AWS DynamoDB

CS516 – Cloud Computing

Computer Science Department

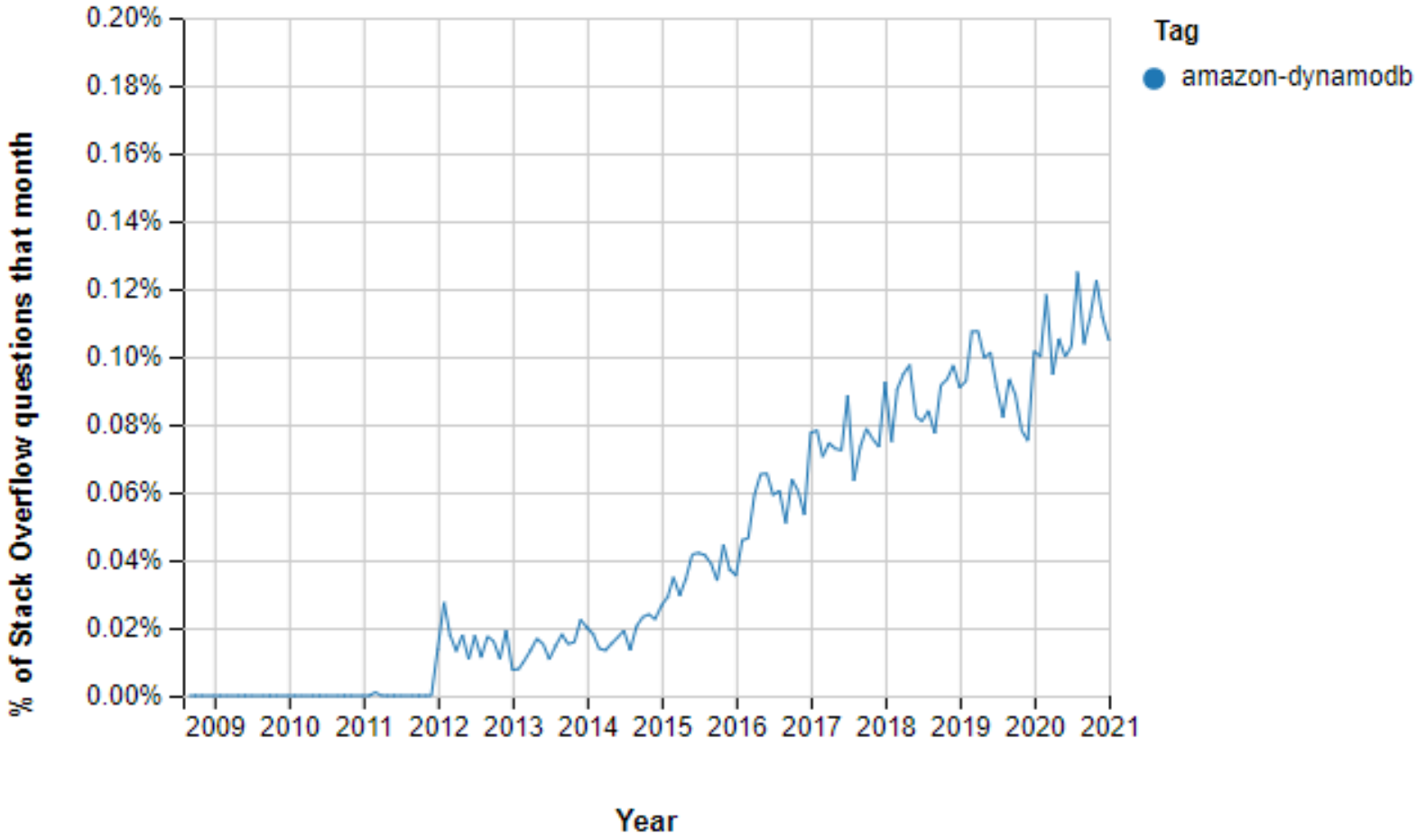
Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

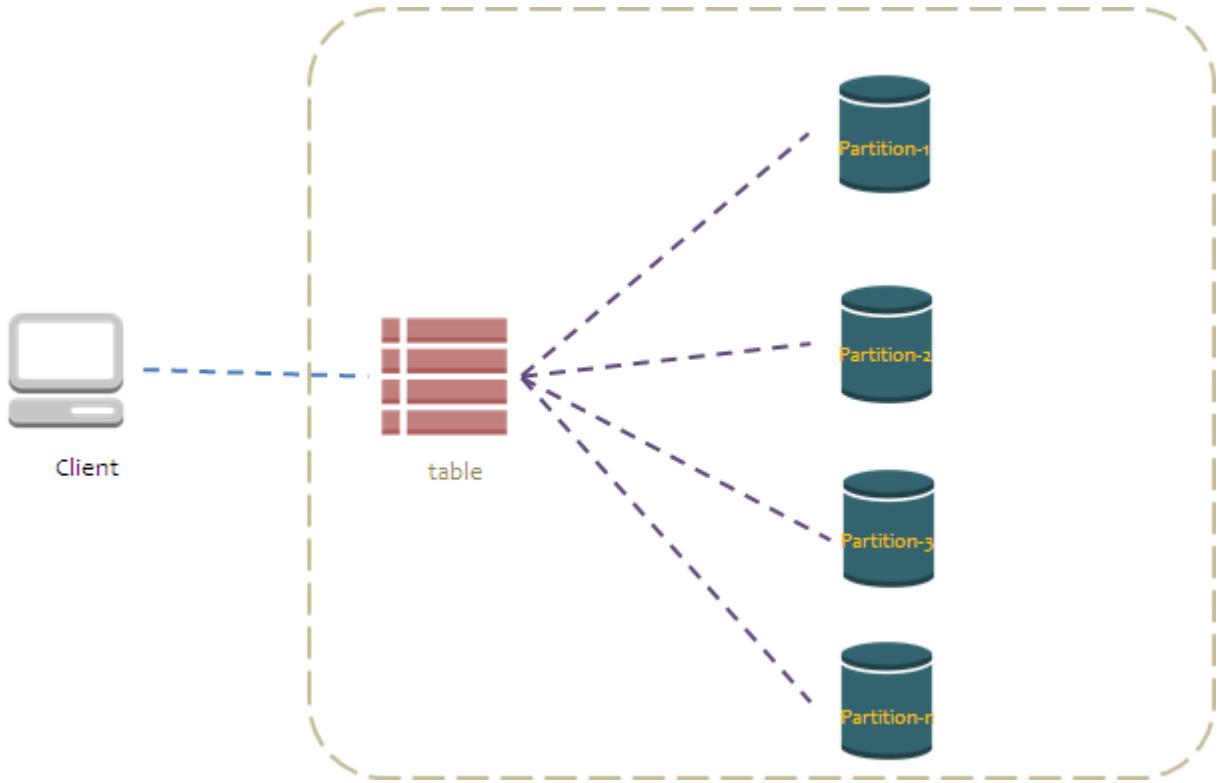
- NoSQL overview
- DynamoDB features
- Basics: Partition key, sort key, indexes
- DynamoDB streams
- Read consistency
- Read/write capacity modes and units
- Backups
- Global table
- DynamoDB TTL
- SQL vs NoSQL

NoSQL databases are increasingly used



What are NoSQL databases?

- NoSQL database is a mechanism for storage and retrieval of data that is modeled **other than the tabular relations** used in relational databases.
- NoSQL databases are widely recognized for their **ease of development**, functionality, and performance at scale.
- NoSQL databases break the traditional mindset of storing data at a single location. Instead, NoSQL distributes and stores data over a set of **multiple servers**.



Why should you use a NoSQL database?

- **Flexibility**: NoSQL databases generally provide flexible schemas that enable faster and more iterative development. The flexible data model makes NoSQL databases ideal for semi-structured and unstructured data.
- **Scalability**: NoSQL databases are generally designed to scale out by using distributed clusters of hardware instead of scaling up by adding expensive and robust servers. Some cloud providers handle these operations behind-the-scenes as a fully managed service.
- **High-performance**: NoSQL database are optimized for specific data models and access patterns that enable higher performance than trying to accomplish similar functionality with relational databases.
- **Highly functional**: NoSQL databases provide highly functional APIs and data types that are purpose built for each of their respective data models.

Types of NoSQL Databases

- **Key-value**: Key-value databases are highly partitionable and allow horizontal scaling at scales that other types of databases cannot achieve. Amazon DynamoDB is designed to provide consistent single-digit millisecond latency for any scale of workloads.
- **Document**: In application code, data is represented often as an object or JSON-like document because it is an efficient and intuitive data model for developers. Amazon DocumentDB (with MongoDB compatibility).
- **Graph**: A graph database's purpose is to make it easy to build and run applications that work with highly connected datasets. Typical use cases for a graph database include social networking, recommendation engines, fraud detection, and knowledge graphs. Amazon Neptune is a fully-managed graph database service.
- **In-memory**: Gaming and ad-tech applications have use cases such as leaderboards, session stores, and real-time analytics that require microsecond response times. Amazon ElastiCache offers Memcached and Redis, to provide low-latency, high-throughput.
- **Search**: Many applications output logs to help developers troubleshoot issues. Amazon Elasticsearch Service (Amazon ES) is purpose built for providing near-real-time visualizations and analytics of machine-generated data by indexing, aggregating, and searching semistructured logs and metrics.

Designing NoSQL database

- It is OK to have duplicate data in your NoSQL table.
- Store every data you see in one page in one table.
- Divide your tables into 3 categories
  - One to one – Store it in one table.
  - One to few (one parent with less than 1000 children records) – Store it in one table.
  - One to many – Store in separate tables. The reason, you have size limit for one item. In DynamoDB, it is 400KB.

Design Considerations

Posts

```
{ _id,
user_id
title,
body,
shares_no
date }
```

users

```
{ _id,
name,
email,
password }
```

comments

```
{ _id,
user_id,
post_id
comment_text
order }
```

post_likes

```
{ post_id,
user_id }
```

Remember that we don't have constrains, so this design will not work as we need to perform too much work (4 joins) in the code to retrieve our data

Design Considerations

```
{
  _id: objectId(),
  user: 'user1', // use it as ID
  title: '',
  body: '',
  shares_no: 0,
  date: ,
  comments: [
     {user:'user2', comment_text:''},
     {user:'user3', comment_text:''}]
  likes: [ 'user1', 'user2']
}
```

This design is optimized for data access pattern so we can access the information much faster with 1 query. Especially that there is no need for data to be updated later.

Amazon DynamoDB

Amazon DynamoDB is a key-value and document database that delivers **single-digit millisecond** performance at **any scale**.

It's a fully managed, multi-region, multi-active, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.

DynamoDB can handle more than 10 trillion requests per day and can support peaks of more than 20 million requests per second.

Many of the world's fastest growing businesses such as Lyft, Airbnb, and Redfin as well as enterprises such as Samsung, Toyota, and Capital One depend on the scale and performance of DynamoDB to support their mission-critical workloads. Snapchat's largest storage write workload, moved to DynamoDB.

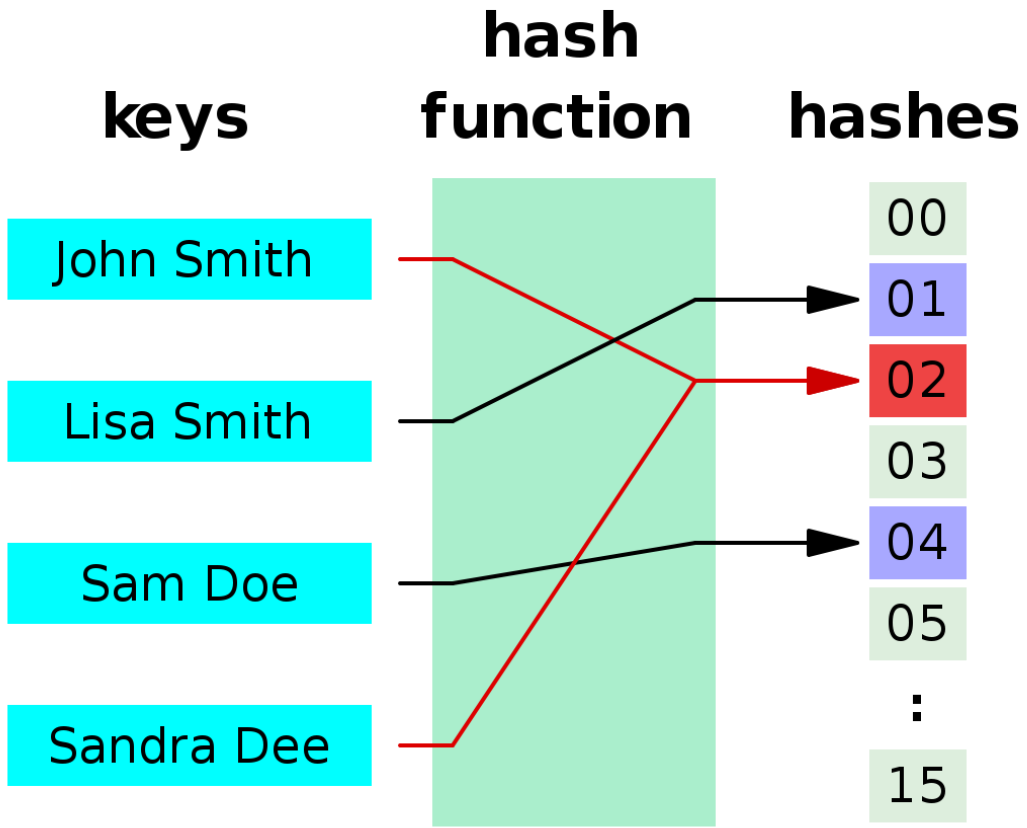
DynamoDB features

- DynamoDB supports both **key-value** and **document** data models. This enables DynamoDB to have a **flexible schema**, so each row can have any number of columns at any point in time.
- **Microsecond latency** with DynamoDB Accelerator (DAX).
- Automated **global replication** with global tables.
- Amazon Kinesis Data Streams for DynamoDB **captures item-level changes** in your DynamoDB tables as a Kinesis data stream. This feature enables you to build advanced streaming applications such as real-time log aggregation, real-time business analytics, and Internet of Things data capture.

Partition key

It is a simple primary key.

DynamoDB uses the partition key's value as input to an internal **hash function**. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.



Sort (Range) key

- There can be duplicate partition keys. We can differentiate the item with a sort key and select one item.
- Also, the sort key is used to represent the one-to-few relationship.
- With the sort key, we can effectively query and improve the speed of writing data.

In this example, there is no throttling when inserting data for the user 2 (U2) with the sort key.

Without sort key

UserID	MessageID
U1	1
U1	2
U1	...
U1	... up to 100
U2	1
U2	2
U2	...
U2	... up to 200

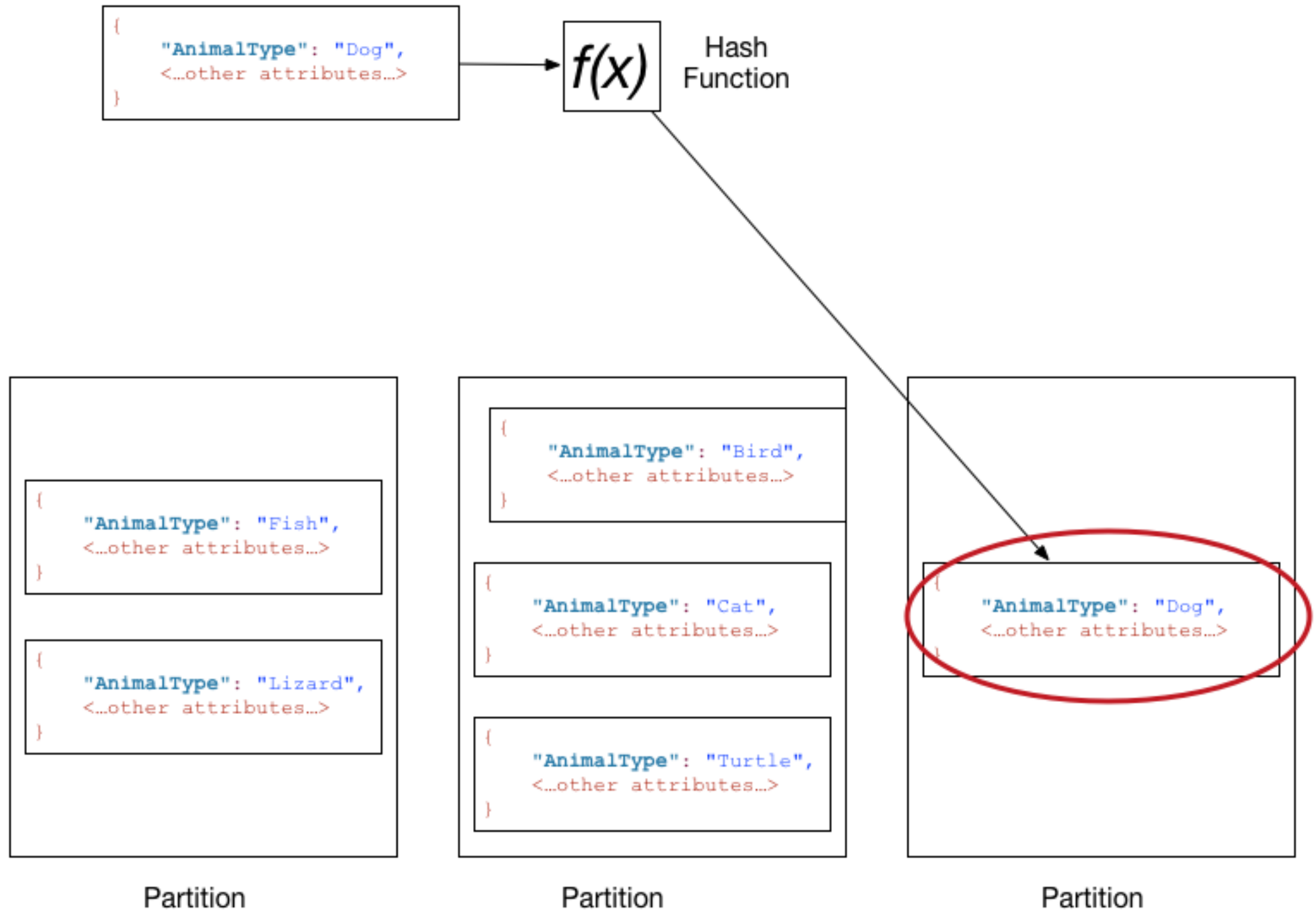
With sort key

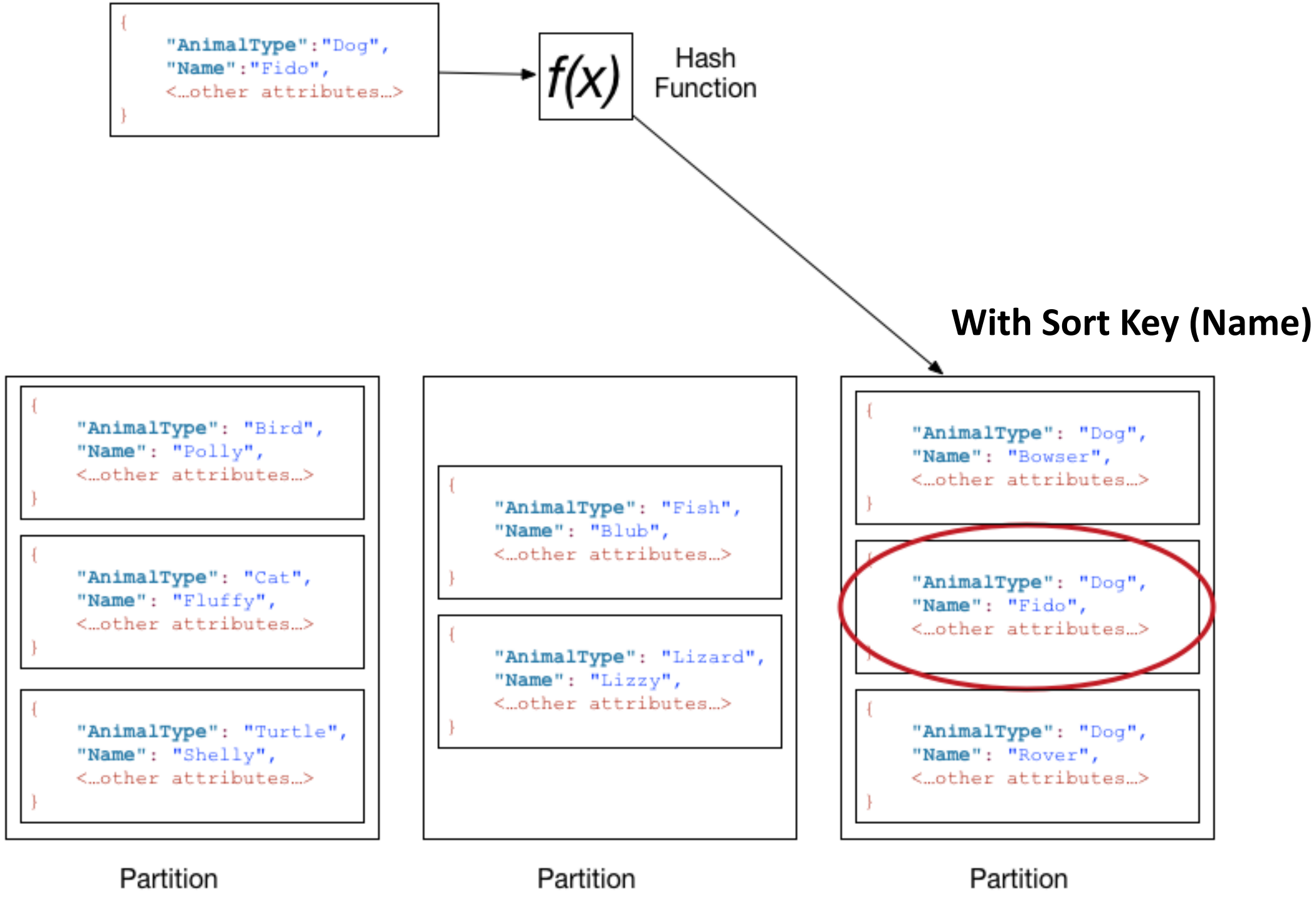
UserID	MessageID
U1	1
U2	1
U3	1
...	...
U1	2
U2	2
U3	2
...	...

Composite primary key

A composite primary key is **composed of two attributes**. The first attribute is the **partition key**, and the second attribute is the **sort key**.

DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored. All items with the same partition key value are stored together, **in sorted order** by sort key value.


{
"AnimalType": "Dog",
<…other attributes…>
}
f(x)
Hash Function
{
"AnimalType": "Fish",
<…other attributes…>
}
{
"AnimalType": "Lizard",
<…other attributes…>
}
{
"AnimalType": "Bird",
<…other attributes…>
}
{
"AnimalType": "Cat",
<…other attributes…>
}
{
"AnimalType": "Turtle",
<…other attributes…>
}
{
"AnimalType": "Dog",
<…other attributes…>
}
Partition
Partition
Partition


{
"AnimalType":"Dog",
"Name":"Fido",
<…other attributes…>
}
f(x)
Hash Function
With Sort Key (Name)
{
"AnimalType": "Bird",
"Name": "Polly",
<…other attributes…>
}
{
"AnimalType": "Cat",
"Name": "Fluffy",
<…other attributes…>
}
{
"AnimalType": "Turtle",
"Name": "Shelly",
<…other attributes…>
}
{
"AnimalType": "Fish",
"Name": "Blub",
<…other attributes…>
}
{
"AnimalType": "Lizard",
"Name": "Lizzy",
<…other attributes…>
}
{
"AnimalType": "Dog",
"Name": "Bowser",
<…other attributes…>
}
{
"AnimalType": "Dog",
"Name": "Fido",
<…other attributes…>
}
{
"AnimalType": "Dog",
"Name": "Rover",
<…other attributes…>
}
Partition
Partition
Partition

Partition key – Artist

Sort key – SongTitle

What if you also wanted to query the data by **Genre** and **AlbumTitle**?

You could create an **index** on Genre and AlbumTitle, and then query the index

Secondary Indexes

A secondary index provides **fast access** and lets you query the data in the table using an **alternate key**, in addition to queries against the primary key.

GameScores

UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
"101"	"Galaxy Invaders"	5842	"2015-09-15:17:24:31"	21	72	…
"101"	"Meteor Blasters"	1000	"2015-10-22:23:18:01"	12	3	…
"101"	"Starship X"	24	"2015-08-31:13:14:21"	4	9	…
"102"	"Alien Adventure"	192	"2015-07-12:11:07:56"	32	192	…
"102"	"Galaxy Invaders"	0	"2015-09-18:07:33:42"	0	5	…
"103"	"Attack Ships"	3	"2015-10-19:01:13:24"	1	8	…
"103"	"Galaxy Invaders"	2317	"2015-09-11:06:53:00"	40	3	…
"103"	"Meteor Blasters"	723	"2015-10-19:01:13:24"	22	12	…
"103"	"Starship X"	42	"2015-07-11:06:53:00"	4	19	…
…	…	…	…	…	…	

GameTitleIndex

GameTitle	TopScore	UserId
"Alien Adventure"	192	"102"
"Attack Ships"	3	"103"
"Galaxy Invaders"	0	"102"
"Galaxy Invaders"	2317	"103"
"Galaxy Invaders"	5842	"101"
"Meteor Blasters"	723	"103"
"Meteor Blasters"	1000	"101"
"Starship X"	24	"101"
"Starship X"	42	"103"
...	...	...

If you need an item (game score) by GameTitle, you would need to **scan** every single record that is inefficient as your table gets bigger.

To speed up, you can create an index with GameTitle along with sort key which is TopScore. The table's **primary key attributes** (partition key and sort key) are always projected into an index.

Choosing the right key

Partition key value	Uniformity
User ID, where the application has many users.	Good
Status code, where there are only a few possible status codes.	Bad
Item creation date, rounded to the nearest time period (for example, day, hour, or minute).	Bad
Device ID, where each device accesses data at relatively similar intervals.	Good
Device ID, where even if there are many devices being tracked, one is by far more popular than all the others.	Bad

Read more: NoSQL best practices

Read Consistency

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 in all nodes.

However, this consistency comes with some disadvantages:

- Strongly consistent reads may have higher latency than eventually consistent reads.
- Strongly consistent reads use more throughput capacity than eventually consistent reads.

Read/write capacity modes

Amazon DynamoDB has two read/write capacity modes for processing reads and writes on your tables.

On-Demand Mode – it is a flexible billing option capable of serving thousands of requests per second without capacity planning. On-demand mode is a good option if there is unknown workloads, unpredictable traffic.

Provisioned Mode – You specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table's provisioned capacity automatically in response to traffic changes.

Read/write capacity units

One **read capacity unit** (RCU) represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. For example,

- an eventually consistent read of an 8 KB item would require one RCU,
- a strongly consistent read of an 8 KB item would require two RCUs,
- a transactional read of an 8 KB item would require four RCUs.

One **write capacity unit** (WCU)represents one write per second for an item up to 1 KB in size.

ACID transactions

Amazon DynamoDB transactions simplify the developer experience of making coordinated, **all-or-nothing** changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, helping you to maintain data correctness in your applications.

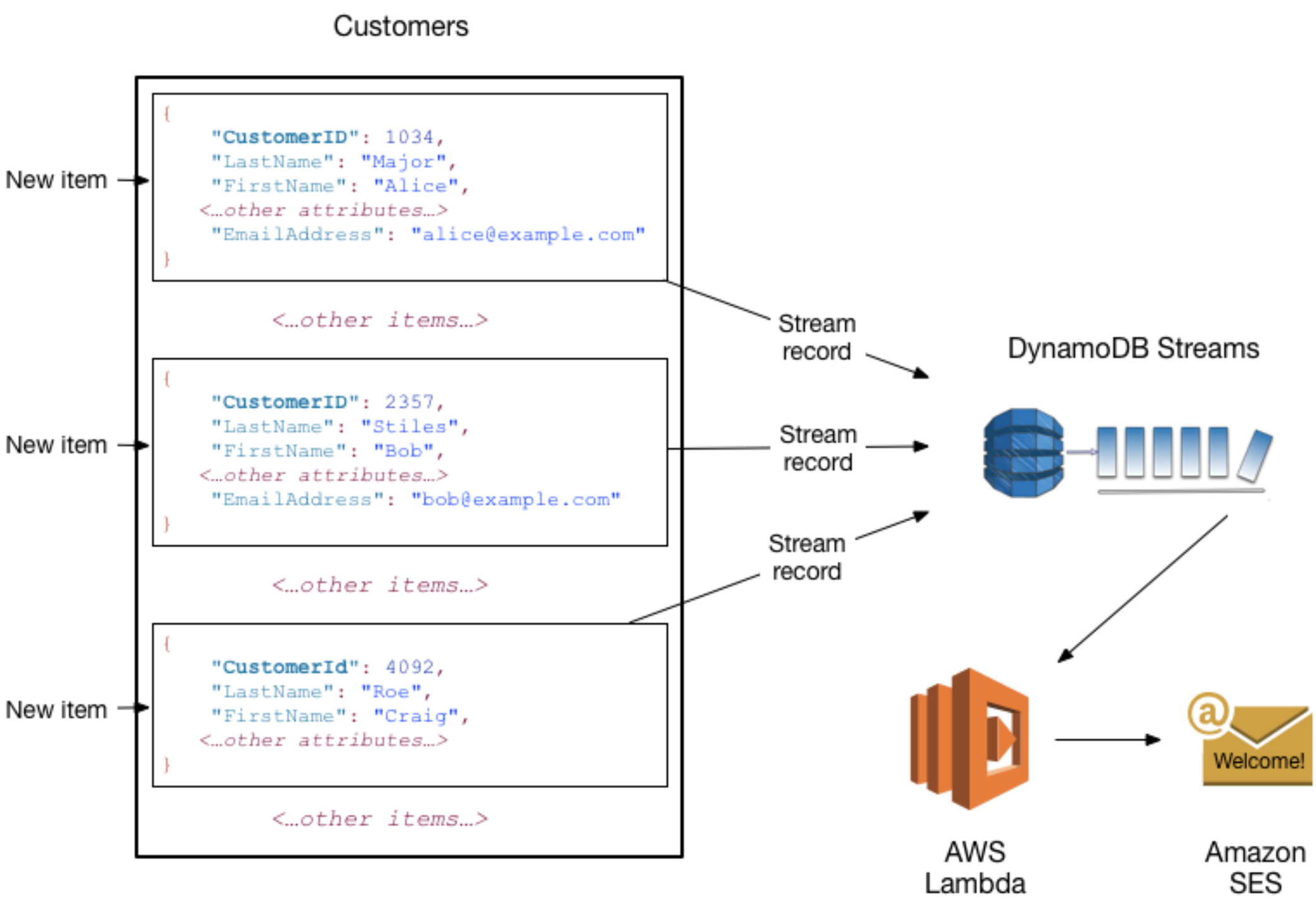
- **TransactWriteItems** - group multiple *Put, Update, Delete*, and *ConditionCheck* actions
- **TransactGetItems** – multiple *Get* actions

DynamoDB Streams

DynamoDB Streams is an optional feature that **captures data modification** events in DynamoDB tables. Near-real time, and events are in order. Events are:

1. A new item is **added**: The stream captures an image of the entire item, including all of its attributes.
2. An item is **updated**: The stream captures the "before" and "after" image of any attributes that were modified in the item.
3. An item is **deleted**: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

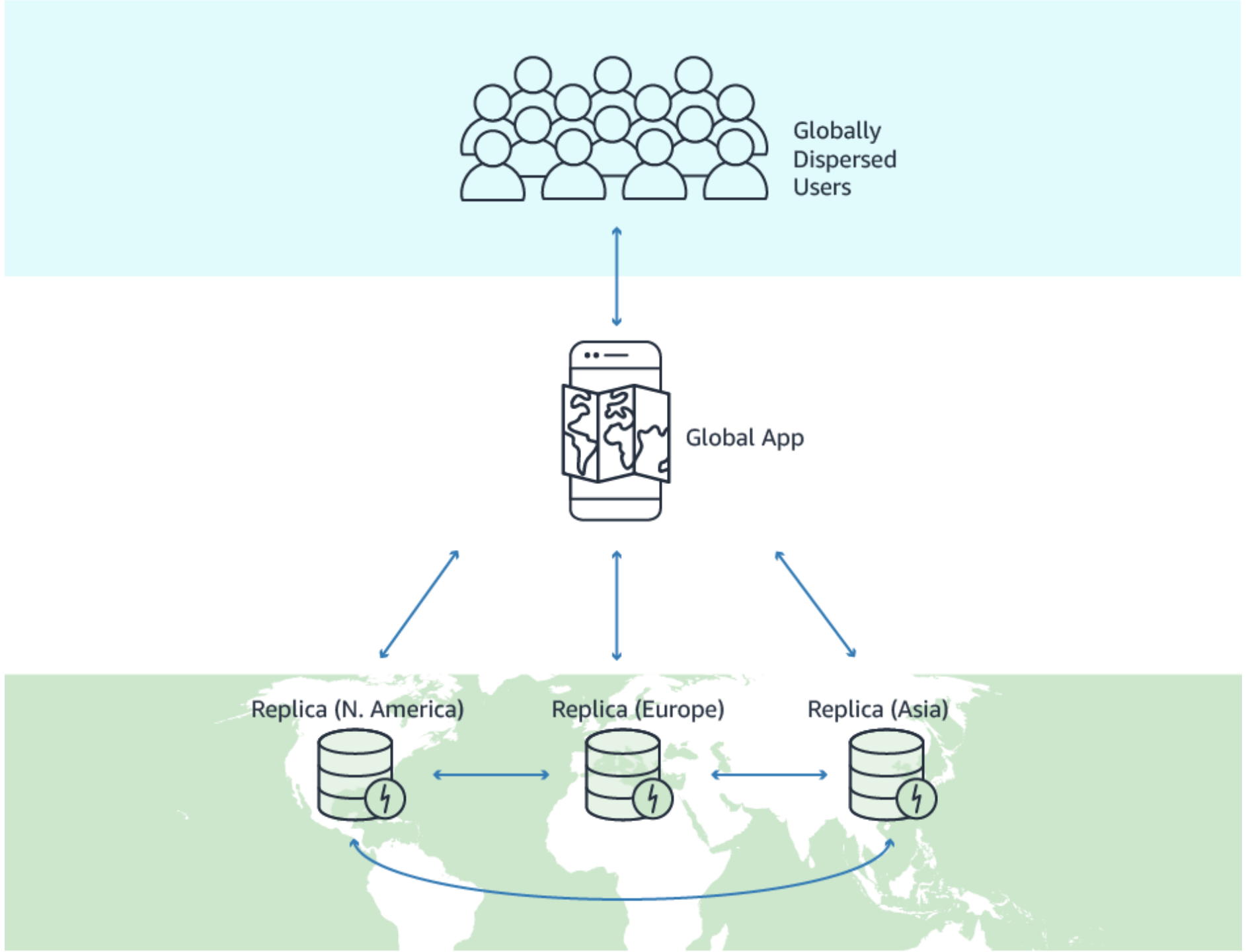

Customers
New item
{
"CustomerID": 1034,
"LastName": "Major",
"FirstName": "Alice",
<…other attributes…>
"EmailAddress": "alice@example.com"
}
<…other items…>
New item
{
"CustomerID": 2357,
"LastName": "Stiles",
"FirstName": "Bob",
<…other attributes…>
"EmailAddress": "bob@example.com"
}
<…other items…>
New item
{
"CustomerId": 4092,
"LastName": "Roe",
"FirstName": "Craig",
<…other attributes…>
}
<…other items…>
Stream record
Stream record
Stream record
DynamoDB Streams
Welcome!
AWS Lambda
Amazon SES

DynamoDB Backups

- Create on-demand backups - You can use the DynamoDB on-demand backup capability to create full backups of your tables for **long-term retention** and **archival** for regulatory compliance needs.
- Enable continuous backups using **point-in-time** recovery - You can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your table.

DynamoDB Global Table

Global tables build on the global Amazon DynamoDB footprint to provide you with a fully managed, **multi-region**, and multi-active database that delivers fast, local, read and write performance for massively scaled, global applications. Global tables **replicate** your DynamoDB tables **automatically** across your choice of AWS Regions.


Globally
Dispersed
Users
Global App
Replica (N. America)
Replica (Europe)
Replica (Asia)

DynamoDB TTL

Amazon DynamoDB Time to Live (TTL) allows you to define a per-item timestamp to determine when an item is no longer needed. Shortly after the specified timestamp, DynamoDB deletes the item from your table without consuming any write throughput.

- Remove user or sensor data after one year of inactivity in an application.
- Archive expired items to an Amazon S3 data lake via Amazon DynamoDB Streams and AWS Lambda.
- Retain sensitive data for a certain amount of time according to contractual or regulatory obligations.