DynamoDB

Overview

- NoSQL scalable cloud database service
- Supports fast and predictable performance even at scale
  - Emphasizes consistent low latency because unexpectedly high latency is bad for customer experience
  - Wants all requests to complete with 0-9 millisecond latency
- Fundamental Properties:
  - Users just create tables and read/write them without having to know anything about their implementation
    - Supports CRUD (create, read, update, delete)
  - Data from multiple users should be on the same machine to ensure we are maximizing the use of a singe machine (multi-tenant)
    - Users should be able to view their data as single-tenant though
  - Tables should be infinitely scalable
    - As they grow, a table's data should be spread among multiple servers
  - Predictable and low latencies
    - Automatically repartitions data to meet requirements
  - Highly available by replicating data across servers
  - Supports flexible use cases
    - Tables don't have a fixed schema and developers can request strong or eventual consistency
- Lessons learned from working on DynamoDB:
  - Adapt to customer traffic patterns to redo the partitioning scheme of databases
  - Perform continuous verification of data-at-rest to prevent against hardware failures / software bugs
  - Be very careful about changes to maintain high availability
  - Designing systems for predictability over absolute performance improves system stability
- History
  - Originally had Dynamo, which was a database system (not a service)
    - Hard to manage and each team had to spin up their own instance
    - Had predictable high performance and scalability
  - SimpleDB was a fully managed database service
    - Small capacity (10GB) for tables
    - Bad request throughput
    - Unpredictable query / write latencies because every table attribute was indexed, which had to be updated with every write
  - Wanted to combine the two to get DynamoDB

Architecture

- Request routers take in requests and then auth them / direct them to the desired server
- Each table contains items, each of which contains a primary key
  - This is hashed and this is used in combination with the sort key value (if present) to determine where the item is stored
- Secondary indices are used to help query a table by an alternative key
- Supports ACID (atomic, consistent, isolated, durable) transactions
- Each table is divide into partitions that handle a disjoint part of the table's key-range
  - Each has mutiple replicas distributed across availability zones
  - Each replication group uses multi-paxos for leader election / consensus
    - For writes, the leader generates a write-ahead log record and the write is acknowledged to application once a quorum (majority) of peers persist the log record

        - Any replica can serve eventually consistent reads, but only leader can do strongly-consistent reads
    - Each storage replica uses a B-tree to store key-value data
    - We can also have log replicas which only persist recent write-ahead log entries
- We have tens of microservices that handle things like repartitioning, determing if storage node is unhealthy, recovery, etc.

Admission Control

- Admission control is used to ensure that a user application didn't overload the DynamoDB servers
- Original:
    - Each table has specified maximum read / write throughput
    - Each partition of a table is allocated a given throughput, and for a machine storing multiple partitions, we make sure that the sum of throughput does not exceed machine capabilities
    - Originally, it was assumed that partitions for a table would uniformly be accessed, so throughput could be split evenly
        - This creates issues in user applications where hot partitions use much more throughput than cold partitions
        - This resulted in application reads / writes being rejected (throttling)
- Bursting
    - Key observation is that partitions had non-uniform access and that not all partitions hosteed by a storage node used their allocated throughput simultaneously
    - To absorb temporal spikes, applications can tap into unused capacity at a partition level in an attempt to absorb short-lived spikes
        - We retain unused capacity for up to 300 seconds and consume it when we need to handle a spike
        - Still need to make sure that this burst doesn't go beyond the replica's capacity, which requires keeping track of allocated capacity at the replica level
- Adaptive Capacity
    - For long-lived spikes that can't be absorbed by burst capacity
    - We monitor the provisioned and consumed capacity of all tables
        - If a table experienced throttling while the table level throughput was not exceeded, then it automatically increases the allocated throughput of the proprotions using a proportional control algorithm
    - Best-effort, but this along with bursting reduced 99.99% of throttling due to skewed accesses
- However, adaptive capacity was only reactive and bursting sometimes couldn't absorb it if the node was overloaded
    - As a result, wanted to decouple admission control from partitions
    - Replaced adaptive capacity with global admission control
    - We keep capacity per table globally, but we keep partition-level capacity buckets to make sure an application doesn't consume all resources on its replica / node
    - We let partitions always burst though
- DynamoDB then has a system to proactively balance partitions based on how much they're bursting / using up a node's overall provisioned capacity
    - It then tries to move partitions to balance them out
- We also scale partitions based on their throughput by splitting them once their consumed throughput goes too high
    - However, sometimes splits don't help when the entire key range is accessed sequentially or when a single item is being accessed, so DynamoDB detects these and then avoids splitting
- Customers frequently don't know exactly the throughput they need and would frequently over-provision
    - On-demand tables look at the traffic and then adjust to accomodate double the previous peak traffic
    - Automatically allocates more capacity as traffic volume increases

Durability / Availability

- Write-ahead logs from the Paxos replicas are central for providing durability

    - We use three replicas per partition
        - If any of these go down, we can very quickly add in a log replica which is fast because they don't need to copy the entire B-tree
    - We periodically archive these to S3, which is incredibly durable
- Sometimes, incorrect data can be written due to hardware failures
    - Makes extensive checksums to validate every data transfer between two nodes
- DynamoDB continuously verifies data at rest with checksums to build confidence in the system
- DynamoDB needs to store a lot of metadata, the most important of which is the mapping between primary keys and its location in the storage nodes
    - Originally this was just stored in DynamoDB and was then cached, but this created issues when the cache missed
    - Instead, Amazon created MemDS, which stored all metadata in memory and has a bunch of fancy stuff to make it fast

Transaction

- Supports transactions through TransactGetItems which atomically gets multiple items at the same time
- TransactWriteItems does multiple updates and also allows you to check values (to make sure your get is still valid) in an atomic action
- This is supported under the hood by the request router sending to Transaction Coordinator
    - Uses OCC and some commit protocol to get this to work
        - Uses timestamp ordering to determine if transactions are serializable