Let’s Play: DynamoDB!

What is DynamoDB?

- AWS Serverless / Managed Service Database
  - Massively scalable
  - No servers to operate or care and feed (incl. API and Database layers)
  - Incredibly cost-efficient
- Key/Value NoSQL Database (similar to MongoDB)
- Data is segregated into Tables
- Individual records in a Table are referred to as Items
- Items are structured objects that can be represented as JSON (incl. nesting!)
- There is no Table schema outside of Primary Key and Index definitions
  - Add additional Fields to any Item at any time without needing to first alter the Table
- Interact with via AWS API using IAM Roles
  - No DB Credentials to maintain/rotate/secure

Deploying Resources

- Like any other AWS Resource
  - AWS Console
  - AWS CLI
  - IaC Tools
    - Terraform
    - CloudFormation
- You will need to deploy:
  - Table Definition (+ Indexes)
  - IAM Roles + Policies for CRUD operations

DynamoDB & SQL Database Similarities

- Indexing specific fields is supported
- Filtering similar to WHERE clauses
- LIMIT supported
- Supports record limits on Query
- Bulk Querying, Updating and Deletion are supported
  - With Limitations
- Transactions are supported (with restrictions)
  - With Limitations

DynamoDB & SQL Database Dissimilarities

- Pagination required to retrieve large datasets
  - 1 Page (1MB of data, unchangeable) returned per Query or Scan call
  - Page Result includes a pointer to retrieve the next Page if present
- No Auto-Increment
  - You must implement a separate methodology to create Unique IDs
    - Use UUIDs; or
    - Use nanosecond accurate timestamps; or
    - Call a separate/custom service that provides monotonic incrementing counters
- No JOINs, GROUP BYs, or Nested Querying
  - Must aggregate items relations at the Application Layer
- No Unique Indexes aside from Primary Key
- No ALTER capability
  - You can add some indexes, but must drop+recreate the table to change Primary Key

The 2 Flavors of Primary Key

- Primary Keys come in 2 Flavors
  - Partition Key only
    - Works like a simple key value store
    - Value can be String or Number
    - Can only CRUD individual items by Partition Key, or Scan the entire table
      - Additional Indexes can be created that support Querying multiple items
  - Composite Primary Key = Partition Key + Sort Key
    - Provide the full composite key for Get, Create, Update, Put
    - Provide just the Partition Key for a Query call
      - Results will be ordered by the values of the Sort Key
      - Simple boolean parameter on the Query will reverse the order

Much ado about Secondary Indexes

- Indexes generally use Composite Keys (Partition Key + Sort Key)
  - Indexes without a Sort Key do not offer predictable ordering
- Any 2 numeric or string fields in a Item can be part of an Index
- Items not containing the fields relevant to an Index are excluded from it
- Indexes do not support Uniqueness
  - Cannot Get an individual Item via an Index
  - Must Query the against Index and assume an unbounded # of records are returned
- Up to 25 Indexes allowed per-Table
  - 20 of the “Global” flavor
  - 5 of the “Local” flavor
  - Quota can be increased by AWS upon request with proper justification

The 2 Flavors of Indexes

- Local Secondary Index (LSI)
  - Allows the re-use of the Table Partition Key with a different Sort Key
  - 10GB limit per-partition
    - Once reached, no new Items can be added to the Partition
    - Won’t scale for low-cardinality tables with many Items
  - Must be created at the time of Table creation
  - Generally not recommended by AWS for these reasons
- Global Secondary Index (GSI)
  - Allows any string/number fields to be a Partition Key and optional Sort Key
  - Can be created at any time
  - Requires between 5 and 20 minutes to add to an existing Table
    - no downtime required to add a new GSI

CRUD Operations

- GetItem
  - Retrieves an individual item by its Primary Key
- Query
  - Can only Query against the Table Partition Key or an Index's Partition Key
  - Retrieves 0 or more Items with the provided Partition Key, results ordered by the Sort Key
- Scan
  - Retrieves 0 or more Items by scanning the entire table
  - Results ordering is not predictable
- PutItem
  - **Creates** a new Item **or Replaces** an existing Item matching the provided Primary Key
  - Can use conditional expression to restrict to Create-only
- UpdateItem
  - **Creates** a new Item **or Updates** an existing Item matching the provided Primary Key
  - Can use conditional expression to restrict to Update-existing-only
- DeleteItem
  - Removes an existing item matching the provided Primary key
  - Will **not** return an error if the item is already absent at the time of the Delete operation

Batching

- BatchGetItem
  - Provide list of Items identified by Primary Key to retrieve
  - Returns 16MB of Data or 100 items, whichever is first
  - You can request a maximum of 100 items per batch call
  - If response for all items > 16MB, the Unprocessed Keys will be provided for Pagination
  - Item-specific failures do not abort the batch (you can get partial results)
- BatchWriteItem
  - Provide 16MB of Item Data or 25 items, whichever is first.
  - Can be a combination of PutItem and DeleteItem instructions
  - PutItem is a FULL OVERWRITE of an existing item by Primary Key, if present
    - No Batch Update/Upsert capability
    - To Batch Update, use BatchGetItem, update the items in code, then BatchWriteItem
  - Item-specific failures do not abort or rollback the batch
    - Successful writes to individual Items will persist

Transactions

- TransactGetItems
  - Like BatchGetItem, but will fail completely if anything goes wrong
  - Supports up to 100 Items per-transaction or 4MB worth of Primary Keys
    - Whichever is smaller
- TransactWriteItems
  - Like BatchWriteItem, but will fail completely if anything goes wrong
  - Supports up to 4MB of Item Data or 100 items, whichever is first.

- Transaction Quotas can be prohibitively small, depending on use case
  - No way to coordinate rollbacks across multiple transactions

Projections

- Projections filter out unwanted fields from the output results
- Applied at the API layer once the Item is retrieved from storage
  - You are still metered in Read Capacity Units as if retrieving the the entire Item
- Could be useful for:
  - Limiting bandwidth between DynamoDB API and your Service
  - Avoiding Pagination due to response payload size

The 2 Flavors of Capacity

- On Demand Units
  - Pay as you go
  - 6x more expensive than Provisioned Units at Scale
  - Can safely/cheaply use for tables with < 1 req/sec
- Provisioned Units
  - Separate provisioning for Read and Write operations
    - Provides flexibility for read-heavy, write-heavy and read-write-heavy workloads
  - Can be adjusted at any time
    - CloudWatch provides useful auto-scaling of Provisioned Units to help control costs

1 Read Unit = 4KB of served data

1 Write Unit = 1KB of posted data

Throttling due to Capacity

- On Demand max Throughput
  - Per Account - Permits 40,000 read and 40,000 write requests concurrently
  - Per Table - Permits 12,000 read request units and 4,000 write requests concurrently
  - Per Partition - Permits 3,000 read request units and 1,000 write requests concurrently
- Provisioned Units
  - Throughput is as-configured
- Your application will be throttled if Max Throughput is reached
  - A ProvisionedThroughputExceededException is returned by DynamoDB
  - AWS recommends using use Exponential backoff when retrying
  - DynamoDB does allow intermittent bursting higher than Capacity, but don’t count on that
  - Temporarily bump up Write or Read Capacity for incidental bulk transactions as needed
- Cache where possible to minimize unnecessary Read Unit consumption

Using TTLs to Auto-Remove Records

- You can specify 1 field in the table as a TTL field
  - Must be an Integer providing Epoch Timestamp (in secs) for when the item is to be removed
  - It may take slightly longer than the TTL for the object to actually be removed
- Items in the table without the designated TTL field are not impacted
- Useful for:
  - Tables that function as a simple Cache
  - Items that are intended to be ephemeral

DynamoDB Streams

- Optional feature, similar to Postgres Triggers
- Detect when items change in a table and sends triggers to the configured locations for further action.
    - The trigger provides the Old and New Item versions, and Action (Put, Delete, Update)

Example:

1. User creates new Org, which adds item to DynamoDB table
2. DynamoDB sends notification to Lambda to provision additional resources for the Org (e.g., an S3 Bucket, a DataDog dashboard, etc.)

Please **Don’t** Use DynamoDB for These Things

- Storing Media Data
  - Store Video, Music, Graphics and Firmware Files in S3 instead, then
  - Store the S3 URL in DynamoDB
    - Generate pre-signed URLs at the application layer as needed
- Time Series data
  - Frequently relies on many different fields for aggregation/pivot
  - Would require a complex application layer
  - Problem is already solved
- Data that is highly normalized/relational or (in SQL land) requires many conjunctions in a WHERE clause
- Data that requires mathematical calculations from the DB engine
  - No equivalent of SUM, AVG, COUNT, etc. in SQL
- Workloads that exceed Max Table/Account-Level Capacities for read or write

Example DynamoDB Helper Library (golang)

https://github.com/jranson/letsplay-dynamodb

- Makes Create vs Upsert vs Update easy
- One-liners for GetItem, Query, Scan
- Gracefully Handles Pagination
  - You just tell it how many items and/or pages to return
- Gracefully handles BatchWriteItem of any size
  - Automatically breaks up large batches for AWS size + item count restrictions
  - But will still be susceptible to Throttling based on Capacity configuration

- Does not currently help with BatchGetItem
- Does not currently help with Transactions