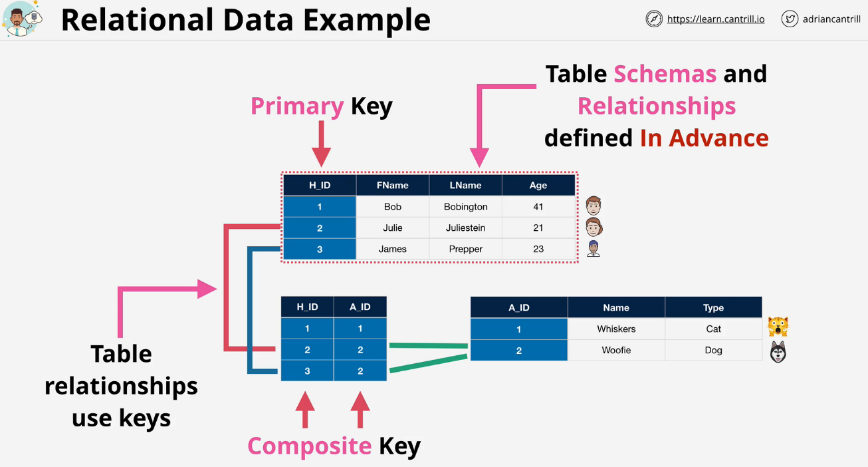
**Relational-Database-Service(RDS)**

### Database Refresher

Systems to store and manage data.

#### Relational (SQL)

* Structured Query Language (SQL) is a feature of most RDS.
* Structure to the data known as a **schema**.
  + Defined in advance.
  + Defines names of things
  + Valid values of things
  + Types of data which is stored and where
* Fixed relationship between tables.
  + This is defined before data is entered into the database.

Every row in a table must have a value for the **primary key**. There **must be a value stored for every attribute in the table.**

SQL systems are relational so we generally **define relationships between tables as well.** This is defined with a **join table**. A join table has a **composite key** which is a key formed of two parts. Composite keys together must be unique.

**Keys** in different tables are **how the relationships between the tables are defined.**

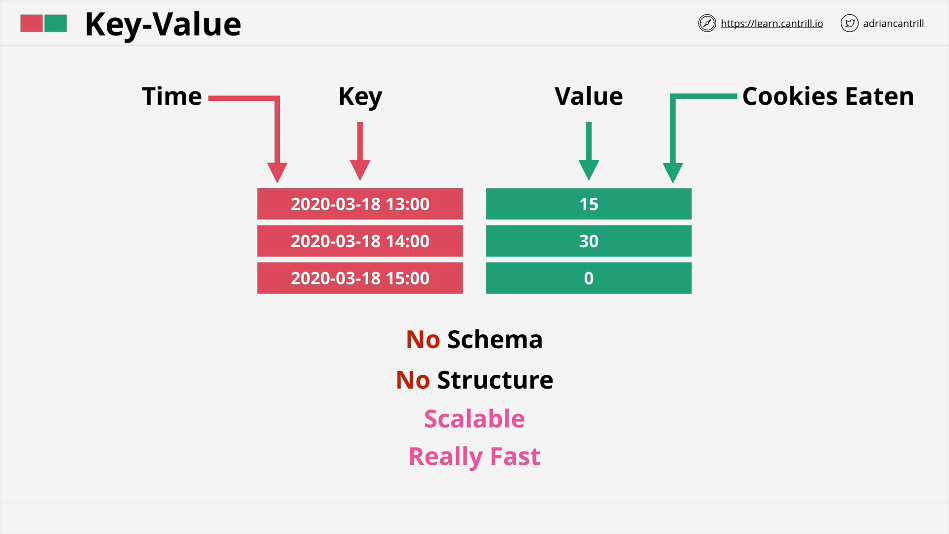
The Table schema and relationships **must be defined in advance** which can be hard to do.

#### Non-Relational (NoSQL)

Not a single thing, and is a catch all for everything else. There is generally no schema or a weak one.

* Key-Value
* Wide Column
* Document
* Column Databases
* Graph

##### **Key-Value databases**

This is just a list of keys and value pairs. So long as every key is unique, there is no real schema or structure needed. These are really **fast** and **highly scalable**. This is also used for **in memory caching**.

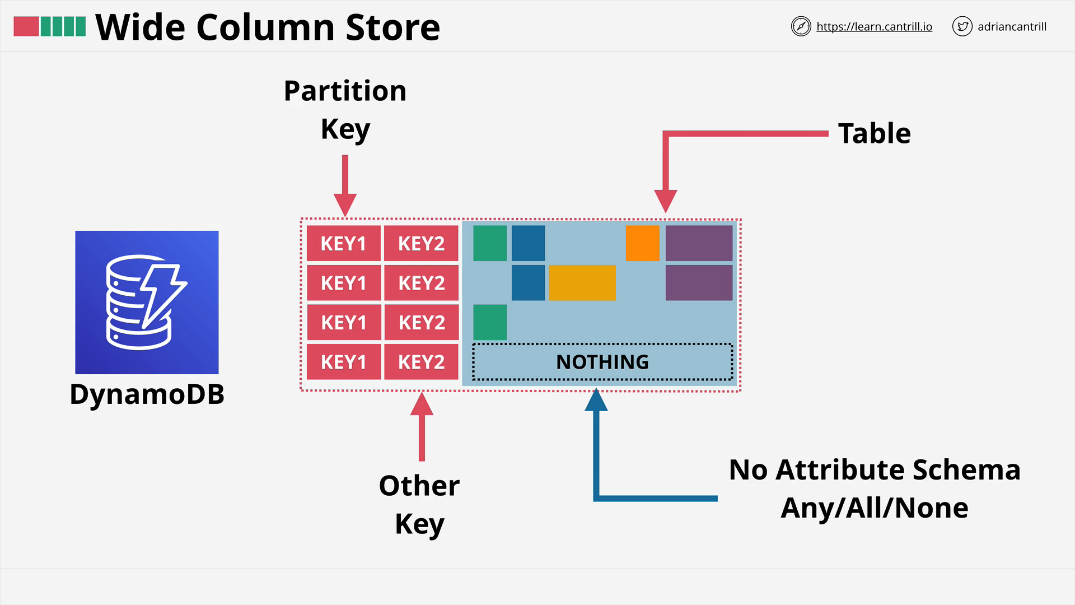
##### **Wide Column Store**

**DynamoDB** is an example of wide column store database.

Each row or item has one or more keys. One key is called the **partition key**.

You can have additional keys other than the partition key called the **sort or range key**.

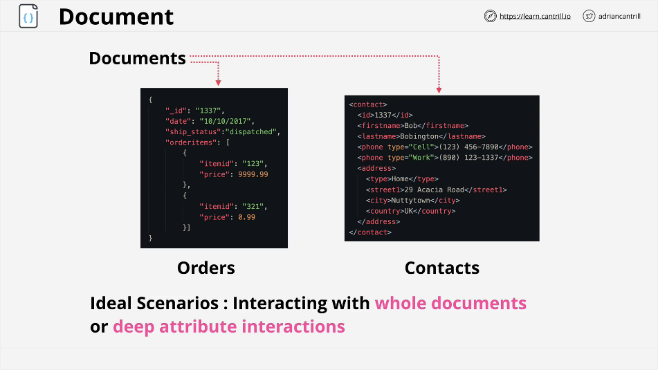
It can be **single key** (only partition key) or **composite key** (partition key and sort key).

Every item in a table can also have attributes, but they don't have to be the same between values. The only requirements is that every item inside the table has to use the **same key structure and it has to have a unique key**.

##### **Document**

Documents are generally **formatted using JSON or XML.**

This is an extension of a key-value store where each document is interacted with via an ID that's unique to that document, but the **value of the document contents are exposed to the database allowing you to interact with it**.

Good for order databases, or collections, or contact stale databases.

Great for nested data items within a **document structure such as user profiles**.

##### **Row Database (MySQL)**

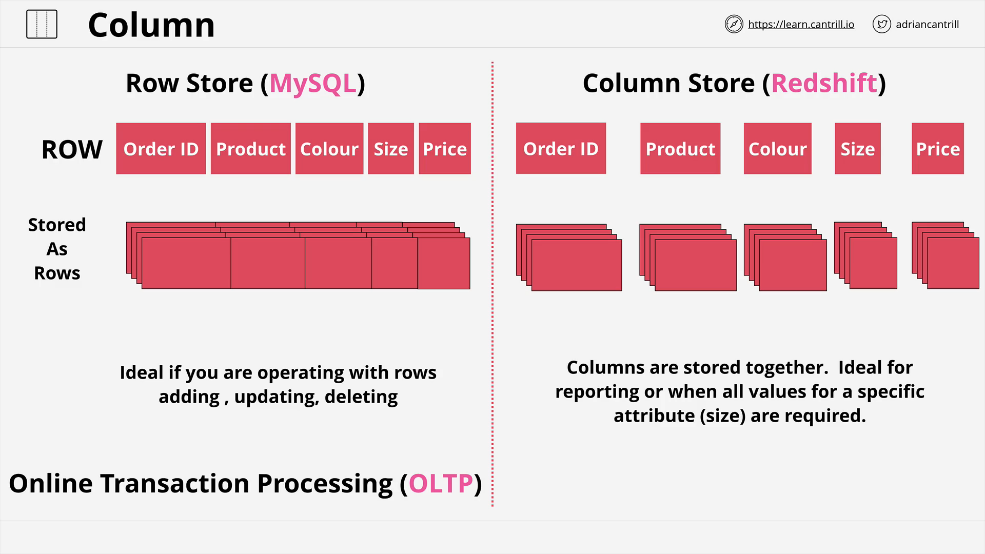
Often called OLTP (**Online Transactional Processing Databases**).

If you needed to read the price of one item you need that row first. If you wanted to query all of the sizes of every order, you will need to check for each row.

Great for things which deal in **rows and items where they are constantly accessed, modified, and removed.**

##### **Column Database (Redshift)**

Instead of storing data in rows on disk, they **store it based on columns**. The data is the same, but it's grouped together on disk, based on column so every order value is stored together, every product item, color, size, and price are all grouped together.

This is **bad for transactional style processing**, but **great for reporting or when all values for a specific size are required.**

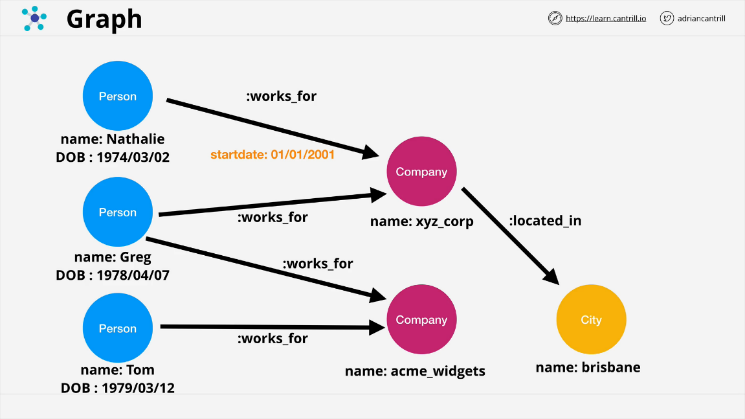
##### **Graph**

**Relationships between things are formally defined and stored** along in the database itself with the data. They are not calculated each and every time you run a query. These are **great for relationship driven data.**

**Nodes** are objects inside a graph database. They can have **properties.**

**Edges are relationships** between the nodes. They have a direction.

**Relationships themselves can also have attached data**, so name value pairs. We might want to **store the start date of any employment relationship**.

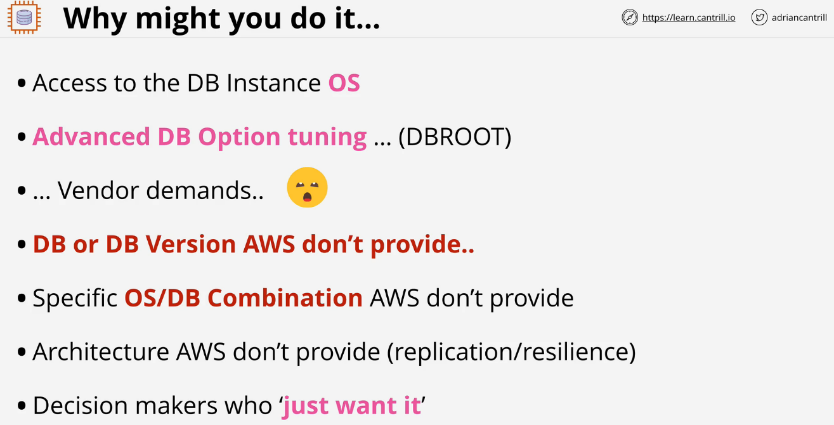
Can store massive amounts of complex relationships between data or between nodes in a database.

### Databases on EC2

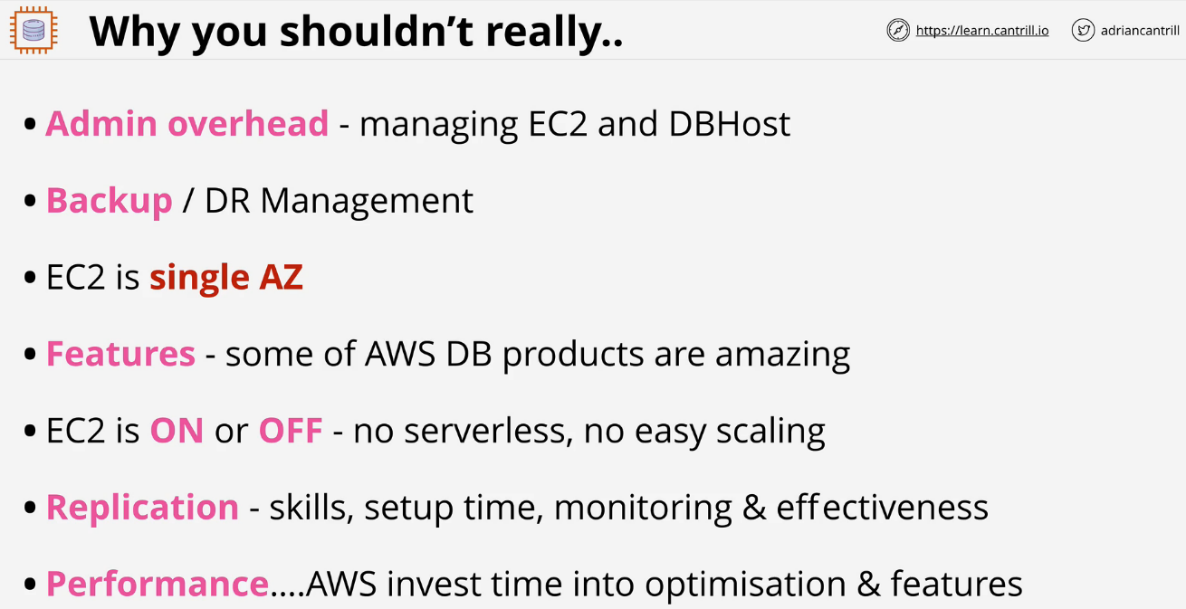
It is always a **bad idea** to do this.

* **Splitting an instance over different AZs**
  + **Adds reliability consideration** between the AZs
  + **Adds a cost to move the data** between AZs

#### Reasons EC2 Database might make sense

* Need **access to the OS of the Database**.
  + You should question if a client requests this, it rarely is needed.
* Advanced **DB Option tuning (DBROOT)**
  + AWS provides options to tune many of these parameters anyways.
  + Can be a vendor that is asking for this.
* DB or DB version that **AWS doesn't provide**.
* You might need a **specific version of an OS and DB that AWS doesn't provide.**

#### Reasons why you really shouldn't run a database on EC2

* **Admin overhead** is intense to manage the EC2 host.
* **Backup and Disaster Management** adds complexity.
* EC2 is running in one AZ. If the **zone fails, access to the database fails**.
* Will **miss out on features from AWS DB products.**
* EC2 is ON or OFF, there is **no way to scale easily.**
* **Replication** can be tricky to manage on your own.
* **Performance will be slower** than other AWS options.

**Relational Database Service (RDS)**

* Database-as-a-service (DBaaS)
  + Not entirely true more of **DatabaseServer-as-a-service**.
  + **Managed Database Instance** for one or more databases.
* **No need to manage the HW** or server itself.
* Handles engines such as **MySQL, MariaDB, PostgreSQL, Oracle, Microsoft SQL.**

**Amazon Aurora. This is so different from normal RDS, it is a separate product.**

**RDS Database Instance**

Runs one of a few types of database engines and can contain multiple user created databases. Create **one when you provision the instance**, but **multiple ones can be created after.**

When you create a database instance, the way you access it is using a **database host-name**, a **CNAME**, and this resolves to the database instance itself.

RDS uses standard database engines so you can **access an RDS instance using the same tooling** as if you were accessing a self-managed database.

The database can be optimized for:

**db.m5 general**

**db.r5 memory**

**db.t3 burst**

There is an **associated size and AZ selected.**

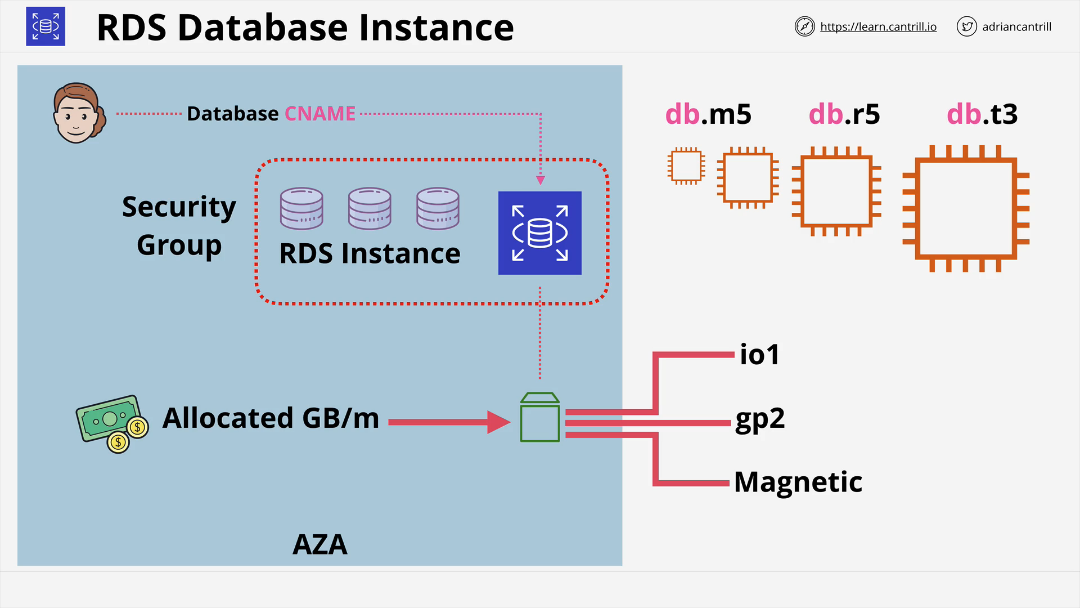
When you provision an instance, you **provision dedicated storage to that instance.** This is **EBS storage located in the same AZ**.

RDS is **vulnerable to failures in that AZ**.

The storage can be allocated with **SSD or magnetic.**

**io1** - lots of IOPS and consistent low latency

**gp2** - same burst pool architecture as it does on EC2, used by default magnetic - compatibility mostly for long term historic uses

**Billing** is **per instance** and **hourly rate** for that **compute**. You are **billed for storage allocated**.

**RDS Multi AZ (High-Availability)**

This is an **option** that you can enable on RDS instances. **Secondary hardware** is allocated inside **another AZ**. This is referred to as the **standby replica** or standby replica instance.

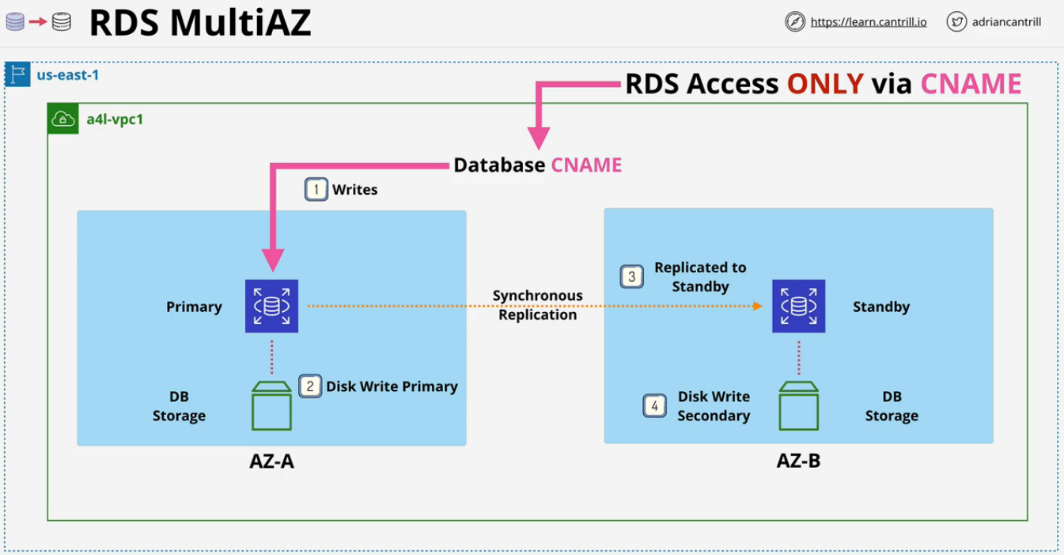
The **standby replica has its own storage in the same AZ as it's located.**

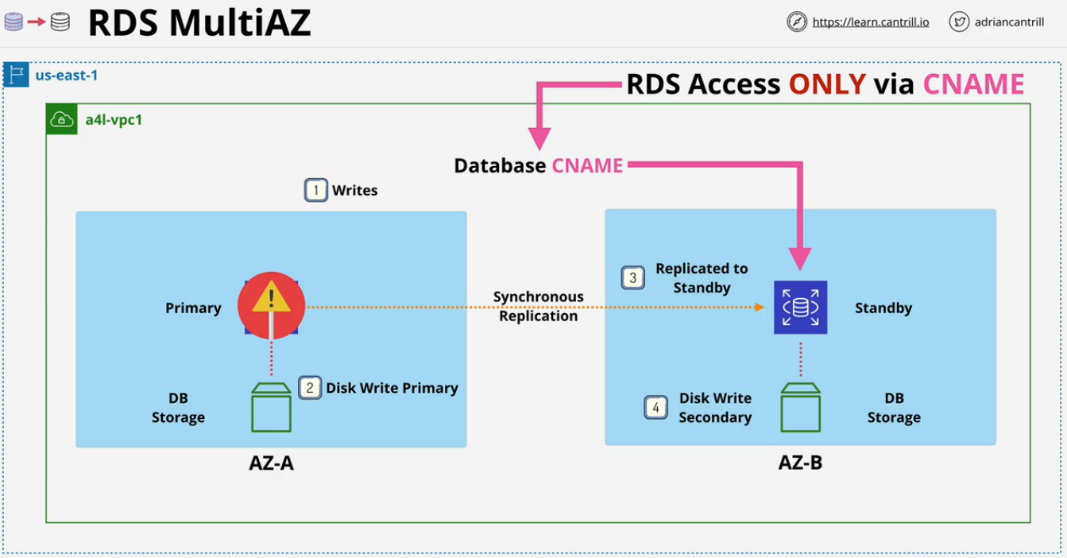
RDS enables **synchronous replication** from the primary instance to the standby replica.

RDS Access ONLY via database CNAME. The CNAME will point at the primary instance. You **cannot access the standby replica for any reason** via RDS.

The standby replica **cannot be used for extra capacity.**

**Synchronous Replication** means:

1. Database writes happen.
2. Primary database instance commits changes.
3. Same time as the write is happening, standby replication is happening.
4. Standby replica commits writes.

If any **error occurs with the primary database, AWS detects this and will failover within 60 to 120 seconds to change to the new database.**

This **does not provide fault tolerance as there will be some impact during change**.

**RDS Exam PowerUp**

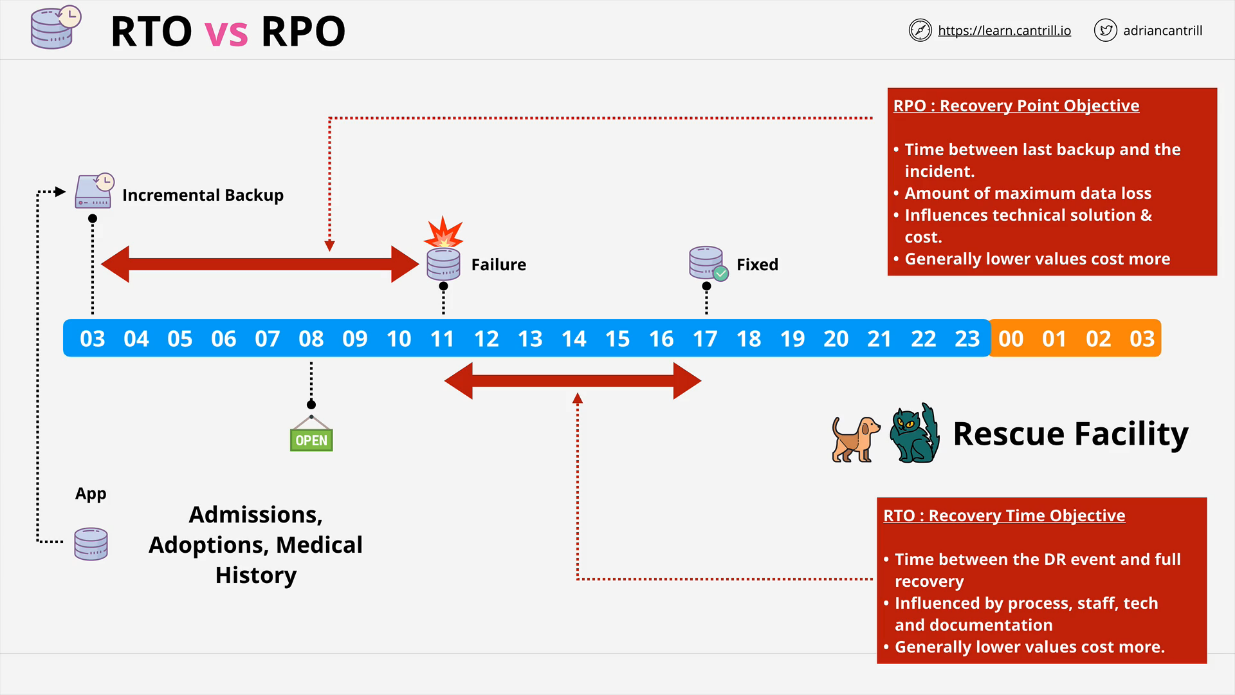
* Multi-AZ feature is not free tier, extra infrastructure for standby.
  + **Generally two times the price.**
* The standby replica **cannot be accessed directly unless a fail occurs.**
* Failover is highly available, not fault tolerant.
* **Same region only** (others AZ in the VPC).
* Backups are taken from standby which removes performance impacts.
* **Failover** can happen for a number of reasons.
  + **Full AZ outage**
  + **Primary RDS failure**
  + **Manual failover** for testing
  + If you **change the type of a RDS instance, it will failover as part of changing that type.**

**RDS Backup and Restores**

**RPO - Recovery Point Objective**

* Time between the last backup and when the failure occurred.
* Amount of maximum data loss.
* Influences technical solution and cost.
* Business usually provides an RPO value.

**RTO - Recovery Time Objective**

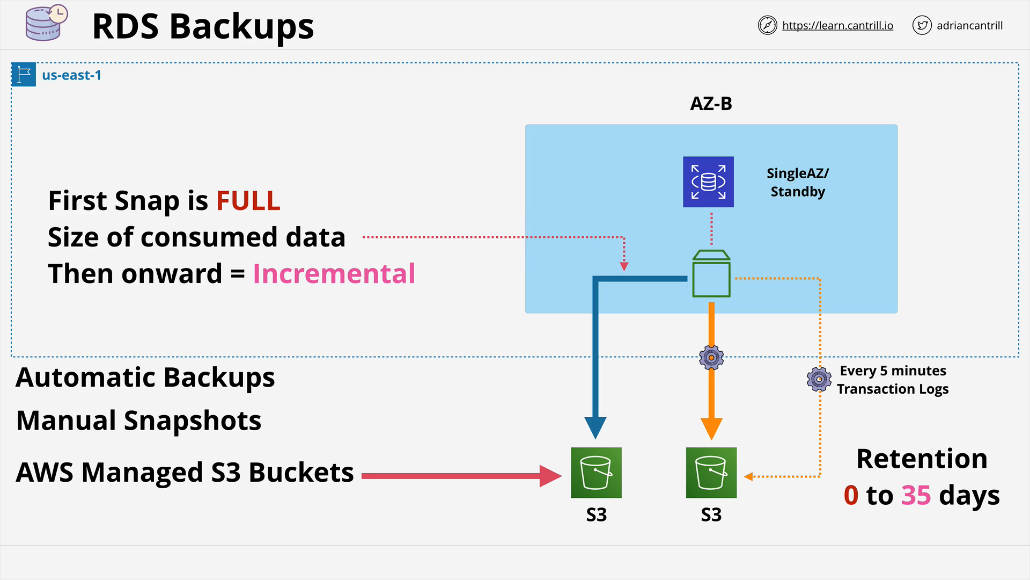
* Time between the disaster recovery event and full recovery.
* Influenced by process, staff, tech and documentation.

**RDS Backups**

**First snap is full copy of the data** used on the RDS volume. From then on, the **snapshots are incremental** and only **store the change in data**.

When any snapshot occurs, there's a **brief interruption to the flow of data between the compute resource and the storage**.

If you are using **single AZ, this can impact your application**. If you are using **Multi-AZ, the snapshot occurs on the standby replica**.

Manual snapshots don't expire, you have to clean them yourself. Automatic Snapshots can be configured to make things easier.

In **addition to automated backup, every 5 minutes database transaction logs are saved to S3**. Transaction logs **store the actual data which changes inside a database so the actual operations that are executed**.

This allows a database to be **restored to a point in time often with 5 minute granularity**.

**Automatic cleanups** can be anywhere from **0 to 35 days**. This means you can restore to any point in that time frame. This will use both the **snapshots and the translation logs**.

When you delete the database, they can be retained but they will expire based on their retention period.

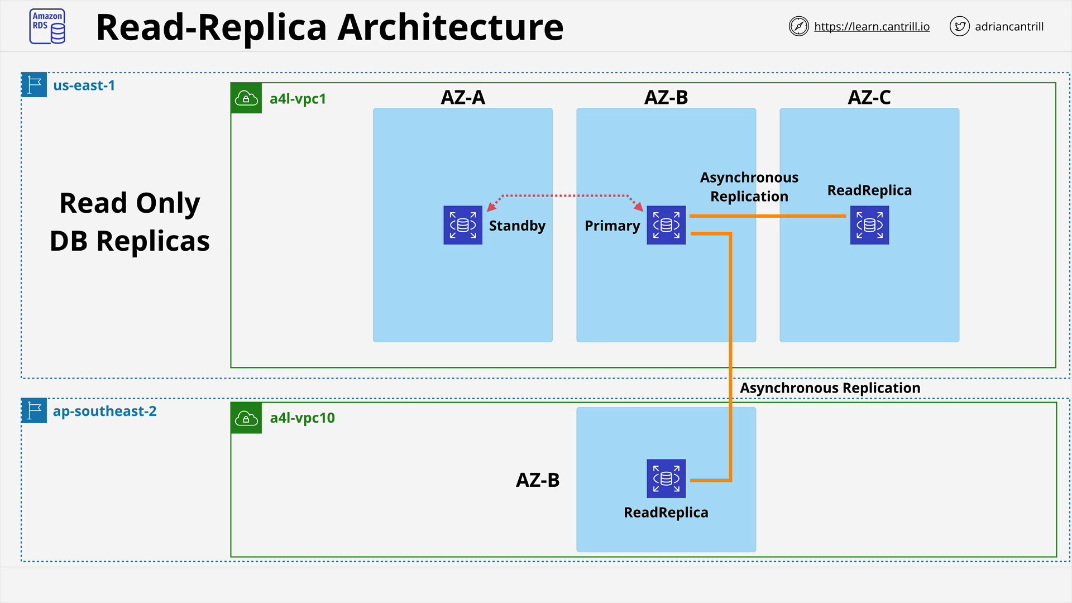
The only way to **maintain backups is to create a final snapshot which will not expire automatically**.

**RDS Backup Exam PowerUp**

* When performing a restore, RDS creates a new RDS with a new endpoint address, which needs to be updated in the application.
* When **restoring a manual snapshot**, you are **setting it to a single point in time**. This influences the **RPO value.**
* **Automated backups** are different, they allow any **5 minute point in time**.
* Backups are restored and transaction logs are replayed to bring **DB to desired point in time**.
* Restores aren't fast, think about **RTO.**
* Any form of replication can in theory **replicate data corruption, only way to protect against data corruption is to use snapshots** because you can always **restore to a point before the transaction happened.**

**RDS Read-Replicas**

Kept in sync using **asynchronous replication**

It is **written fully to the primary and standby instance first**. Once its stored on disk, it is then pushed to the replica. This means **there could be a small lag**. These can be created in the **same region or a different region**. This is known as **cross region replication**. AWS handles all of the encryption, configuration, and networking without intervention.

**Why do these matter**

READ performance

* **5 direct read-replicas** per DB instance.
* Each of these provides an additional instance of read performance.
* This allows you to scale out read operations for an instance.
* Read-replicas can chain, but lag will become a problem.
* Can provide global performance improvements.
* Provides global resilience by using cross region replication.
* They don't improve RTO
* Read Replicas provide read scaling, they don’t provide write scaling.

**Read Replicas provide near 0 RPO**

* If the primary instance fails, you can promote a read-replica to take over.
* Once it is promoted, it allows for read and write.
* **Only works for failures.**
  + Read-replicas will **replicate data corruption**.
  + In this case you must default back to snapshots and backups.
* **Promotion cannot be reversed**.

\*\***SNAPSHOTS**: takes a back up of a DB Instance in a **“particular point in time”.** So snapshots represents a discrete point in time.

\*\***SIZE of Storage** of DB Instance can be between **20Gib** to **16384Gib**

\*\*To enable Multi-AZ set, the **“Multi-AZ Deployment”** option to **“Yes”**

\*\*Can apply changes immediately or in the next scheduled maintenance window.

While Choosing Options in the Exam, Remember:

* **Reduce RTO** by conducting more **frequent snapshots**, Reduce **RTO** even further **restoring to point in time(transaction logs & automatic backups)**
* If we want to have **Lower RPO**, we can't use snapshots restore because this would take time in case of large databases, we **would be needing read replicas** but that **would not work to protect against data corruption issues.**

**Amazon Aurora**

Aurora architecture is **VERY different from RDS.**

It uses a **cluster** which is:

* A single **primary instance and 0 or more replicas**.
* The replicas within Aurora can be used for reads during normal operation.
  + Provides benefits of **RDS multi-AZ and read-replicas.**
* Aurora **doesn't use local storage for the compute instances.**
  + An Aurora cluster has a **shared cluster volume.**
  + Provides faster provisioning.
  + Improved availability.
  + Better performance.

Aurora cluster functions across a number of availability zones.

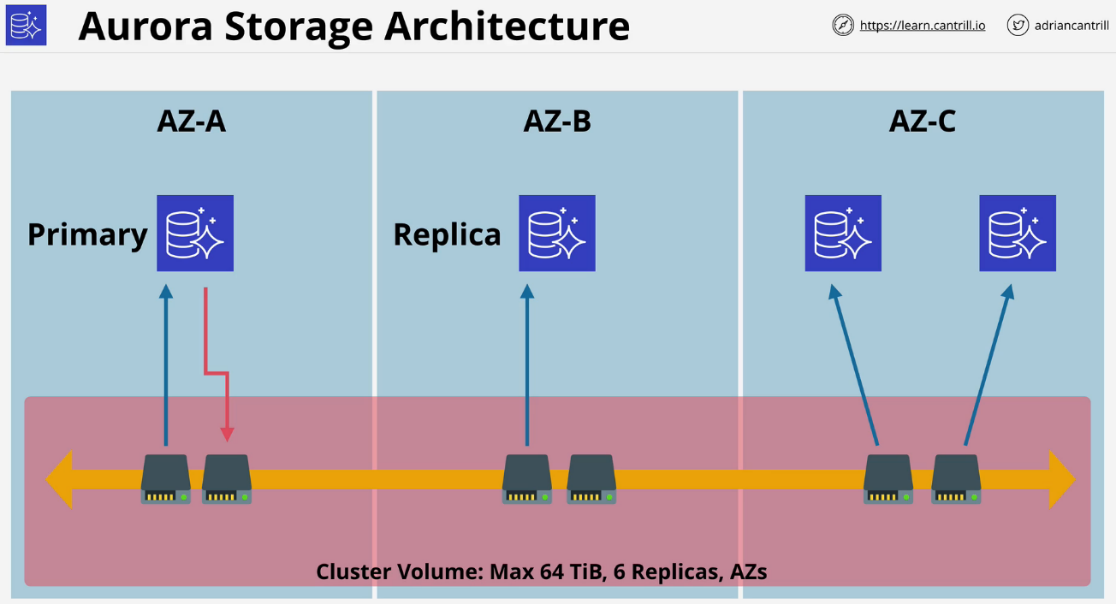
There is a primary instance and a number of replicas. The **read operations** from applications can use the **replicas**.

There is a shared storage of **max 64 TiB** across all replicas. This uses **6 copies across AZs.**

All instances have access to these storage nodes. This **replication happens at the storage level**. **No extra resources** are consumed **during replication.**

By default the **primary instance is the only one who can write**. The **replicas will have read access.**

**Aurora** automatically detect hardware failures on the shared storage. If there is a failure, it immediately repairs that area of disk and recreates that data with no corruption.

With Aurora you can have up to **15 replicas** and **any of them can be a failover target.** The failover operation will be quicker because it **doesn't have to make any storage modifications.**

* Cluster shared volume is based on **SSD storage by default**.
  + Provides so high IOPS and low latency.
  + **No way to select magnetic storage.**
* Aurora cluster **does not specify the amount of storage needed.**
  + This is **based on what is consumed.**
* **High water mark billing** or **billed for the most used.**
  + Storage which is freed up can be re-used.
  + If you reduce a lot of storage, you will need to create a brand new cluster and migrate data from the old cluster to the new cluster.
* **Storage is for the cluster** and not the instances which means **Replicas can be added and removed without requiring storage, provisioning, or removal**.

**Aurora Endpoints**

Aurora clusters like RDS use endpoints, so these are DNS addresses which are used to connect to the cluster. Unlike RDS, Aurora clusters have multiple endpoints that are available for an application.

Minimum endpoints

* **Cluster endpoint** always points at the primary instance.
  + This is used for **read and write applications**.
* **Reader endpoint**
  + Will **point at primary instance if that is all there is**.
  + Will **load balance across all available replicas for read operations**.
  + **Additional replicas which are used for reads** will be **load balanced automatically**.

**Costs**

* **No free-tier option**
* Aurora **doesn't support micro instances**
* Beyond RDS single AZ (micro) Aurora provides best value.
* Compute is **billed per second with a 10 minute minimum.**
* **Storage is billed using the high watermark for the lifetime using GB-Month**.
  + **Additional IO cost per request made to the cluster shared storage.**
* **100% DB size in backups are included for free.**
  + **100 GB cluster** will have **100 GB of storage for backups**.

**Aurora Restore, Clone and Backtrack**

**Backups in Aurora** work in the same way as RDS. **Restores create a brand new cluster**.

**Backtrack must be enabled on a per cluster basis.** This allows you to roll back your data base to a previous point in time. This helps for data corruption.

You can **adjust the window backtrack will work for**.

**Fast clones** make a new database much faster than copying all the data. It references the **original storage and only write the differences between the two**. It uses a tiny amount of storage and only stores data that's changed in the clone or changed in the original after you make the clone.

To restore a snapshot of any other RDS Product to Aurora, you need to **“Migrate” the snapshot.**

**Aurora Serverless**

Provides a version of Aurora database product without managing the resources. You still create a cluster, but it uses ACUs or **Aurora Capacity Units.**

For a cluster, you can set a **min and max ACU based on the load** and **can even go down to 0 to be paused**. In this case you would **only be billed for storage consumed.**

Billing is based on **resources used on a per-second basis.**

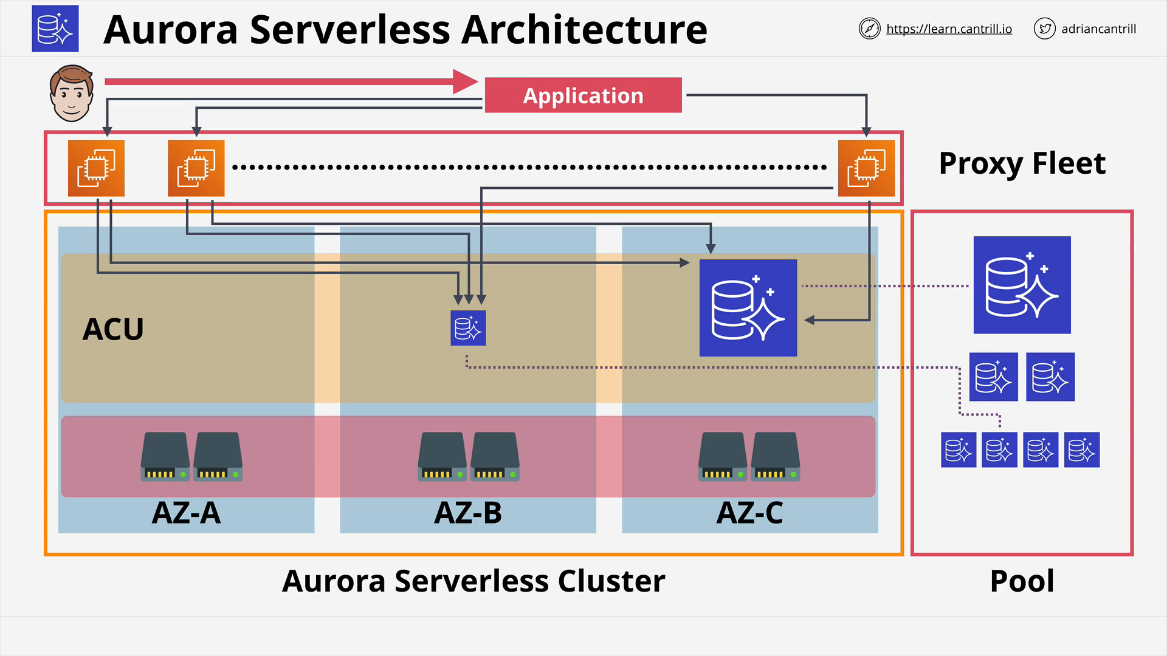
Same resilience as **Aurora (6 copies across AZs).**

**ACUs are stateless and shared across many AWS customers** and have no local storage.

They can be **allocated to your Aurora Serverless cluster** rapidly when required.

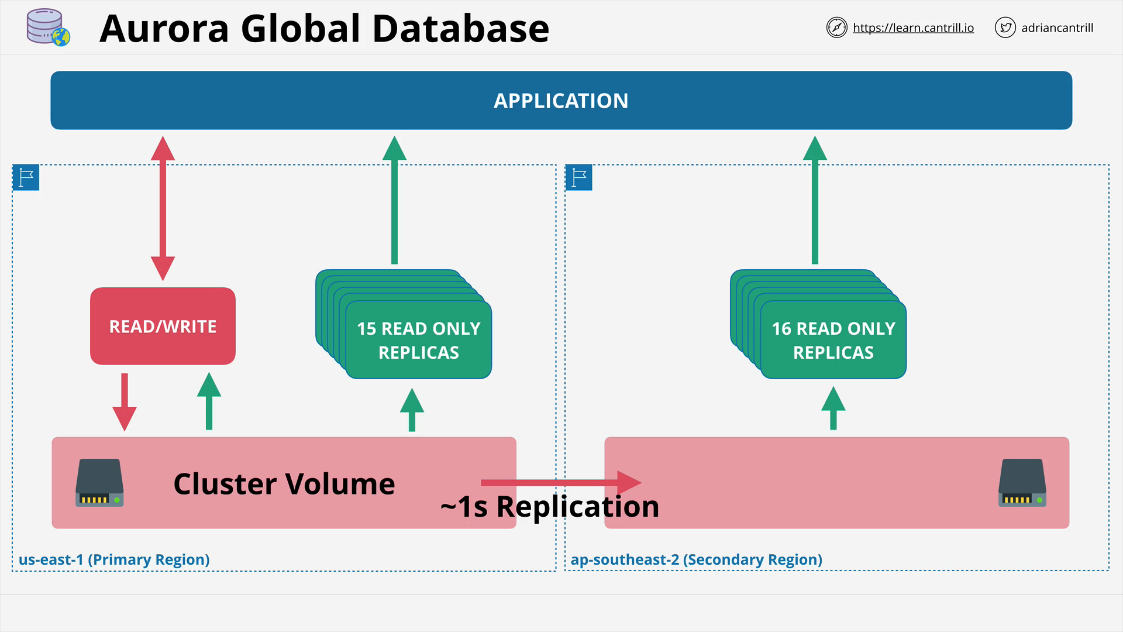
Once ACUs are allocated to a cluster, they have access to cluster storage in the same way as an **Aurora Provisioned cluster.**

There is a **shared proxy fleet**. When a customer interacts with the data they are actually **communicating with the proxy fleet.** The **proxy fleet brokers an application** with the ACU and ensures you can scale in and out without worrying about usage. This is **managed by AWS on your behalf.**

**Aurora Serverless - Use Cases**

* Infrequently used applications.
  + Low volume blog site.
  + You only pay for resources as you consume them on a per second basis.
* New applications with unpredictable workloads.
* Great for variable workloads such as sales cycles. It can scale in and out based on demand
* Good for development and test databases, can scale back when not needed.
* Great for **multi-tenant applications.**
  + **Billing a user a set dollar amount** per month per license.
  + **If your incoming load is directly tied to more revenue this makes sense.**

**Aurora Global Database**

Introduces the idea of secondary regions with up to **16 read only replicas**. Replication from primary region to **secondary regions happens at the storage layer and typically occurs within one second.**

* Great for **cross region disaster recovery and business continuity.**
* Global read scaling
  + Low latency performance improvements for international customers.
* The application can perform read operations against the read replicas.
* There is **~1s or less replication between regions**.
* It is **one way replication**.
* No additional CPU usage is needed, it **happens on the storage layer.**
* Secondary regions can have **16 replicas.**
  + All can be promoted to **Read or Write in a DR situation**.
* Maximum of **5** secondary regions.

**Aurora Multi-Master Writes**

Allows an aurora cluster to have multiple instances capable of reads and writes.

**Single-master Mode**

* one R/W and zero or more read only replicas
* **Cluster endpoint** is normally used to write
* **Read endpoint** is used for load balancing

**Aurora Multi-master**

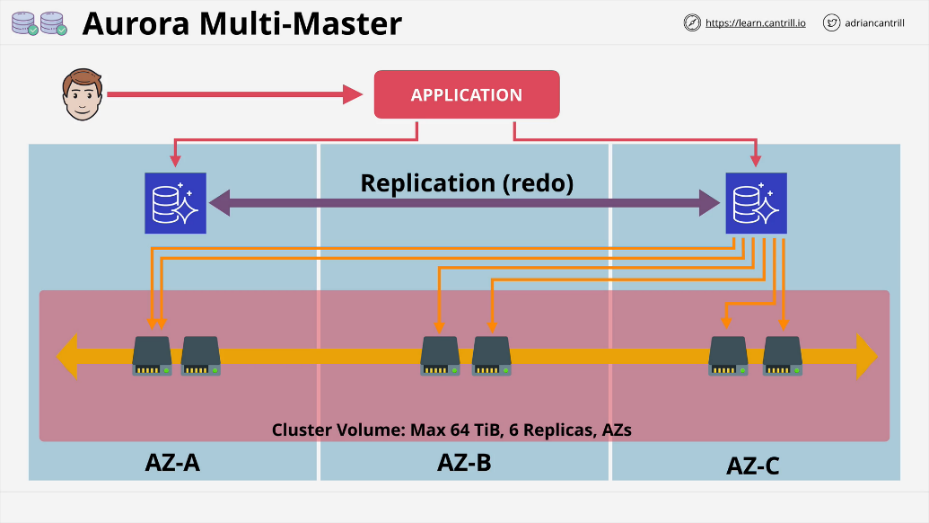
Aurora Multi-master has **no endpoint or load balancing**. An application **can connect with one or both of the instances inside a multi-master cluster.**

When one of the R/W nodes receives a write request from the application, it immediately proposes that data be **committed to all of the storage notes in that cluster.**

At this point, **each node that makes up a cluster either confirms or rejects the proposed change**. It will reject if this conflicts with something already in flight.

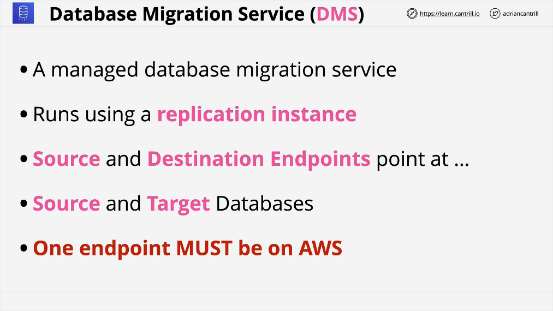
The **writing instance is looking for a bunch of nodes to agree**. If the **group rejects it, it cancels the write in error. If it commits, it will replicate on all storage nodes** in the cluster.

This also ensures **storage is updated on in-memory cache's of other nodes**.

If a writer goes down in a multi-master cluster, the application will shift all future load over to a new writer with little if any disruption. Can be used to implement fault tolerance.

**Database Migration Service (DMS)**

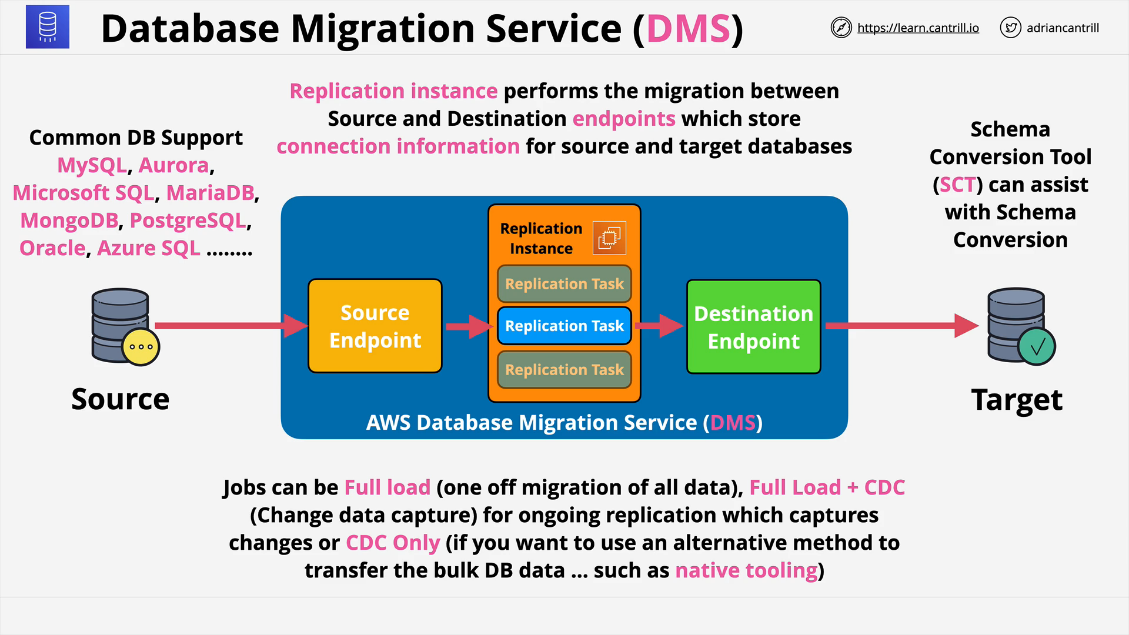
A **managed database migration service.** Starts with a **replication instance which runs on top of an EC2 instance.** This replication instance runs one or more replication tasks. This is where the configuration is defined for the migration of databases. This **runs using a replication instance.**

Need to define the source and destination endpoints. These point at the physical source and target databases. One of these end points must be on AWS.

**Full load migration** is a one off process which transfers everything at once. This requires the database to be down during this process. This might take several days.

Instead **Full Load + CDC(Change Data Capture)** allows for a full load transfer to occur and it monitors any changes that happens during this time. Any of the **captured changes can be applied to the target.**

CDC only migration is good if you have a vendor solution that works quickly and only changes need to be captured.

**Schema Conversion Tool** or SCT can perform conversions between database types.