AMAZON DYNAMODB

A FULLY-MANAGED DATABASE WITH HIGH PERFORMANCE AT ANY SCALE

INTRODUCTION 👋

Why should you care about DynamoDB?
Because it's **fully managed**, **highly available** & **scales on-demand** with **low latencies**.
For getting you hooked, at Prime Days 2021 DynamoDB served **89.2 million requests per second** at its peak!

HOT PARTITIONS 🔥

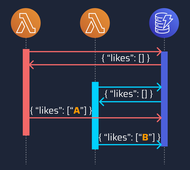
Your provisioned read & write capacity units will be distributed among all internal partitions.
If your partition keys are not well-distributed, it will be easier to get your requests throttled as a subset of your partitions (or worse: a single) can receive the **majority** of read and/or write requests.

DynamoDB will increase capacity for **hot partitions**, as long as the table-level throughput is not exceeded (**Adaptive Capacity**).
Even **more** awesomeness: If there's **disproportionately high traffic** to one or several items in the same partition, DynamoDB will **rebalance partitions** such that those items don't reside on the same partition.

RACE CONDITIONS ⚡

Often, there are possible race conditions due to multi-tenancy where writes can be **lost**.
Example:

1. **Lambda A** reads Document X
2. **Lambda B** reads Document X
3. **Lambda A** writes Document X
4. **Lambda B** writes Document X ⚡



UPDATE EXPRESSIONS 🔶

Safely modify certain fields of a complex document.

- **SET** - adding one or several attributes to an item.
- **REMOVE** - removing one or several attributes.
- **ADD** - adding new attributes with its value.
- **DELETE** - remove one or more elements from a set.

CAPACITY MODES 🏗️

You can choose between those two types, which can be changed **any time**.

- **provisioned** - specifying the capacity units for your table & you'll be independently charged of your actual usage.
- **on-demand** - you'll be charged per request.

💡 The best fit highly depends on your **traffic patterns**.

DATA TYPES 🔶

Besides your primary key, your document can contain other fields of different **Scalar**, **Document** and **Set Types**.

SCALAR

Represented by exactly one value.

- String (**S**)
- Number (**N**)
- Binary (**B**)
- Boolean (**BOOL**)
- Null (**NULL**)

DOCUMENT

Complex structure with nested attributes.

- List (**L**)
- Map (**M**)

SET

Multiple scalar values.

- String Set (**SS**)
- Number Set (**NS**)
- Binary Set (**BS**)

```
{
  "id": {
    "S": "sgsoqilsurumbhtn"
  },
  "createdAt": {
    "N": "1644857201"
  },
  "confirmed": {
    "BOOL": true
  }
}
```

```
{
  "aList": {
    "L": [
      { "S": "Pizza" },
      { "S": "Pasta" }
    ]
  },
  "aMap": {
    "M": {
      "firstKey": { "N": "30000" },
      "secondKey": { "BOOL": true }
    }
  },
}
```

OPTIMISTIC LOCKING 🔐

DynamoDB's **Optimistic Locking** allows us to verify that we're really updating the item we're expecting by using a dedicated version field.

In the previous example, both read operations would read version **1** and also expect to write version **2**. The first operation would succeed, but the second would fail as the expected version would not match with the expectations.

ON-DEMAND VS. PROVISIONED 💸

Go with **On-Demand** if you got unpredictable traffic, as it scales immediately and you're only paying for what you actually use.
With steady load or known patterns, pick **provisioned** as it can be up to almost **7 times** less expensive.

Rule of thumb:

- variable, **unpredictable** traffic → **on-demand**
- variable, **predictable** traffic → **provisioned**
- **steady**, **predicatable** traffic → **reserved**

RETRIEVING ITEMS 🔽

That's where it gets interesting and you see differences to SQL or other NoSQL solutions.
You can only **query** via your partition key (and sort key condition, if there's any) of your main or secondary indexes.
Everything else needs **scan**.

SCAN 🚤

A scan is just running through your table looking for items that are matching your expression.
You'll be **charged for the items that are scanned**, not the items that are retrieved.

QUERY 🎯

With **query**, you're only paying for the retrieved items. It's only looking for the items at a specific partition. So generally speaking: query is way **faster** and **cheaper**.

OBSERVABILITY 🔎

CloudWatch comes with default metrics like **used read & write capacity units** or the **number of throttles**. It helps to analyse DynamoDB traffic patterns so you can optimise for low costs.

Third-party tools like Dashbird.io help you with **Slack notifications** for critical events like throttles & give you general guidance with **well-architected tips**.

GLOBAL TABLES 🌎

Distribute your data gobally via global tables for redundancy and faster latencies.
A **bi-directional synchronization** will replicate your data between different regions.

FREE TIER 💰

Your first year includes **25** Read & Write Capacity Units each for **provisioned** capacity mode every month.

```
resources:
  Resources:
    Users:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: users
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        BillingMode: PAY_PER_REQUEST
```

```
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 25
        WriteCapacityUnits: 25
```

BASIC CONCEPTS 📝

In comparison to SQL, a document in DynamoDB doesn't have a fixed schema. What is defined by the table: the **primary key**, which **uniquely** identifies a single document.

PRIMARY KEYS 🥇

It's is a documents **unique** identifier & must be provided when inserting a new item.
There are two different types of primary keys:

- **simple** - your partition key & therefore a single field.
- **composite** - a combination of your partition & sort key.

PARTITION KEYS 🖐️

Internally, DynamoDB consists of **different partitions** and your partition key will determine the partition where an item will be stored at.

A good partition key is equally distributed to avoid **hot partitions**.

SORT KEYS 🥈

Using sort keys come in with benefits:

- they can be used with operators in queries such as **begins_with**, **between**, **>**, **<**, and so on.
- you can create hierarchal (1:n) releationships - citing an example from the docs: **[country]#[region]#[state]#[county]#[city]#[neighborhood]**

💡 The combination of a partition & sort key (= your primary key) needs to be **unique**.

BACKUPS 🔒

AWS offers you a lot of options for backing up your tables.

- **On-Demand Backups** - trigger backups manually or via a scheduled event.
- **Continuous Backup**s via **P**oint-**I**n-**T**ime-**R**ecovery (**PITR**) - enables you to restore your table to any state with the last 35 days
- **Exporting Backups to S3** - export your data to an S3 bucket. You can also automatically trigger this process via Lambda & EventBridge rules.

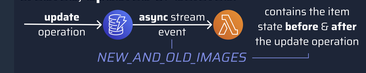
SECONDARY INDEXES 🔑

You can create two types of indexes, which are specifying **alternative key structures**. Those can also be used to query items.

- **L**ocal **S**econdary **I**ndex (**LSI**) - needs to have the same partition key, but an alternative sort key.
- **G**lobal **S**econdary **I**ndex (**GSI**) - partition & sort key can be different.

STREAMS 🫧

Asynchronously trigger invocations of other services like Lambda when an item is **created**, **updated** or **deleted**.



aws