automatically boost the allocated throughput.

However, this was a reactive approach and kicked in only after the customer had experienced a brief period of unavailability.

To solve this problem, Amazon implemented Global Admission Control or GAC.

Here's how GAC works:

- It builds on the idea of token buckets by implementing a service that centrally tracks the total consumption of a table's capacity using tokens.
- Each request router instance maintains a local token bucket to make admission decisions.
- The routers also communicate with the GAC to replenish tokens at regular intervals.

- When a request arrives, the request router deducts tokens.
- When it runs out of tokens, it asks for more tokens from the GAC.
- The GAC instance uses the information provided by the client to estimate global token consumptions and provides the tokens available for the next time unit to the client's share of overall tokens.

5 Challenges Faced to Ensure High Availability

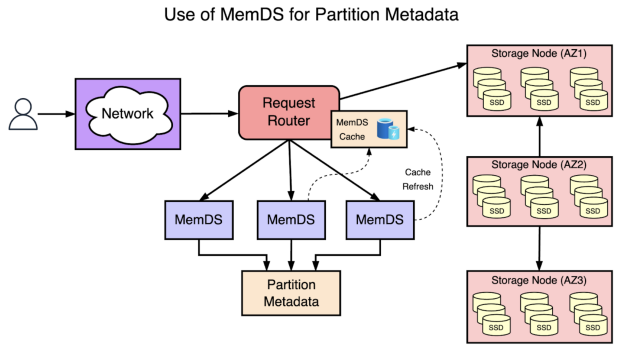
5.1 Metadata Availability

Metadata is the mapping between a table's primary keys and the corresponding storage nodes. Without this information, the requests cannot be routed to the correct 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 down- loaded 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.

However, the downside was that 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

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



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

References

- https://blog.bytebytego.com/p/a-deep-dive-into-amazon-dynamodb
- https://www.usenix.org/system/files/atc22-elhemali.pdf