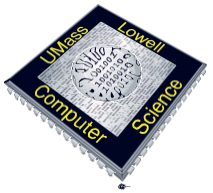
Kennedy College of Sciences

Computer Science Department



Cloud Computing Club

10/12/2023

Intro into Cloud Databases and DynamoDB

Prof. Johannes Weis

Databases in the Cloud

Very big topic – each database option could be its own course and more.

The two of the most popular/important database types in the cloud are:

Relational – SQL (think MySQL, Postgress, Oracle,…)

NoSQL (key-value pair, document, graph)

There are more types of databases:

In Memory databases

Cache based databases

https://cloud.google.com/blog/topics/developers-practitioners/your-google-cloud-database-options-explained

https://aws.amazon.com/free/database

Simplistic view:

SQL - RDS : good for structured data (records) organizing our data in tables, complex queries, combining multiple data sets ,….

NoSQL - DynamoDB : good for large scale and performance

https://www.youtube.com/watch?v=ORxMMo7it_Y - Relational (SQL) vs Non Relational (NoSQL)

DynamoDB supports *eventually consistent* and *strongly consistent* reads.

Eventually Consistent Reads

When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data.

Strongly Consistent Reads

When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. However, this consistency comes with some disadvantages:

- A strongly consistent read might not be available if there is a network delay or outage. In this case, DynamoDB may return a server error (HTTP 500).
- Strongly consistent reads may have higher latency than eventually consistent reads.
- Strongly consistent reads are not supported on global secondary indexes.
- Strongly consistent reads use more throughput capacity than eventually consistent reads. For details, see Read/write capacity mode

Choosing which database to use is one of the biggest and most impactful design decisions you will ever make!

There are tradeoffs and migrating from one database technology to another is often very difficult and time consuming! In my industry career I was majorly involved at least 3 times in multi-year projects where a major part was to migrate from one database to another.

Recommendation:

make this one of your most scrutinized design decisions for any project and …

if

- you can keep your data-model simple,
- do not need complex queries
- can live with consistency limitations

go with DynamoDB and if you need to go SQL go with a managed service (RDS).

But there is of course much more to this ….

RDS – AWS Relational Databases in the cloud

Think SQL (select statements) , tables, queries, inserts, deletes …..

Has been around for a long time and still widely used.

If you want to use a relational database on AWS, you have two options:

1) Use the managed relational database service Amazon RDS, which is offered by AWS.
2) Operate a relational database yourself on top of virtual machines.

Table 11.1 Managed service RDS vs. a self-hosted database on virtual machines

	Amazon RDS	Self-hosted on virtual machines
Cost for AWS services	Higher because RDS costs more than virtual machines (EC2)	Lower because virtual machines (EC2) are cheaper than RDS
Total cost of ownership	Lower because operating costs are split among many customers	Much higher because you need your own manpower to manage your database
Quality	AWS professionals are responsible for the managed service.	You'll need to build a team of professionals and implement quality control yourself.
Flexibility	High, because you can choose a relational database system and most of the configuration parameters	Higher, because you can control every part of the relational database system you installed on virtual machines

If SQL go with RDS where you can!

Tweaking SQL database performance

Scale Vertically:

1. Faster CPU
2. More memory
3. Faster storage

Optimize data schema (avoid too many joins) and table size (age out records, partioning)

Challenges when using relational databases:

1. Things change over time! – Always. Features (tables) are added, data model complexity increases over time and performance goes down
2. Scale mostly vertical – typically you will need a bigger box to scale – once you get to records in the 100's of millions maintaining performance becomes a big challenge
3. Maintenance task: including deletion of obsolete data, "aging" out data, data migrations to new schema, rebuilding indices and sometimes backups are expensive and often require down-time. System Updates, Replication and failover are your responsibility especially in self maintained environments

DynamoDB Developer guide

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

---

Tables

Similar to other database systems, DynamoDB stores data in tables. A *table* is a collection of data.

Items

Each table contains zero or more items. An *item* is a group of attributes that is uniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar in many ways to **rows, records, or tuples** in other database systems.

Attributes

Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to **fields or columns** in other database systems.

Primary key

Partition key – A simple primary key, composed of one attribute known as the *partition key*.

Partition key and sort key – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.

Secondary indexes

You can create one or more secondary indexes on a table. A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key.

DynamoDB supports two kinds of indexes:

- Global secondary index – An index with a partition key and sort key that can be different from those on the table.
- Local secondary index – An index that has the same partition key as the table, but a different sort key.

Each table in DynamoDB has a quota of 20 global secondary indexes (default quota) and 5 local secondary indexes

DynamoDB Streams

DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near-real time, and in the order that the events occurred.

Data types

DynamoDB supports many different data types for attributes within a table. They can be categorized as follows:

- **Scalar Types** – A scalar type can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.
- **Document Types** – A document type can represent a complex structure with nested attributes, such as you would find in a JSON document. The document types are list and map.
- **Set Types** – A set type can represent multiple scalar values. The set types are string set, number set, and binary set.

Autoscaling

Automatically scale read and write capacity up and down – best for "slow" scaling

Best invention for databases since the invention of sliced bread!

DynamoDB Best Practices (list not complete):

Understand your use case first: do not start with the datamodel (like in SQL). Instead ask: what queries do I need?

Understand your data: size, shape (match your queries) and velocity (number of partitions and how to distribute between them)

Keep related data together: As a general rule, you should maintain as few tables as possible in a DynamoDB application.

Use sort order: Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.

Distribute queries: It is also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding "hot spots."

Use global secondary indexes: By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

Avoid scanning tables: it is slow

Scaling: understand your scaling needs and use autoscaling whenever appropriate