# NoSQL Databases

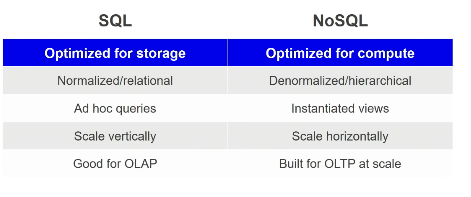
Below are the agenda we need to consider for a NoSQL db design.

* Why NoSQL
* Overview of the database (ex.DynamoDB)
* NoSQL data modeling (Normalized v/s De-normalized schema)
* Common NoSQL design patterns (Composite keys, Hierarchical data, Relational data)
* Modeling real applications

Relational data base stores normalize data which in fact drastically reduces the store cost of data and increases CPU cost (because of complex queries by joining table extremely expensive).

Now the problem reversed the most expensive resource is CPU not storage anymore.

This is why we are moved to NoSQL for more cost effective and easier on wallet.



NoSQL databases don’t like complex queries (inner joins, calculate values) these are good for simple queries

## Amazon DynamoDB properties

* Fully managed NoSQL
* Document or key value
* Scale to any workload
* Fast and Consistent
* Access control
* Event driven programming

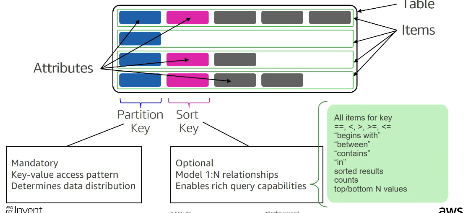
## Table

Table in DynamoDB is like an attributes where we can put many items. Items may or may not have same attributes. They do have an attribute which uniquely identifies the items and that’s the **partition key.**

We can optionally define a **sort key** which gives ability to execute complex range queries against the items in those partitions.

Sort key order the items with in the partition. So when I query the items in the partition I can use complex range operators.

Ex. Partition key might me a customer id and sort key could be order date. So I can query all the customer items in less than 24 hrs.



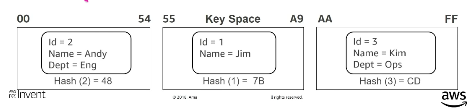
### Partition Keys

Partition key uniquely identifies an item.

Partition key also used to distribute the items across the key space. Every table in DynamoDB defines an key space where we distribute the items by creating unordered hash index from partition key.

So when we **scale** the database we just chop the key space and spread those items across multiple physical devices. So from partition key the system can identify which physical device has that data and go read from there.

This makes all NoSQL database fast and consistent at any scale. There is an automatically routing the queries to exact storage node to serve the request.



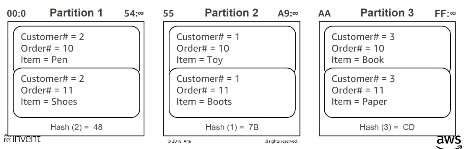
### Partition: Sort Key

Sort key uses two attributes together to uniquely identify an item.

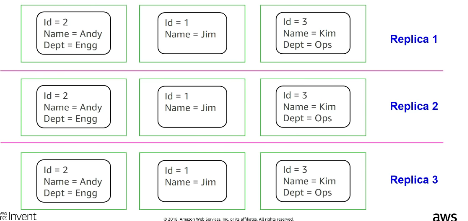
Within unordered hash index data is arranged by the sort key.

No limits on the number of items (infinite) per partition key.

This is how all NoSQL data bases maintain that fast and consistent behavior.



Partitions are **three-way replicated**.



When comes to read in DynamoDB we have

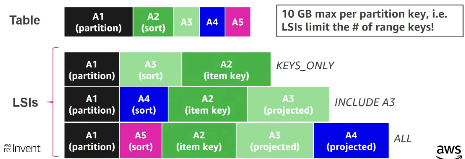
* Strongly consistent (read will happen from Master node).
* Eventually Consistent (recommended – as millisecond different and cost optimized).

## Index in DynamoDB

### Local Secondary Index (LSI)

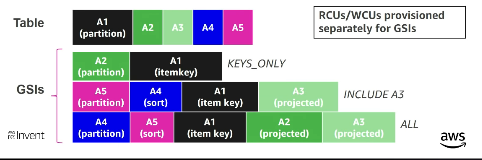
LSI is used to resort the data inside the partition. LSI should use the same partition key as the table. So it’s a way to resort the data but not re-group the data.

This is local to a partition key.



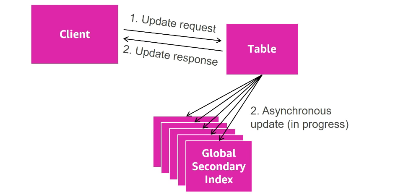
### Global Secondary Index (GSI)

GSI allow to create completely new aggregation of data. Primary table may groups the table by “customer” but GSI groups by “warehouse”.



**Note:**

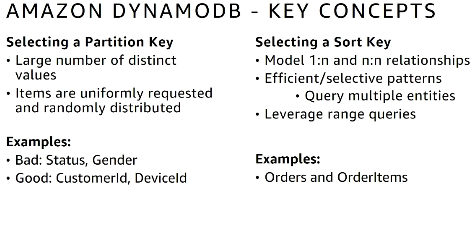
GSI updates are eventually consistent and LSI updates are strongly consistent.



## Getting the most out of Amazon DynamoDB throughput

To get the most out of DynamoDB throughput, create tables where the partition key element has a large number of distinct values and values are requested fairly uniformly as random as possible.

**Space:** access is evenly spread over the key-space.



## Tenets of NoSQL data modeling

1. Understand the use case
   * Nature of the application (OLTP / OLAP / DSS)
   * Define the Entity-Relationship model
   * Identify the data lifecycle (TTL, Backup archival etc.)
2. Define all of the access patterns (Read/Write workloads)
   * Identify data sources
   * Define query aggregations
   * Document all workflows
3. Data modeling
   * Avoid relational design patterns, use one table
   * 1 application service = 1table (reduces round trips, simplify access pattern)
   * Identify primary keys (How will be items inserted and read)
   * Define indexes for secondary access patterns
4. Review -> Repeat -> Review

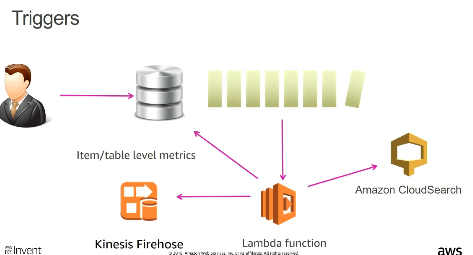
## DynamoDB streams and AWS Lambda

This is a completely replaceable for store procedure and best store procedure engine in business because it completely disconnected from table space.

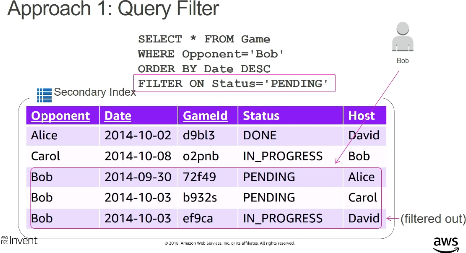
Stream is the **change log** for the DynamoDB table. And once data is into stream we can invoke Lambda functions and Lambda has 2 IAM roles **Invocation role** (what it can see, which items can read) and **Execution role** (what it can do, what other services within our AWS account space it has access to)

One of the common thing people do with stream and Lambda is compute aggregation.

Lambda can update Amazon Cloud Search/Elastic Search



## Composite keys



In the above case the FILTER is applied after read. So the cost of read is same with or without FILTER.

