**NoSQL-and-DynamoDB**

**DynamoDB Architecture**

NoSQL Database as a Service (DBaaS)

* Wide column Key/Value database.
* Not like RDS which is a Database Server as a Product.
  + This is only the database.
* Capacity can be **provisioned** or **use on-demand mode**
* Highly resilient across AZs and optionally globally resilient.
* Data is **replicated across multiple storage nodes by default.**
* Really fast, single digit millisecond access to data.
* Supports backups with **point in time recovery and encryption at rest.**
* Allows **event-driven integration**. Do things when data changes.

**Dynamo DB Tables**

* **Table** a grouping of items which share the same primary key.
* **Items** within a table are how you manage the data.
  + There is no limit to the number of items in a table.
* Two types of primary key:
  + **Simple** (Partition)
  + **Composite** (Partition and Sort)
* Every item in the table needs a unique primary key.
* Attributes may or may not be there. This is not necessary.
* Items can be at most **400KB** in size. This includes the **primary key and attributes.**

In DynamoDB, **capacity means speed**. If you choose on-demand capacity model you don't have to worry about capacity. You only pay for the operations for the table. If you choose provisioned capacity, you must set this on a per table basis.

Capacity is set **per WCU or RCU**

1 WCU means you can write 1KB per second to that table 1 RCU means you can read 4KB per second for that table

**Dynamo DB Backups**

**On-demand Backups**: Similar to manual RDS snapshots. Full back up of the table that is retained until you manually remove that backup. This can be used to restore data in the same region or cross-region. You can adjust indexes, or adjust encryption settings.

**Point-in-time Recovery**: Must be enabled on each table and is off by default. This allows continuous record of changes for 35 days to allow you to replay any point in that window to a 1 second granularity.

**Dynamo DB Considerations**

* NoSQL, you should jump towards DynamoDB.
* Relational data, this is NOT DynamoDB.
* If you see key value and DynamoDB is an answer, this is likely the proper choice.

Access to Dynamo is from the console, CLI, or API. You don't have SQL access.

**Billing based on:**

* RCU and WCU
* Storage on that table
* Additional features on that table

Can purchase reserved capacity with a cheaper rate for a longer term commit.

**DynamoDB Operations, Consistency, and Performance**

**DynamoDB Reading and Writing**

**On-Demand**: Unknown or unpredictable load on a table. This is also good for as little admin overhead as possible. Pay a price per million Read or Write units. This is as much as 5 times the price as provisioned.

**Provisioned**: RCU and WCU set on a per table basis.

Every operation consumes at least 1 RCU/WCU

1 RCU = 1 x 4KB read operation per second. This rounds up. 1 WCU = 1 x 1KB write operation per second.

Every single table has a WCU and RCU burst pool. This is 500 seconds of RCU or WCU as set by the table.

**Query**

You have to pick one Partition Key (PK) value to start.

The PK can be the sensor unit, the Sort Key (SK) can be the day of the week you want to look at.

Query accepts a single PK value and **optionally** a SK or range. Capacity consumed is the size of all returned items. Further filtering discards data, but capacity is still consumed.

In this example you can only query for one weather station.

If you query a PK it can return all fields items that match. It is always more efficient to pull as much data as needed per query to save RCU.

You have to query for at least one item of PK and are charged for the response of that query operation.

**If you filter data and only look at one attribute, you will still be charged for pulling all the attributes against that query.**

**Scan**

Least efficient when pulling data from Dynamo, but the **most flexible.**

Scan moves through the table item by item consuming the capacity of every item. Even if you consume less than the whole table, it will charge based on that. It adds up all the values scanned and will charge rounding up.

**DynamoDB Consistency Model**

* **Eventually** Consistent: easier to implement and scales better
* **Strongly (Immediately)** Consistent: more costly to achieve

Every piece of data is replicated between storage nodes. There is one Leader storage node and every other node follows.

Writes are always directed to the **leader node**. Once the leader is complete, it is **consistent**. It then starts the process of replication. This typically takes milliseconds and assumes the lack of any faults on the storage nodes.

Eventual consistent could lead to stale data if a node is checked before replication completes. You get a discount for this risk.

A strongly consistent read always uses the leader node and is less scalable.

Not every application can tolerate eventual consistency. If you have a stock database or medical information, you must use strongly consistent reads. If you can tolerate the cost savings you can scale better.

**WCU Example Calculation**

* Store 10 items per second with 2.5K average size per item.
* Calculate WCU per item, round up, then multiply by average per second.
* (2.5 KB / 1 KB) = 3 \* 10 p/s = 30 WCU

**RCU Example Calculation**

* Retrieve 10 items per second with 2.5K average size per item.
* Calculate RCU per item, round up, then multiply by average per second.
* (2.5 KB / 4 KB) = 1 \* 10 p/s = 10 RCU for strongly consistent.
  + 5 RCU for eventually consistent.

**DynamoDB Streams and Triggers**

DynamoDB stream is a time ordered list of changes to items in a DynamoDB table. A stream is a **24 hour rolling window of the changes**. It uses Kinesis streams on the backend.

This is enabled on a per table basis. This records

* Inserts
* Updates
* Deletes

Different view types influence what is in the stream.

There are four view types that it can be configured with:

* **KEYS\_ONLY :** only shows the item that was modified
* **NEW\_IMAGE :** shows the final state for that item
* **OLD\_IMAGE :** shows the initial state before the change
* **NEW\_AND\_OLD\_IMAGES :** shows both before and after the change

Pre or post change state might be empty if you use **insert** or **delete**

**Trigger Concepts**

Allow for actions to take place in the event of a change in data

**Item change generates an event that contains the data which was changed**. The specifics depend on the view type. The action is taken using that data. This will combine the capabilities of stream and lambda. **Lambda will complete some compute based on this trigger.**

This is great for reporting and analytics in the event of changes such as stock levels or data aggregation. Good for data aggregation for stock or voting apps. This can provide messages or notifications and eliminates the need to poll databases.

**DynamoDB Local (LSI) and Global (GSI) Secondary Indexes**

* Great for improving data retrieval in DynamoDB.
* Query can only work on 1 PK value at a time and optionally a single or range of SK values.
* Indexes are a way to provide an **alternative view on table data.**
* You have the ability to **choose which attributes are projected to the table.**

**Local Secondary Indexes (LSI)**

* Choose **alternative sort key** with the **same partition key** on base table data.
  + If item does not have sort key it will not show on the table.
* These must be created with a base table in the beginning.
  + This **cannot be added later.**
* Maximum of **5 LSIs per base table.**
* Uses the **same partition key, but different sort key.**
* **Shares the RCU and WCU with the table.**
* It **makes a smaller table** and makes **scan** operates easier.
* In regards to Attributes, you can use:
  + **ALL**
  + **KEYS\_ONLY**
  + **INCLUDE(Specifically pick attributes)**

**Global Secondary Index (GSI)**

* Can be created at **any time and much more flexible.**
* There is a default limit of **20 GSIs for each table.**
* Allows for **alternative PK and SK.**
* GSI will have their **own RCU and WCU allocations.**
* You can then choose which attributes are included in this table.
* GSIs are **always eventually consistent.** Replication between base and GSI is Async

**LSI and GSI Considerations**

* Must be careful which projections are used to manage capacity.
* If you don't project a specific attribute, then you require the attribute when querying data, it will then fetch the data later in an inefficient way.
* This means you should try to plan what will be used on the front.

**GSI as default** and only use LSI when **strong consistency** is required

**Indexes are designed when data is in a base table needs an alternative access pattern.** This is great for a security team or data science team to look at other attributes from the original purpose.

**DynamoDB Global Tables**

* Global tables provide **multi-master cross-region replication.**
  + All tables are the same.
* Tables are **created in multiple AWS regions.** In one of the tables, you configure the links between all of the tables.
* DynamoDB will enable replication between all of the tables.
  + Tables become table replicas.
* Between the tables, **last writer wins** in conflict resolution.
  + DynamoDB will pick the most recent write and replicate that.
* **Reads and Writes can occur to any region** and are replicated within a second.
* Strongly Consistent Reads **only** in the same region as writes.
  + **Application should allow for eventual consistency** where data may be stale.
  + Replication is generally sub-second and depends on the region load.
* Provides Global HA and disaster recovery or business continuity easily.

**DynamoDB Accelerator (DAX)**

This is an in memory cache for Dynamo.

**Traditional Cache**: The application needs to access some data and checks the cache. If the cache doesn't have the data, this is known as a cache miss. The application then loads directly from the database. It then updates the cache with the new data. Subsequent queries will load data from the cache as a cache hit and it will be faster

**DAX**: The application instance has DAX SDK added on. DAX and dynamoDB are one in the same. Application uses DAX SDK and makes a single call for the data which is returned by DAX. If DAX has the data, then the data is returned directly. If not it will talk to Dynamo and get the data. It will then cache it for future use. The benefit of this system is there is only one set of API calls using one SDK. It is tightly integrated and much less admin overhead.

**DAX Architecture**

This runs from within a VPC and is designed to be deployed to multiple AZs in that VPC. Must be deployed across AZs to ensure it is highly available.

DAX is a cluster service where nodes are placed into different AZs. There is a **primary node** which is the read and write note. This replicates out to other nodes which are **replica nodes** and function as read replicas. With this architecture, we have an EC2 instance running an application and the DAX SDK. This will communicate with the cluster. On the other side, the cluster communicates with DynamoDB.

DAX maintains two different caches. First is the **item cache** and this caches individual items which are retrieved via the **GetItem** or **BatchGetItem** operation. These operate on single items and must specify the items partition or sort key.

There is a **query cache** which holds data and the parameters used for the original query or scan. Whole query or scan operations can be rerun and return the same cached data.

Every DAX cluster has an endpoint which will load balance across the cluster. If data is retrieved from DAX directly, then it's called a cache hit and the results can be returned in microseconds.

Any cache misses, so when DAX has to consult DynamoDB, these are generally returned in single digit milliseconds. Now in writing data to DynamoDB, DAX can use write-through caching, so that data is written into DAX at the same time as being written into the database.

If a cache miss occurs while reading, the data is also written to the primary node of the cluster and the data is retrieved. And then it's replicated from the primary node to the replica nodes.

When writing data to DAX, it can use write-through. Data is written to the database, then written to DAX.

**DAX Considerations**

* Primary node which writes and Replicas which support read operations.
* Nodes are HA, if the primary node fails there will be an election and secondary nodes will be made primary.
* In-memory cache allows for much faster read operations and significantly reduced costs. If you are performing the same set of read operations on the same set of data over and over again, you can achieve performance improvements by implementing DAX and caching those results.
* With DAX you can scale up or scale out.
* DAX supports write-through. If you write data to DynamoDB, you can use the DAX SDK. DAX will handle that data being committed to DynamoDB and also storing that data inside the cache.
* DAX is not a public service and is deployed within a VPC. Anything that uses that data many times will benefit from DAX.
* Any questions which talk about caching with DynamoDB, assume it is DAX.

**Amazon Athena**

* You can take data stored in S3 and perform Ad-hoc queries on data. Pay only for the data consumed.
* Start off with structured, semi-structured and even unstructured data that is stored in its raw form on S3.
* Athena uses **schema-on-read**, the original data is never changed and remains on S3 in its original form.
* The schema which you define in advance, modifies data in flight when its read.
* Normally with databases, you need to make a table and then load the data in.
* With Athena you create a schema and load data on this schema on the fly in a relational style way without changing the data.
* The output of a query can be sent to other services and can be performed in an event driven fully serverless way.

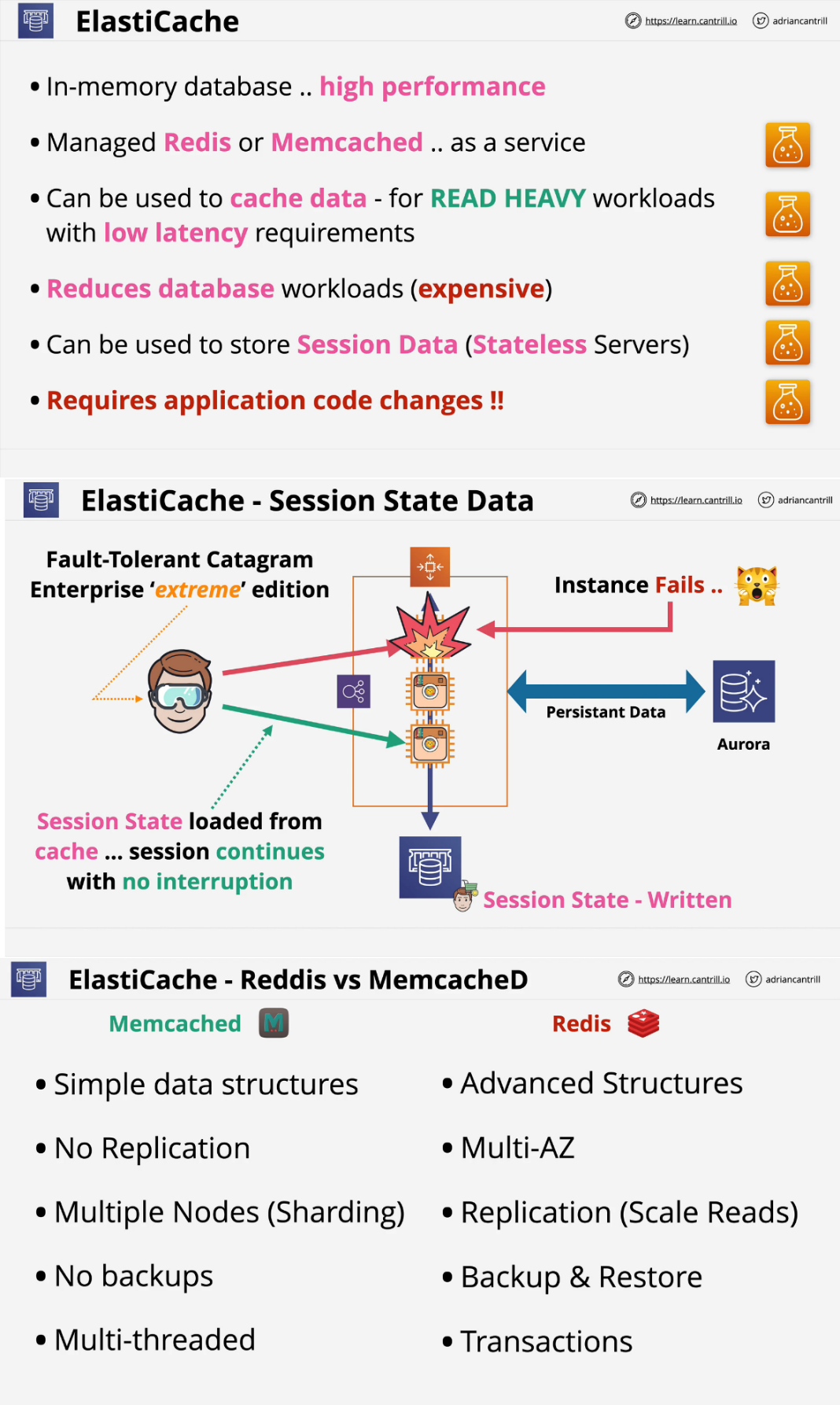
**Athena Explained**

The source data is stored on S3 and Athena can read from this data. In Athena you are defining a way to get the original data and defining how it should show up for what you want to see.

Tables are defined in advance in a data catalog and data is projected through when read. It allows SQL-like queries on data without transforming the data itself.

This can be saved in the console or fed to other visualization tools.

You can optimize the original data set to reduce the amount of space uses for the data and reduce the costs for querying that data.

**Elasticache:**