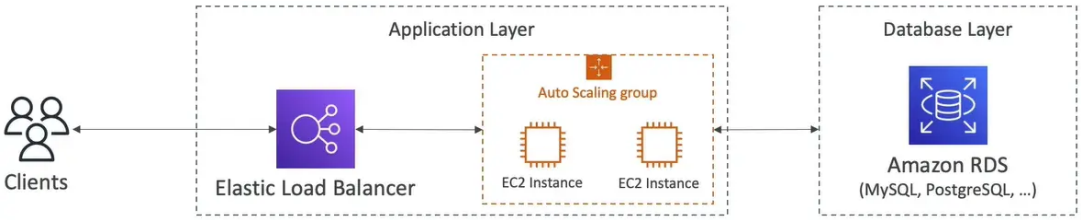
AWS Serverless: DynamoDB

Traditional Architecture



- traditional applications leverage RDBMS databases
- these tradtional have the SQL query language`
- Strong requirement about how the data should be modeled
- ability to do query joins, aggregations [聚合] ,complex computations
- vertical scaling (getting a more powerful CPU / RAM / IO)
- horizontal scaling (increasing reading capability by adding EC2 / RDS read Replicas)

NoSQL databases

- NoSQL database are non-relational databases and are distributed
- NoSQL database include MongoDB,DynamoDB
- NoSQL database do not support query joins (or just limited support)
- all the data that is needed for a query is present in one row
- NoSQL database don't perform aggregation such as "SUM","AVG"..
- NoSQL database scale horizontally

- there's no "right or wrong" for NoSQL vs SQL, they just require to model the data differently and think about user queries differently

Amazon DynamoDB

- fully managed highly available with replication across multiple AZs
- NoSQL database – not a relational database
- Scales to massive workloads ,distributed database
- millions of requests per seconds ,trillions of row, 100s of TB of storage
- fast and consistent [持续] in performance (low latency on retrieval [恢复])
- integrated with IAM for security, authorization [授权] and administration
- enables event driven programming with DynamoDB Streams
- low cost and auto-scaling capabilities
- standard & infrequent [不常见] Access (IA) Table Class

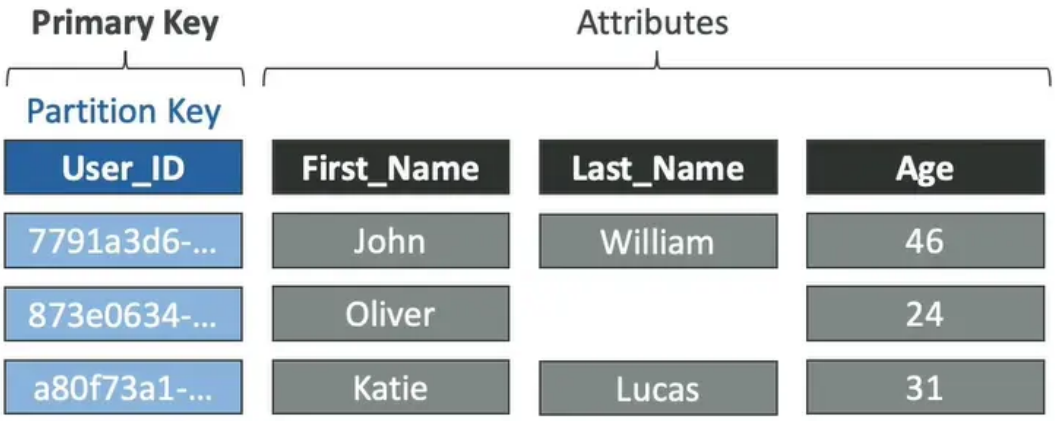
Basics

- DynamoDB is made of Tables
- each table has a primary key (must be decided at creation time)
- each table can have an infinite number of items (=rows)
- each item has attributes (can be added over time – can be null)

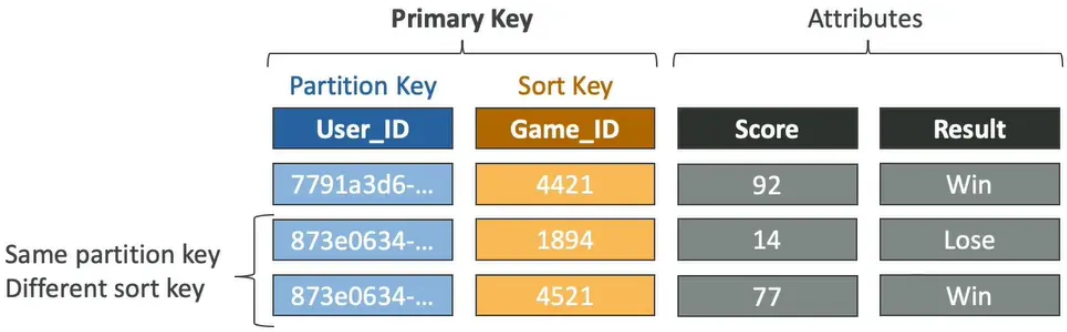
- maximum size of an item is 400KB
- data type supported are
  - scalar types – String, Number, Binary, Boolean, Null
  - document types – List, Map
  - set types – String Set,Number Set, Binary Set

Primary Keys

- option 1 : Partition Key (HASH)
  - partition key must be unique for each item
  - partition key must be "diverse" [各种各样] so that the data is distributed
  - example:"User_ID" for users table



- option 2 :partition key + sort key (HASH + RANGE)
  - the combination must be unique for each item
  - data is grouped by partition key
  - example: users–games table, "User_ID" for Partition Key and "Game_ID" for Sort Key



Read / Write Capacity Modes

- control how you mange your table's capacity (read / write throughput)

- provisioned mode (default)
  - you specify the number of reads / writes per second
  - you need to plan capacity beforehand
  - pay for provisioned read & wirte capacity units
- on–demand mode
  - read / writes automatically scale up / down with your workloads
  - no capacity planning needed
  - pay for what you use, more expensive

- you can switch between different modes once every 24 hours

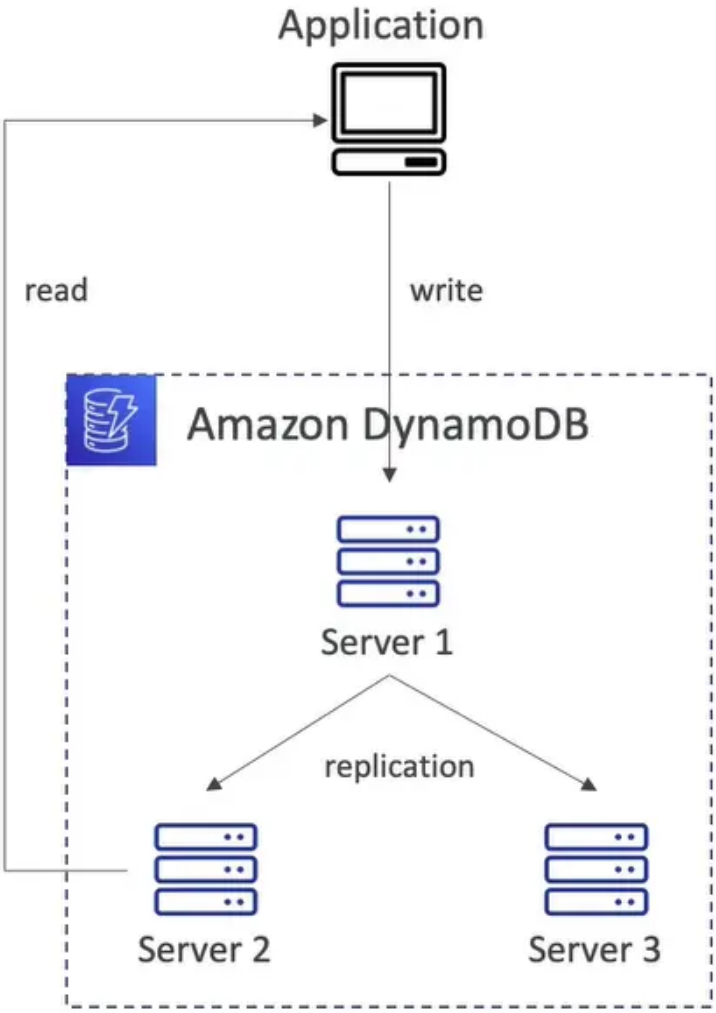
R/W Capacity Modes – Provisioned

- table must have provisioned read and write capacity units
- **Read Capacity Units(RCU)** – throughput for reads
- **Write Capacity Units(WCU)** – throughput for writes
- option to setup auto-scaling of throughput to meet demand
- throughput can be exceeded [超出] temporarily using **"Burst Capacity"**
- if Burst Capacity has been consumed, you'll get a **"ProvisionedThroughputExceededException"**
- it's then advised to do an **exponential backoff** retry

DynamoDB – Write Capacity Units (WCU)

- one write capacity unit (WCU) represents **one write** per second for an item up to **1KB** in size
- if the items are larger than 1 KB, more WCUs are consumed
- Example 1 : we write 10 items per second, with item size 2KB
  - we need *10*(2KB / 1KB) = 20 WCUs*
- Example 2: we write 6 items per second, with item size 4.5KB
  - we need *6*(5KB / 1KB) = 30WCUs* (4.5 gets rounded to the upper KB)
- Example 3: we write 120 items per minute, with item size 2KB
  - we need *(120 / 60) *(2KB /1KB) = 4WCUs*

Strongly Consistent Read vs Eventually Consistent Read



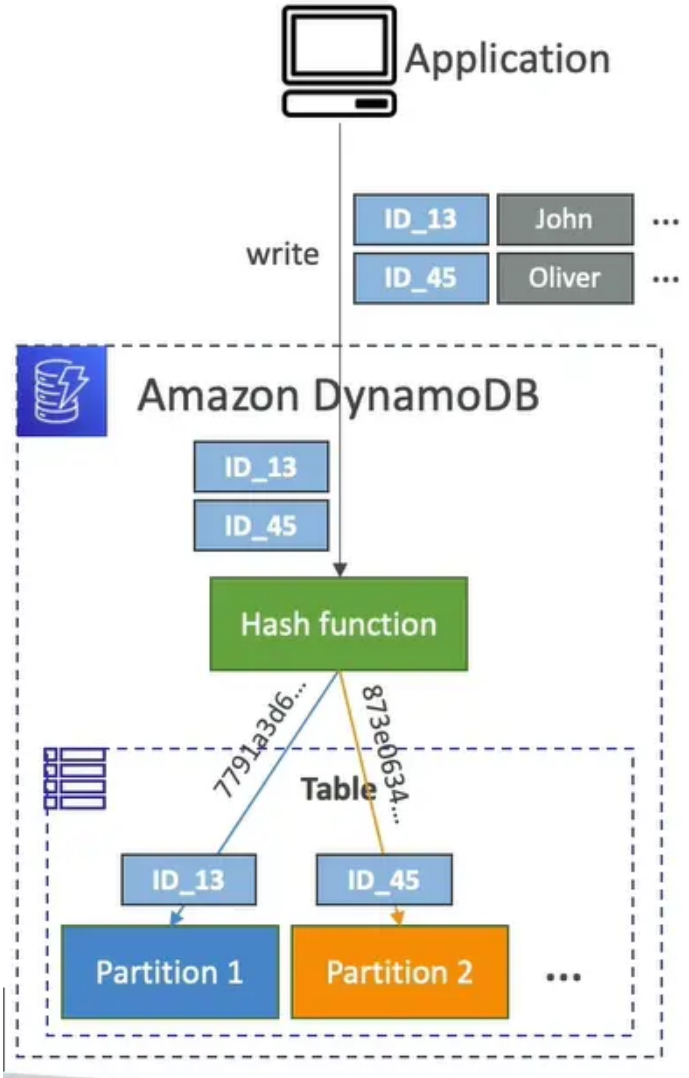
- Eventually [最终] Consistent Read (default)
  - if we read just after a write, it's possbile we'll get some stale [陈旧] data because of replication [复制]
- Strongly Consistent Read
  - if we read just after a write, we will get the correct data.
  - set "ConsistentRead" parameter to True in API calls (GetItem,BatchGetItem,Query,Scan)

Read Capacity Unit (RCU)

- one *Read Capacity Unit (RCU)* represents one Strongly Consistent Read per second,or two Eventually Consistent Reads per second, for an item up to 4KB in size
- if the item are larger than 4KB, more RCUs are consumed

- example 1 :10 Strongly Consistent Reads per second, with item size 4KB
  - we need 10*(4KB/4KB) = 10 RCUs
- Example 2: 16 Eventually Consistent Reads per second, with item size 12KB
  - we need (16 / 2)* (12KB /4KB) = 24RCUs
- Example 3: 10 Strongly Consistent Reads per second, with item size 6KB
  - we need 10 * (8KB / 4KB) = 20RCUs (we must round up 6KB to 8KB)

Partitions [分区] Internal [内部]



- data is stored in partitions

- partition keys go through a hashing algorithm to know which partition they go to
- to compute the number of partitions:
  - $\#\ of\ partitions_{by\ capacity} = \left(\frac{RCUs_{Total}}{3000}\right) + \left(\frac{WCUs_{Total}}{1000}\right)$
  - $\#\ of\ partitions_{by\ size} = \frac{Total\ Size}{10\ GB}$
  - $\#\ of\ partitions = \text{ceil}(\max(\#\ of\ partitions_{by\ capacity}, \#\ of\ partitions_{by\ size}))$
- WCUs and RCUs are spread [传播] evenly [均匀地] across partitions

Throttling

- if we exceed [超过] provisioned RCUs or WCUs, we get "ProvisionedThroughputExceededException"
- Reasons:
  - Hot Keys – one partition key is being read too many times (eg,popular item)
  - Hot partitions
  - very large items,remember RCU and WCU depends on size of items
- Solutions
  - Exponential backoff when exception is encountered [遭遇] (already in SDK)
  - Distribute partition keys as much possible
  - if RCU issue, we can use DynamoDB Accelerator [加速器](DAX)

R/W Capacity Modes – On-Demand

- Read/Write automatically scale up / down with your workloads
- no capacity planning needed (WCU / RCU)
- unlimited WCU & RCU,no throttle, more expensive
- you're charged for reads/writes that you use in iterms of RRU and WRC
- Read Request Units(RRU) – throughout for reads (same as RCU)
- Write Request Unites(WRU) – throughput for writes (same as WCU)
- 2.5x more expensive than provisioned capacity (use with care)
- use case: unknown workloads, unpredictable application traffic...

Writing Data

- PutItem
  - create a new item or fully replace an old item(same primary key)
  - consumes WCUs
- UpdateItem
  - edits an existing item's attributes or adds a new item if it doesn't exist
  - can be used to implement Atomic Counters – a numeric attribute that's unconditionally incremented [无条件递增]
- conditional writes
  - accept a write/update/delete only if condition are met, otherwise returns an error
  - helps with concurrent access [并发访问] to items
  - no performance impact

Reading Data

- GetItem
  - read based on primary key
  - primary key can be HASH or HASH+RANGE
  - eventually consistent read (default)
  - option to use Strongly Consistent Reads (more RCU – might take longer)
  - projectExpression can be specified to retrieve [取回] only certain attributes

Reading Data (Query)

- Query returns items based on:
  - KeyConditionExpression
    - partition key value (must be = operator) – required
    - sort key value (=,<,<=,>,>=,Between,Begins with) – optional
  - FilterExpression
    - additional filtering after the Query operation (before data returned to you)
    - use only with non-key attributes (does not allow HASH or RANGE attributes)
- Returns
  - the number of items specified in Limit
  - or up to 1MB of data
- ability to do pagination on the results
- can query table,a local Secondary Index,or a Global Secondary Index

Reading Data (Scan)

- Scan the entire table and then filter out data (inefficient [效率低下])
- returns up to 1MB of data – use pagination to keep on reading
- consumes a lot of RCU
- limit impact using Limit or reduce the size of the result and pause
- for faster performance, use Parallel Scan
  - multiple workers scan multiple data segments [段] at the same time
  - increase the throughput and RCU consumed
  - limit the impact of parallel scans just like you would for scans
- can use ProjectionExpression & FilterExpression (no changes to RCU)

Deleting Data

- DeleteItem
  - delete an individual item
  - ability to perform a conditional delete
- DeleteTable
  - delete a whole table and all its items
  - much quicker deletion than calling DeleteItem on all items

Batch Operations

- allows you to save in latency by reducing the number of API calls
- operations are done in parallel for better efficiency [效率]
- part of a batch can fail; in which case we need to try again for the failed items

- BatchWriteItem

    - up to 25 PutItem and /or DeleteItem in one call
    - up to 16MB of data written, up to 400KB of data per item
    - can't update items (use UpdateItem)
    - UnprocessedItems for failed write operations (exponential backoff or add WCU)
- BatchGetItem
    - return items from one or more tables
    - up to 100 items ,up to 16 MB of data
    - items are retrieved in parallel to minimize latency
    - UnprocessedKeys for failed read operations (exponential backoff or add RCU)

PartiQL

```
SELECT OrderID, Total
FROM Orders
WHERE OrderID IN [1, 2, 3]
ORDER BY OrderID DESC
```

- SQL – compatible [兼容] query language for DynamoDB
- allows you to select, insert, update, and delete data in DynamoDB using SQL
- run queries across multiple DynamoDB tables
- Run PartiQL queries from:
    - AWS management console
    - NoSQL Workbench for DynamoDB
    - DynamoDB APIs
    - AWS CLI
    - AWS SDK

Conditional Writes

- for PutItem,UpdateItem,DeleteItem,and BatchWriteItem
- you can specify a condition expression to determin which items should be modified
    - attribute_exists
    - attribute_not_exists
    - attribute_type
    - contains (for string)
    - begins_with (for string)
    - ProductCategory [产品分类] IN （:cat1,:cat2） and Price between :low and :high
    - size (string length)

- Note: filter expression filters the results of read queries, while condition expressions are for write operations

example on update item

```
aws dynamodb update-item \
    --table-name ProductCatalog \
    --key '{ "Id": { "N": "456" } }' \
    --update-expression "SET Price = Price - :discount" \
    --condition-expression "Price > :limit" \
    --expression-attribute-values file://values.json
```

```
{
    ":discount": {
        "N": "150"
    },
    ":limit": {
        "N": "500"
    }
}
```

values.json

```
{
    "Id": {
        "N": "456"
    },
    "Price": {
        "N": "650"
    },
    "ProductCategory": {
        "S": "Sporting Goods"
    }
}
```

↓

```
{
    "Id": {
        "N": "456"
    },
    "Price": {
        "N": "500"
    },
    "ProductCategory": {
        "S": "Sporting Goods"
    }
}
```

example on delete item

- attribute_not_exists
  - only succeeds if the attribute doesn't exist yet (no value)

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{ "Id": { "N": "456" } }' \
    --condition-expression "attribute_not_exists(Price)"
```

- attribute_exists
  - opposite of attribute_not_exists

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{ "Id": { "N": "456" } }' \
    --condition-expression "attribute_exists(ProductReviews.OneStar)"
```

do not overwrite elements

- attribute_not_exists(partition_key)
  - make sure the item isn't overwritten
- attribute_not_exists(partition_key) and attribute_not_exists(sort_key)
  - make sure the partition / sore key conbination is not overwritten

example complex condition

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{ "Id": { "N": "456" } }' \
    --condition-expression "(ProductCategory IN (:cat1, :cat2)) and (Price between :lo and :hi)" \
    --expression-attribute-values file://values.json
```

```
{
    ":cat1": {
        "S": "Sporting Goods"
    },
    ":cat2": {
        "S": "Gardening Supplies"
    },
    ":lo": {
        "N": "500"
    },
    ":hi": {
        "N": "600"
    }
}
```

values.json

```
{
    "Id": {
        "N": "456"
    },
    "Price": {
        "N": "650"
    },
    "ProductCategory": {
        "S": "Sporting Goods"
    }
}
```

example of string comparisons

- begin_with – check if prefix matches

- contains – check if string is contained in another string

```
aws dynamodb delete-item \
    --table-name ProductCatalog \
    --key '{ "Id": { "N": "456" } }' \
    --condition-expression "begins_with(Pictures.FrontView, :v_sub)" \
    --expression-attribute-values file://values.json
```

```
{
    ":v_sub": {
        "S": "http://"
    }
}
```

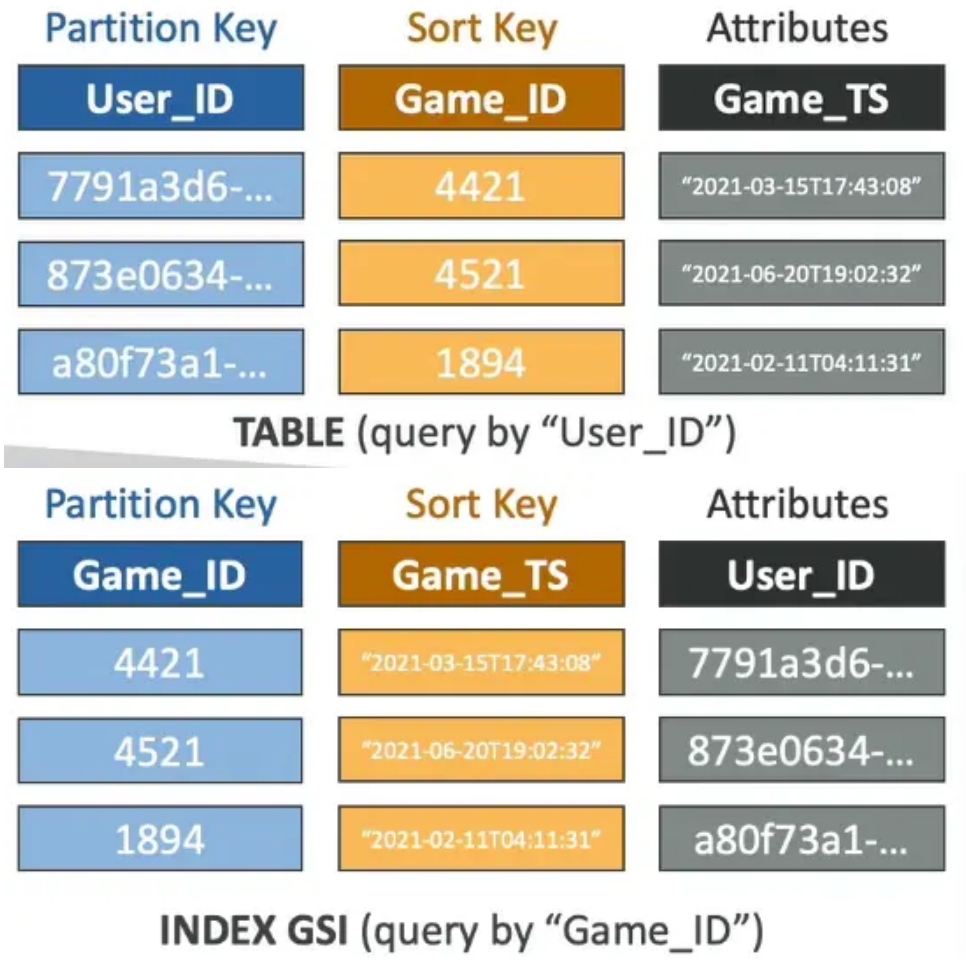
values.json

Local Secondary Index (LSI)

- Alternative Sort Key for your table (same Partition Key as that of base table)
- the sort key consists of one scalar attribute (string, number , or binary)
- up to 5 local Secondary Indexes per table
- must be defined at table creation time
- attribute projections – can contain some or all the attributes of the base table (KEYS_ONLY,INCLUDE,ALL)

Global Secondary Index (GSI)

- alternative primary key (HASH or HASH+RANGE) from the base table
- speed up queries on non-key attributes
- the index key consists of scalar attributes (string, number, or binary)
- attribute projections – some or all the attributes of the base table (KEYS_ONLY,INCLUDE,ALL)
- must provision RCUs & WCUs for the index
- can be added / modified after table creation



Indexes and Throttling

- Global Secondary Index (GSI)
  - if the writes are throttled on the GSI, then the main table will be throttled !
  - even if the WCU on the main tables are fine
  - choose your GSI partition key carefully!
  - assign [分配] your WCU capactiy carefully !

- Local Secondary Index (LSI)
  - uses the WCUs and RCUs of the main table
  - no special throttling considerations

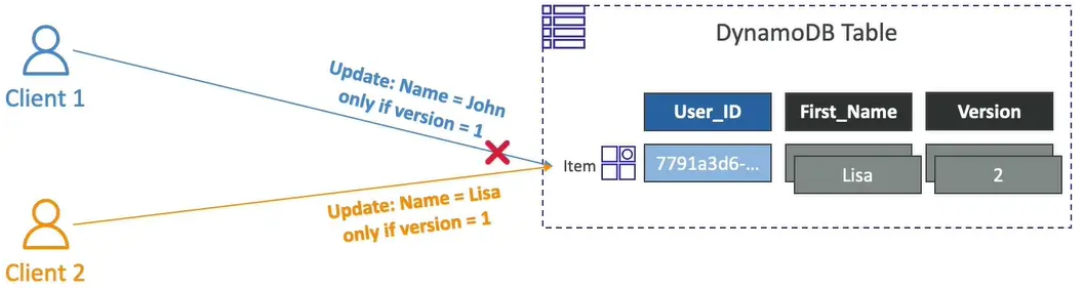
PartiQL

- use a SQL-like syntax to manipulate DynamoDB tables

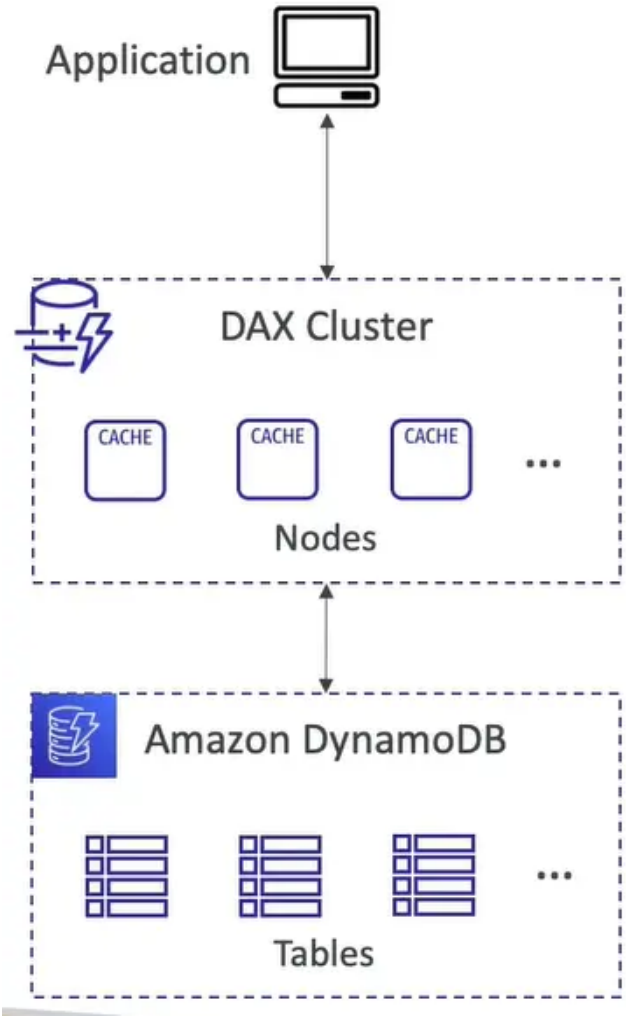
- supports some (but not all) statements
  - insert
  - update
  - select
  - delete
- it supports batch operations

Optimistic Locking [乐观锁]

- DynamoDB has a feature called "conditional writes"
- a strategy [方法] to ensure an item hasn't changed before you update / delete it
- each item has an attribute that acts as version number

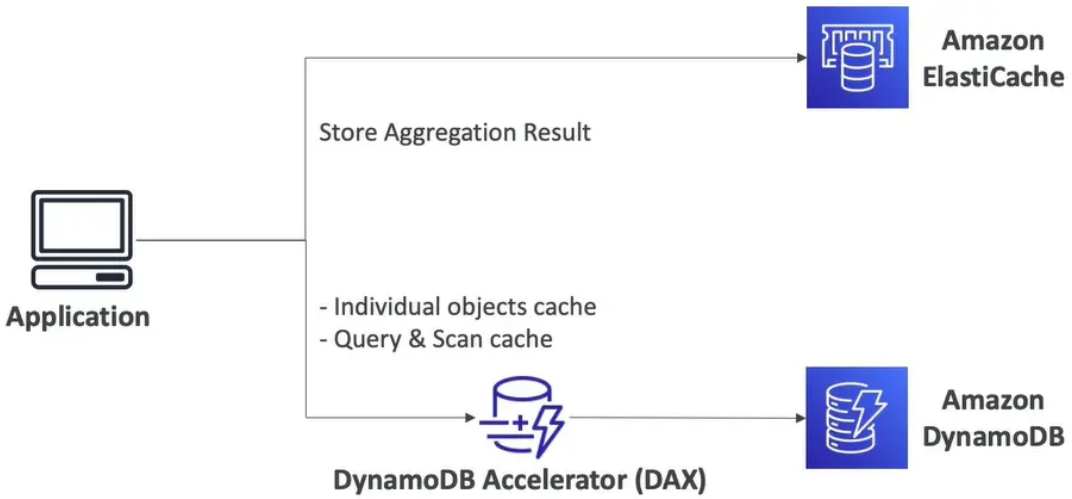


DynamoDB Accelerator [加速器] (DAX)



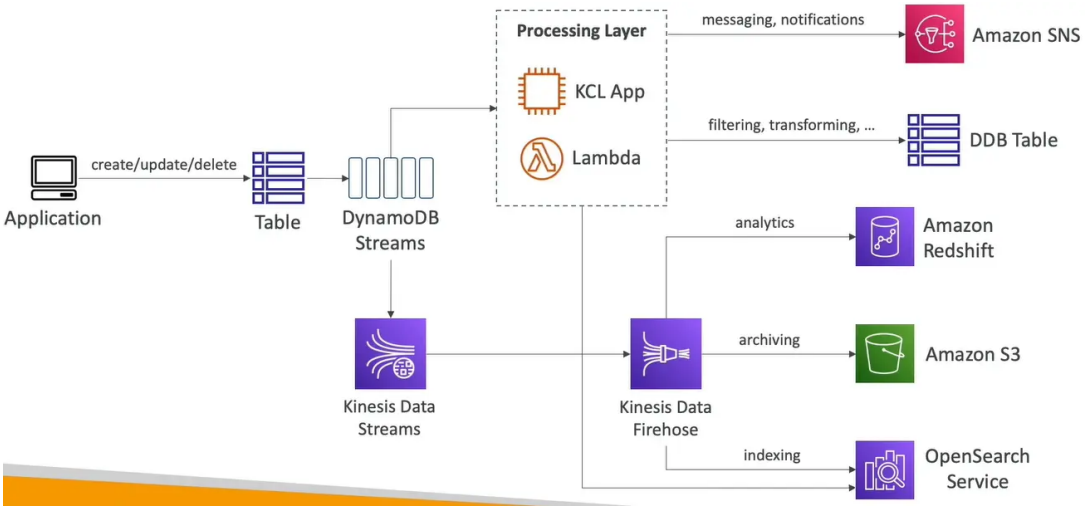
- fully-managed, highly available, seamless [无缝的] in-memory cache for DynamoDB
- microseconds latency for cached reads & queries
- doesn't require application logic modification(compatible with existing DynamoDB APIs)
- solves the "Hot Key" problem (too many reads)
- 5 minutes TTL for cache (default)
- up to 10 nodes in the cluster
- multi-AZ (3 nodes minimum recommended for production)
- secure (encryption at rest with KMS, VPC, IAM,CloudTrail,..)

DynamoDB Accelerator (DAX) vs, ElastiCache



DynamoDB Streams

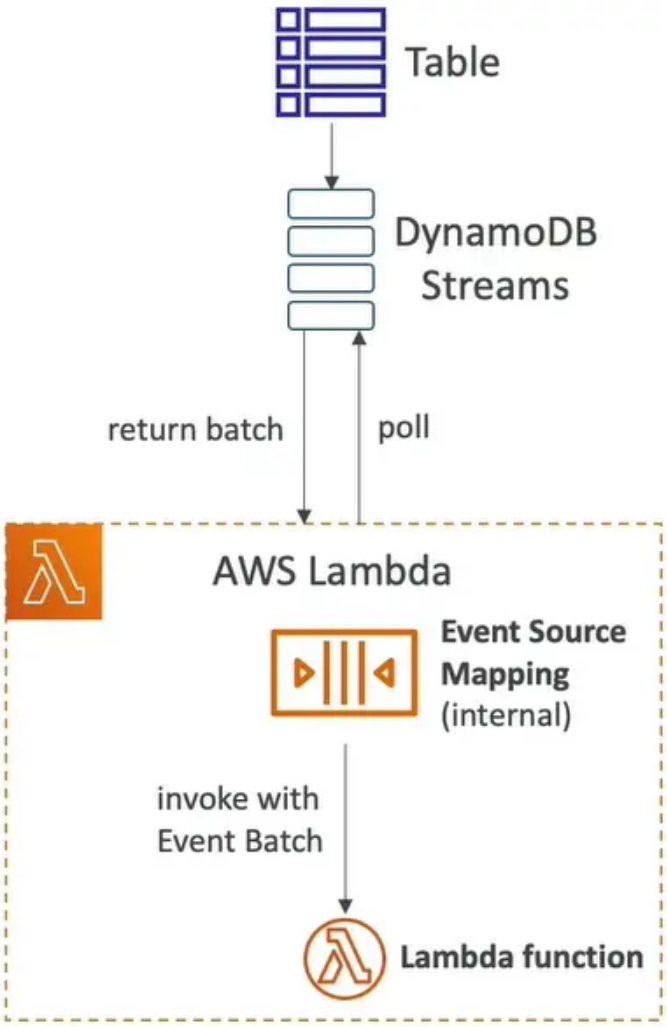
- ordered stream of item-level modifications (create/update/delete) in a table
- stream records can be:
  - sent to kinesis data streams
  - read by AWS Lambda
  - read by Kinesis Client Library applications
- Data retention for up to 24 hours
- use cases
  - react to changes in real-time (welcome email to users)
  - analytics
  - insert into derivative [衍生] tables
  - insert into opensearch service
  - implement cross-region replication



- ability to choose the information that will be written to the stream
  - KEYS_ONLY – only the key attributes of the modified item
  - NEW_IMAGE – the entire item, as it appears after it was modified
  - OLD_IMAGE – the entire item,as it appeared before it was modified
  - NEW_AND_OLD_IMAGES – both the new and the old images of the item
- DynamoDB Streams are made of shards , just like Kinesis Data Streams
- you don't provision shards, this is automated by AWS

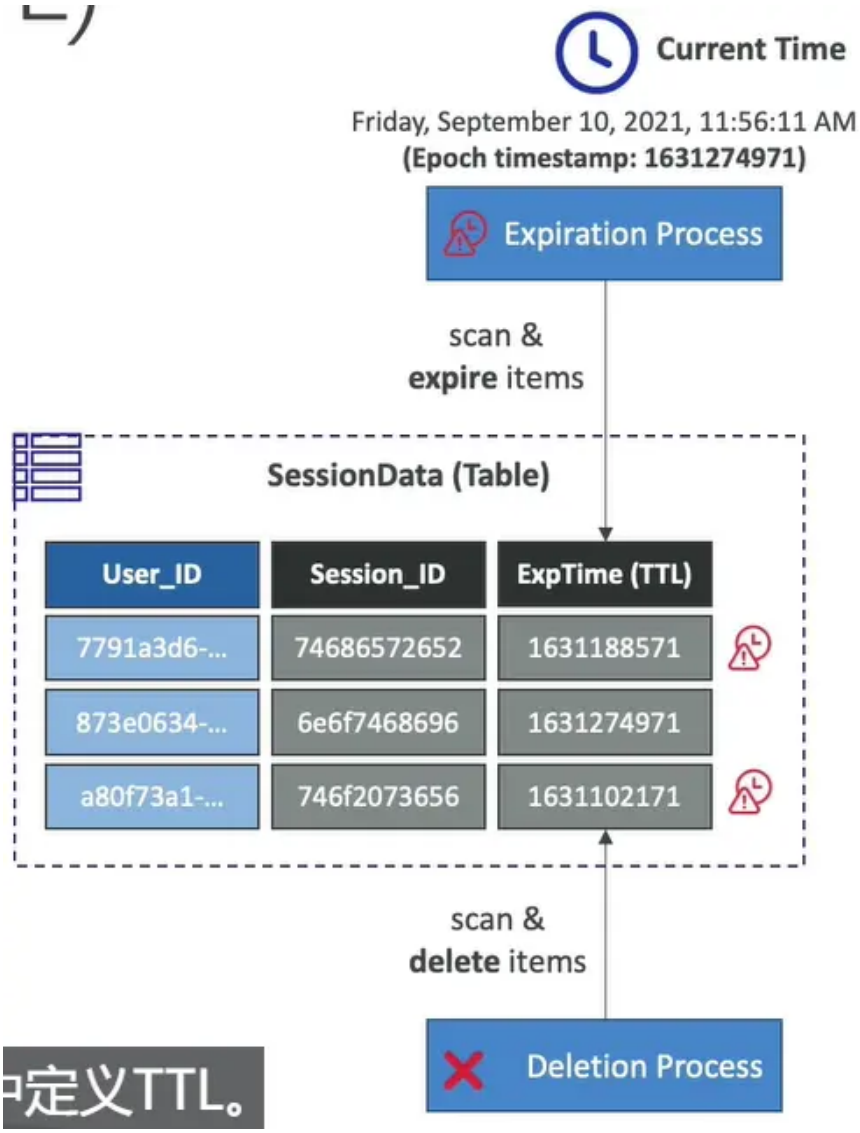
- Records are not retroactively [追溯] populated in a stream after enabling it

DynamoDB Streams & AWS Lambda



- you need to define an **Event Source Mapping** to read from a DynamoDB Sreams
- you need to ensure the Lambda function has the **appropriate permissions**
- **your lambda function is invoked synochronously**

Time To Live (TTL)



定义TTL。

- automatically delete items after an expiry timestamp
- doesn't consume any WCUs (no extra cost)
- the TTL attribute must be a "Number" data type with "Unix Epoch timestamp" value
- expired items deleted within 48 hours of expiration
- expired items, that haven't been deleted, appears in reads / queries /scans (if you don't want them, filter them out)
- expired items are deleted from both LSIs and GSIs
- a delete operation for each expired item enters the DynamoDB Streams (can help recover expired items)
- use case : reduce stored data by keeping only current items, adhere [坚守]to regulatory obligations [义务]

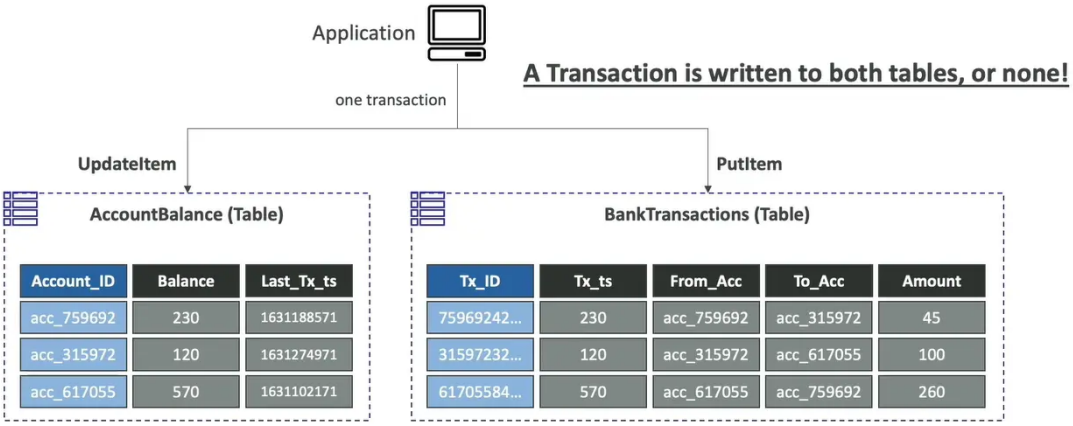
DynamoDB CLI --Good to Know

- --projection-expression : one or more attributes to retrieve
- --filter-expression : filter item before returned to you

- General AWS CLI Pagination options (eg, DynamoDB ,S3,..)
  - --page-size: specify that AWS CLI retrieves the full list of items but with a larger number of API calls instead of one API call (default : 1000items)
  - --max-items : max,number of items to show in the CLI (return NextToken)
  - --starting-token : specify the last NextToken to retrieve the next set of items

DynamoDB Transactions [事务]

- coordinated [协调], all-or-nothing operations (add/update/delete) to multiple item across one or more tables
- provides Atomicity [原子性], Consistency [一致性],Isolation [隔离性], and Durability [持久性] (ACID)
- Read Mode - eventual [最终] consistency,strong consistency,transactional
- Write Mode - standard, Transactional
- Consume 2x WCUs & RCUs
  - DynamoDB perform 2 operations for every item (prepare & commit)
- Two operations
  - TransactGetItems - one or more GetItem operations
  - TranactWriteItems - one or more PutItem, UpdateItem, and DeleteItem operations
- use cases： financial transactions, managing orders, multiplayer games...



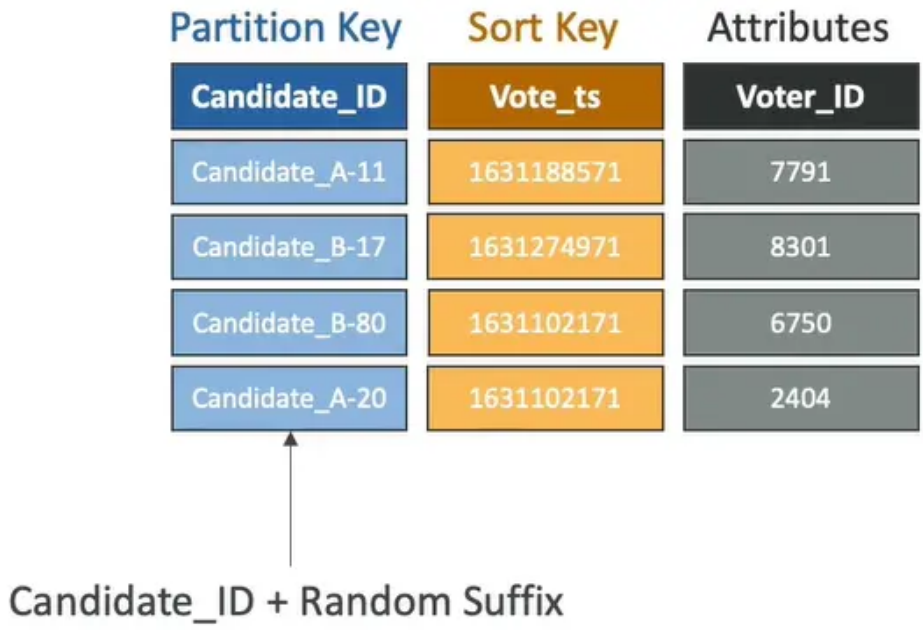
Capacity Computations

- important for the exam !
- Example 1 : 3 Transactional writes per second, with item size 5 KB
  - we need 3*(5KB / 1KB)*2 (transactional cost) = 30 WCUs
- Example 2: 5 Transaction reads per second, with item size 5KB
  - we need 5*(8KB / 4KB)*2 (transactional cost) = 20 RCUs
  - 5 gets rounded to the upper 4KB

DynamoDB as Session State Cache

- it's common to use DynamoDB to store session state
- vs ElastiCache:
  - ElastiCache is in-memory, but DynamoDB is serverless
  - both are key/value stores
- vs EFS:
  - EFS must be attached to EC2 instances as a network drive
- vs EBS & Instance Strore
  - EBS & Instance Store can only be used for local caching, not shared caching
- vs S3
  - S3 is higher latency,and not meant for small objects

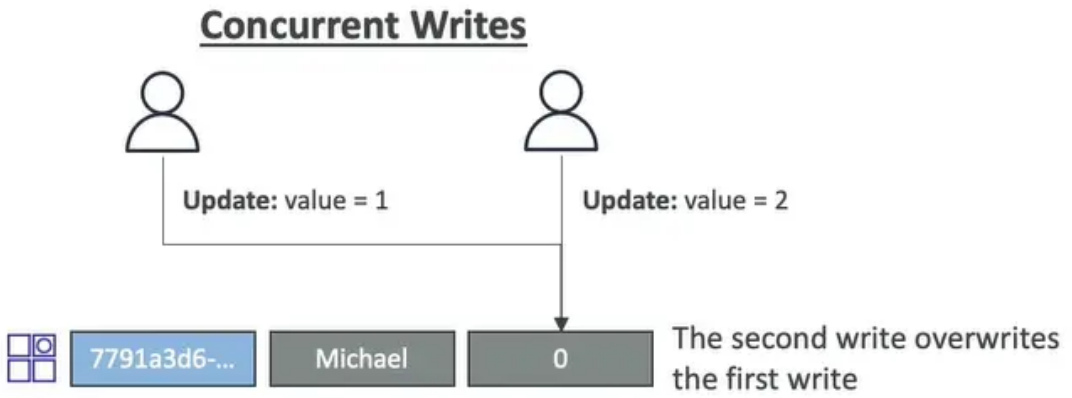
DynamoDB Write Sharding

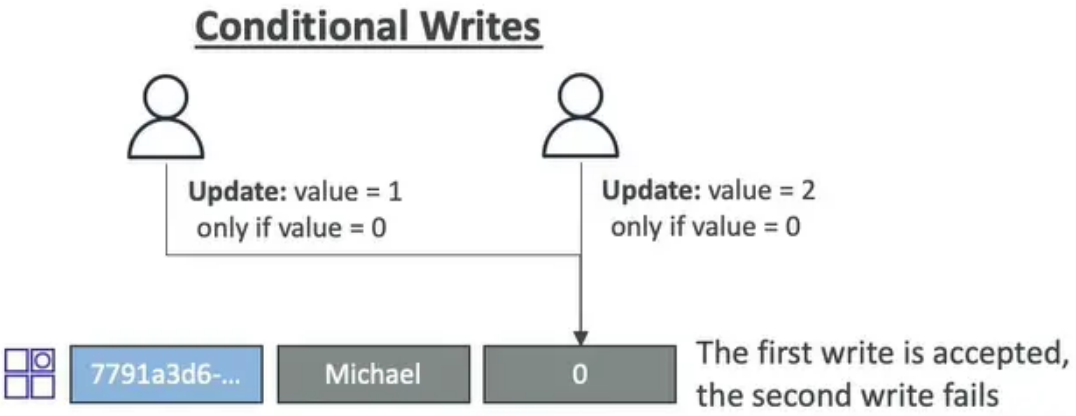


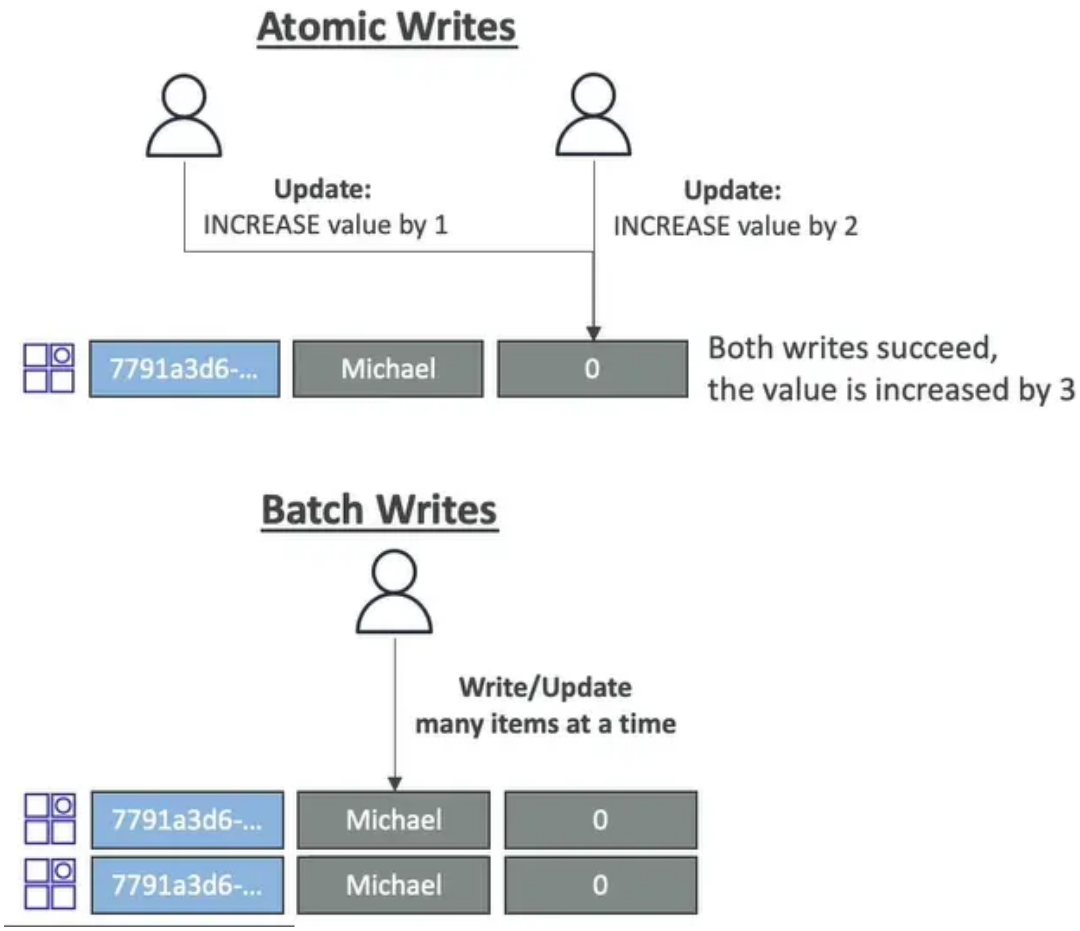
- imagine we have a voting application with 2 candidate, candidate [候选人] A and candidate B
- if partition Key is "Candidate_ID", this results into two partitions, which will generate issues (ag,hot partition)

- a strategy that allows better distribution of items evenly across partitions
- add a suffix [后缀] to partition key value
- 2 methods
  - sharding using random suffix
  - sharding using calculated [计算的] suffix

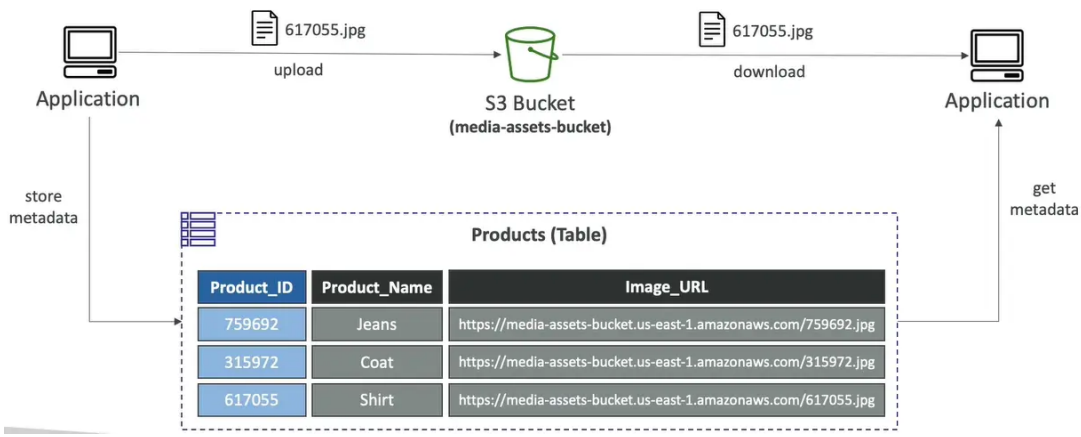
Write Types



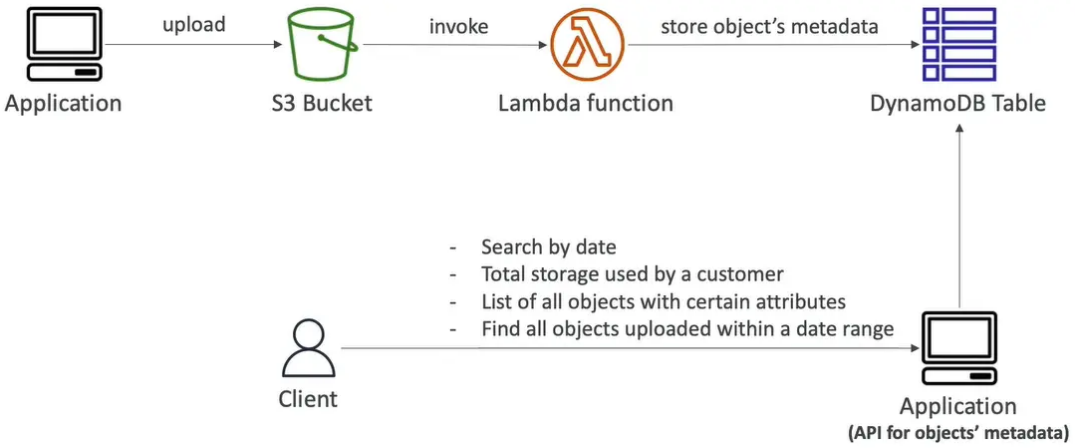




Large Object Pattern

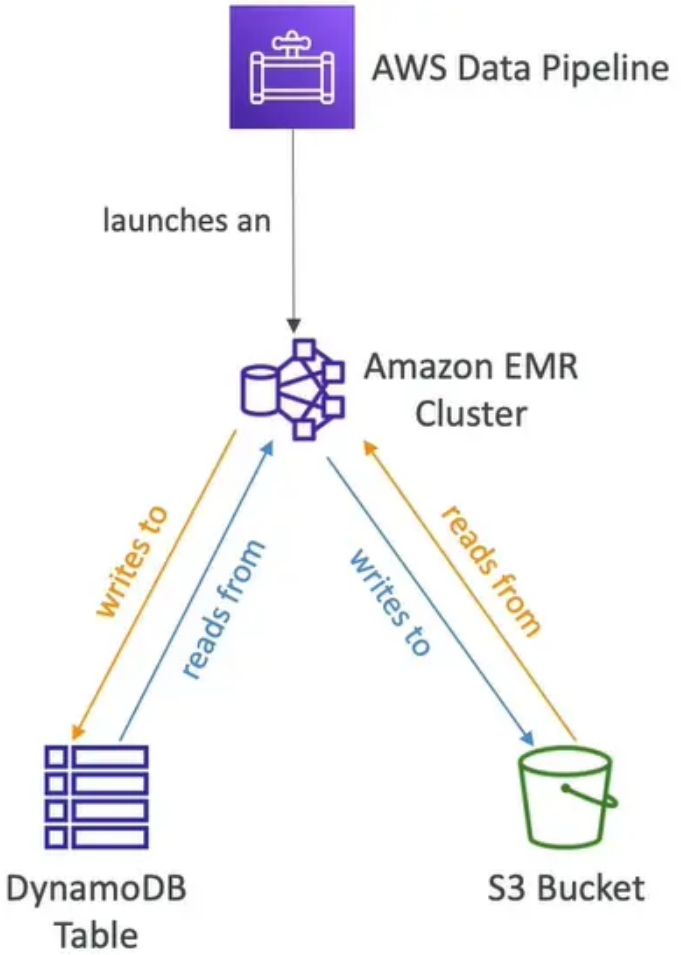


Indexing [索引] S3 Objects Metadata



DynamoDB Operation

- table Cleanup
  - Option 1 : Scan + DeleteItem
    - very slow,consumes RCU & WCU, expensive
  - Option 2 : Drop Table + Recreate table
    - fast, efficient, cheap
- Copying a DynamoDB Table
  - option 1: using AWS Data Pipeline

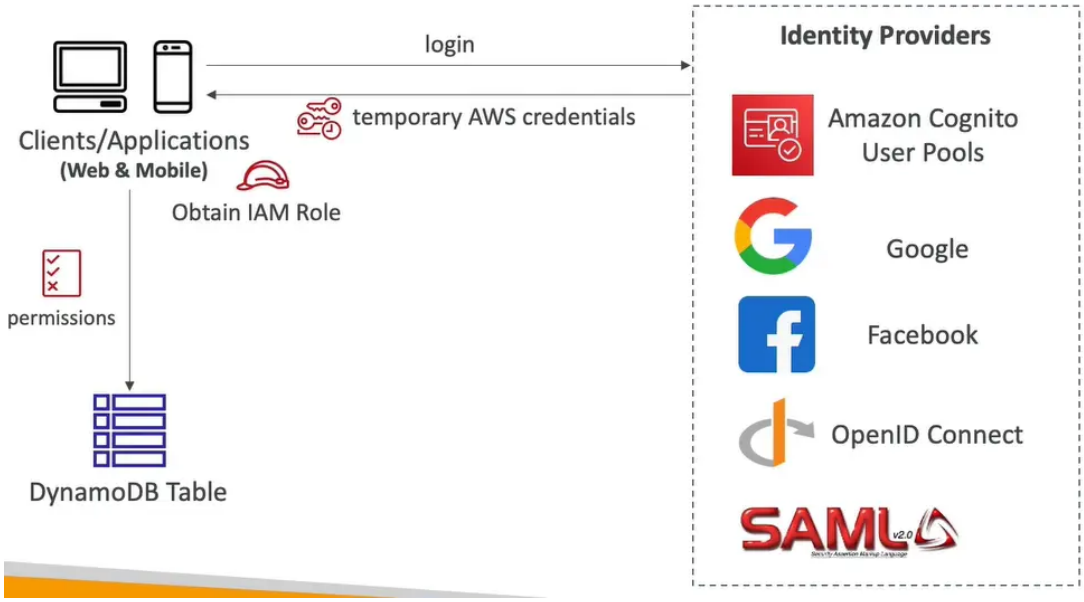


  - option 2: backup and restore into a new table
    - takes some time
  - option 3: scan + PutItem or BatchWriteItem
    - write your own code

Security & Other Features

- Security
  - VPC Endpoint available to access DynamoDB without using the Internet
  - access full controlled by IAM
  - encryption at rest using AWS KMS and in-transit using SSL/TLS
- Backup and Restore feature available
  - point-in-time recovery (PITR) like RDS
  - no performance impact
- Global Tables
  - mult-region,multi-active,fully replicated, high performance
- DynamoDB Local
  - develop and test apps locally without accessing the DynamoDB web service (without internet)
- AWS Database Migration Service (AWS DMS)j can be used to migrate to DynamoDB (from MongoDB, Oracle,MySQL,S3,...)

Users Interact [交互] with DynamoDB Directly



Fine-Grained Access Control

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetIem", "dynamodb:BatchGetItem", "dynamodb:Query",
                "dynamodb:PutItem", "dynamodb:UpdateItem", "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "arn:aws:dynamodb:us-west-2:123456789012:table/MyTable",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": ["${cognito-identity.amazonaws.com:sub}"]
                }
            }
        }
    ]
}
```

More at: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/specifying-conditions.html

- using Web Identity Federation or Cognito Identity Pools, each user gets AWS credentials
- you can assign an IAM Role to these users with a Condition to limit their APU access to DynamoDB
- LeadingKeys – limit row-level access for users on the Primary Key
- Attribute – limit specific attributes the user can see