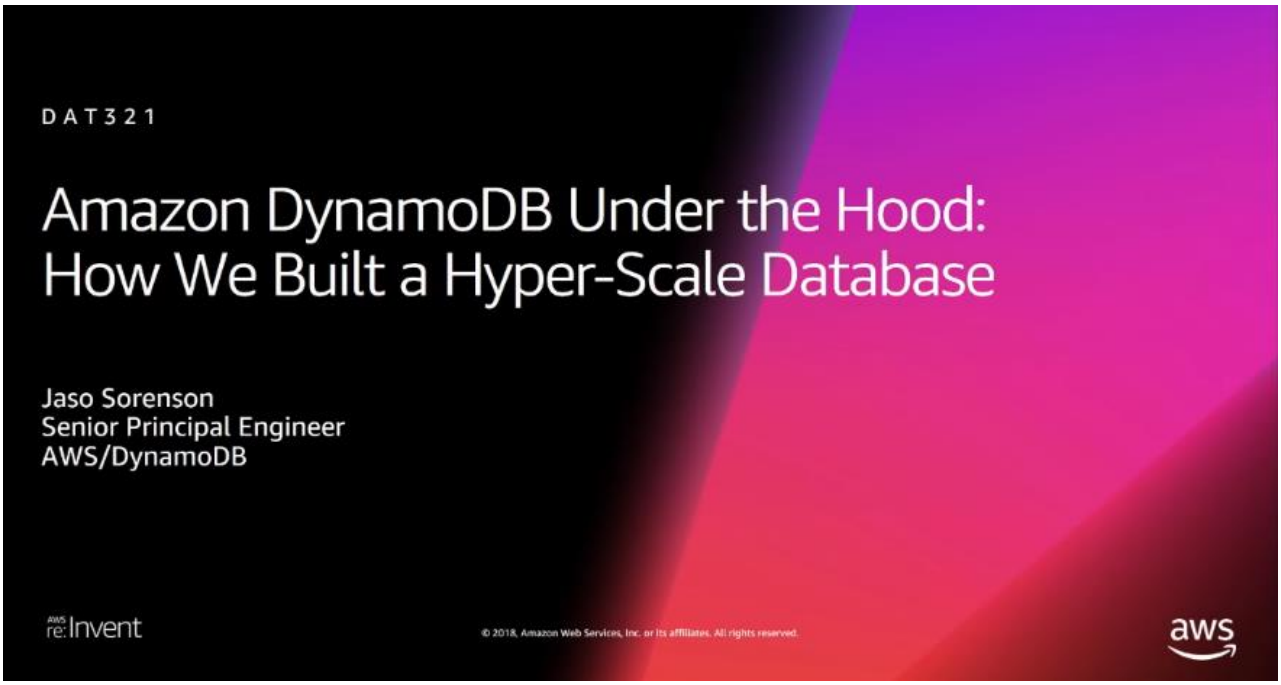
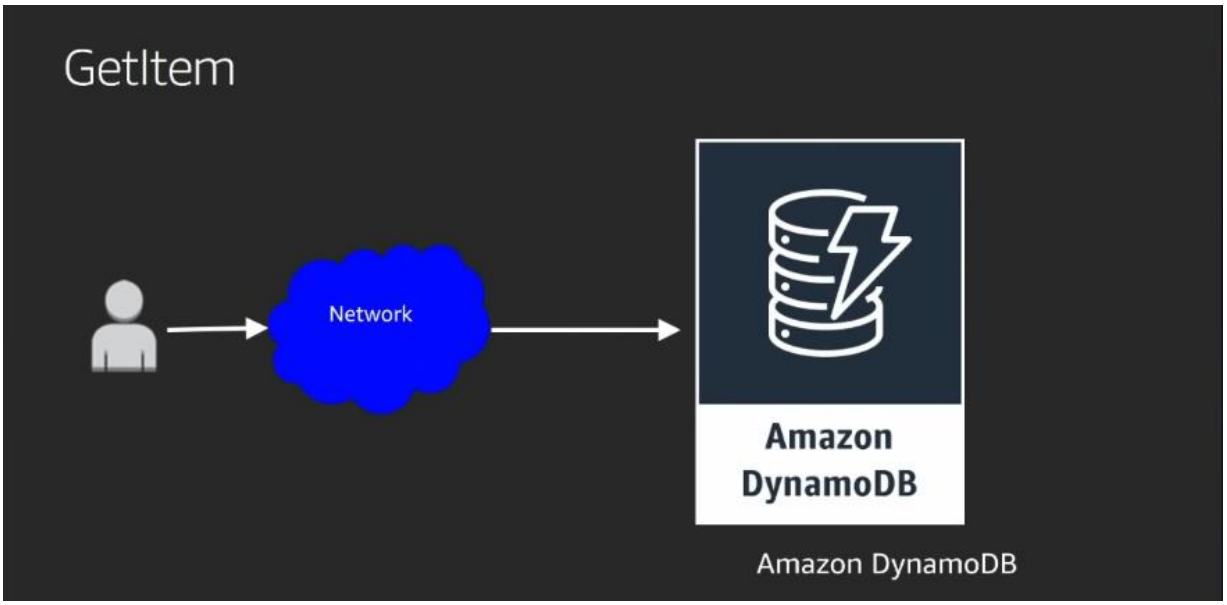
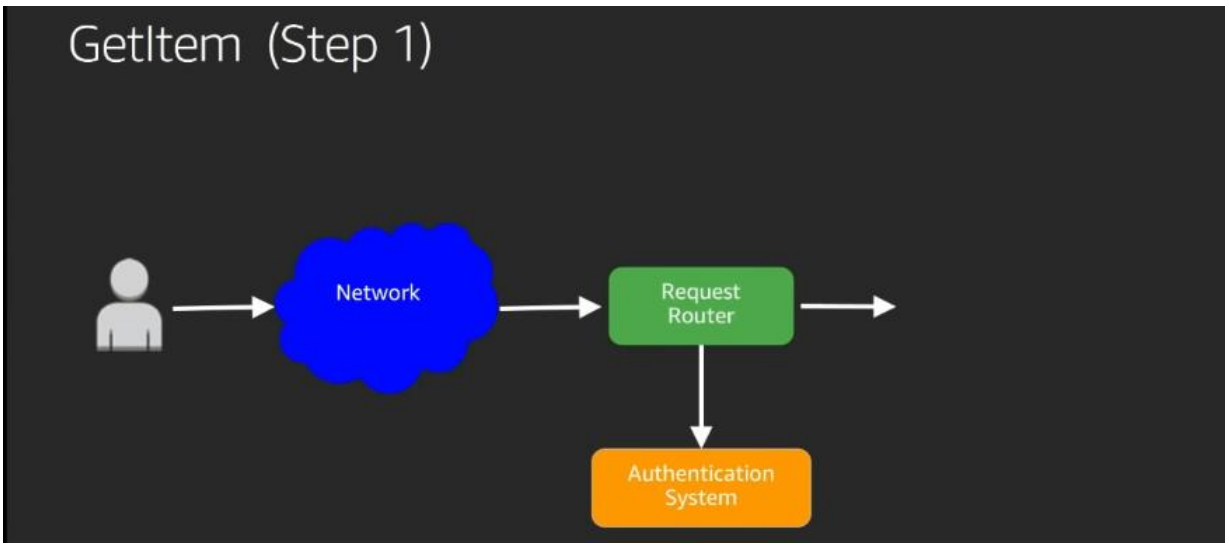


Come to this session to learn how **Amazon DynamoDB was built as the hyper-scale database for internet-scale applications**. In January 2012, Amazon launched DynamoDB, a cloud-based No-SQL database service designed from the ground up to support extreme scale, with the security, availability, performance, and manageability needed to run mission-critical workloads. This session discloses for the first time the underpinnings of DynamoDB, and how we run a fully managed non-relational database used by more than 100,000 customers. We cover the underlying technical aspects of how an application works with DynamoDB for authentication, metadata, storage nodes, streams, backup, and global replication.

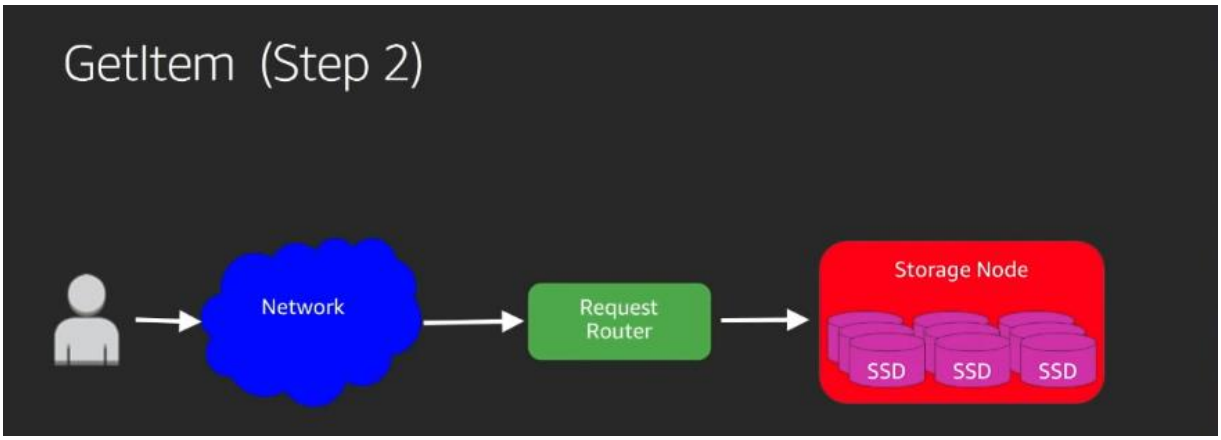




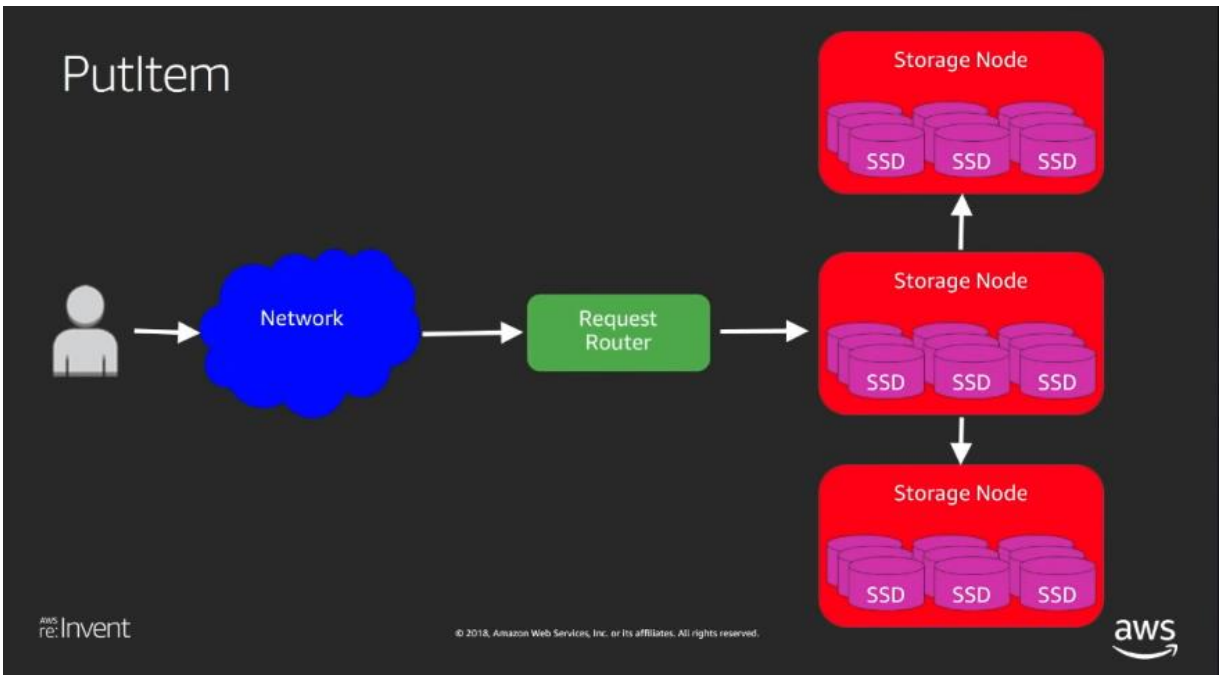
The Request Router is the public facing API for DynamoDB that does things like authentication of the request's caller to make sure they can do what they want to do. We use this same Authentication system for all our AWS services at the entry point.

Sample Policy

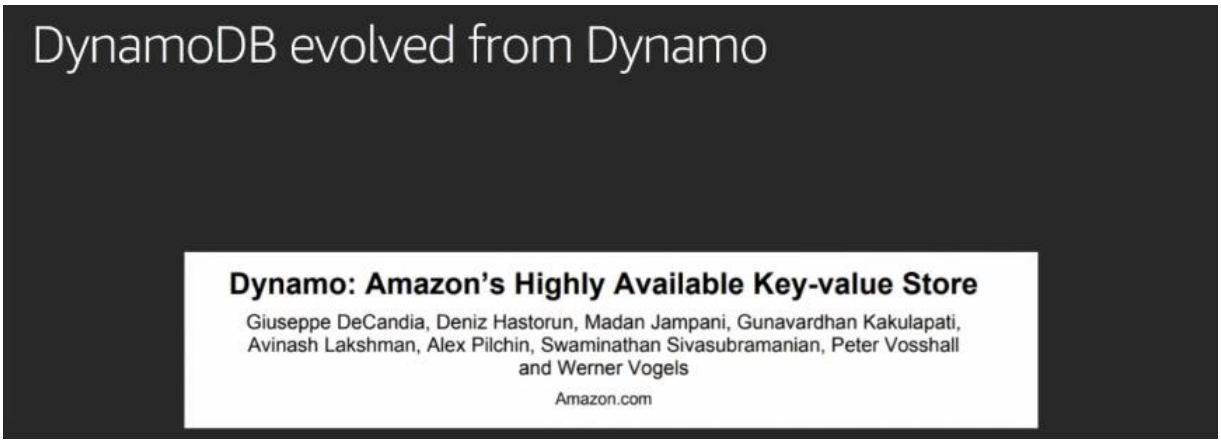
```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [
    "dynamodb:GetItem"
   ],
   "Resource": [
    "arn:aws:dynamodb:us-west-2:1234567890:table/customer"
   ],
   "Condition": {
    "ForAllValues:StringEquals": {
     "dynamodb:LeadingKeys": [
      "${www.amazon.com:user_id}"
     ]
    }
   }
  }
 ]
}
```



It then makes a call out to our storage nodes for the actual data you are asking for.

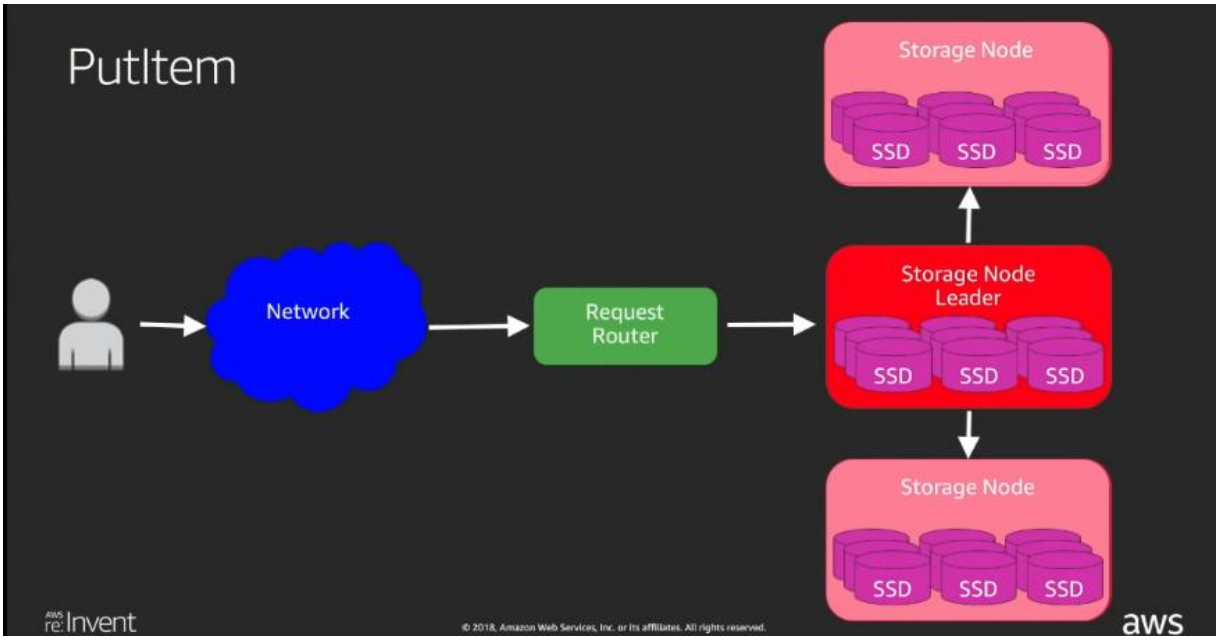


A Put is a little bit more complicated because we need to durably store the data by writing to 3 storage nodes and waits for 1 node to acknowledge the request was successfully stored.



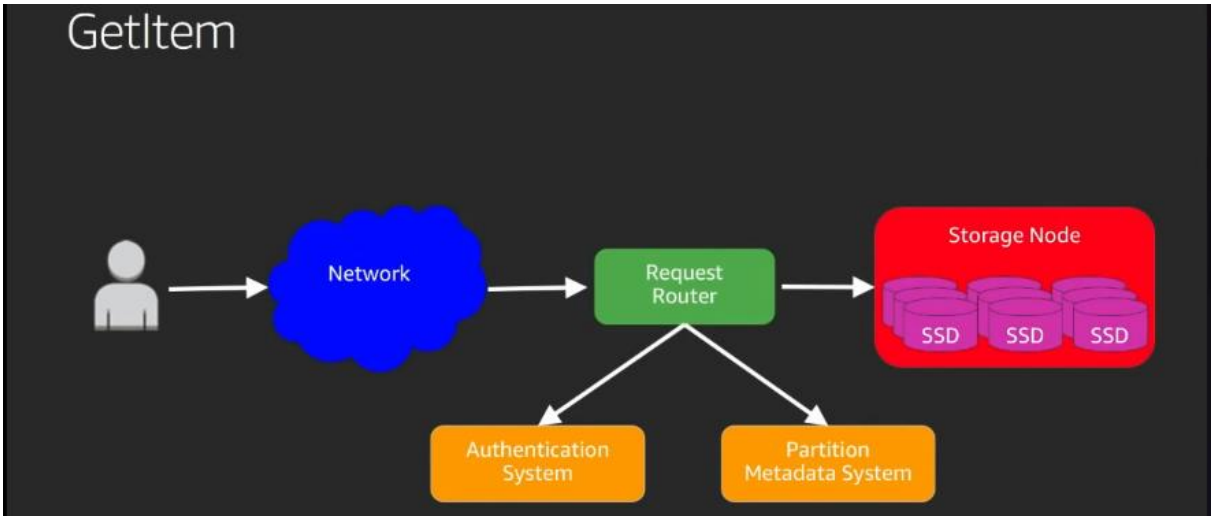
DynamoDB has evolved from the Dynamo described in this paper, but instead used Paxos

Paxos is an algorithm for to make a set of nodes agree on something



All the DynamoDB storage nodes run Paxos and we are trying to get agreement on a leader for a partition or table. The Leader is always up-to-date and accepts all the data writes and propagates the data to other follower nodes for the partition. A leader is always sending back a periodic heartbeat, failure to do so will cause a new leader election to take place.

A Request Router (RR) is stateless so it doesn't matter which one of the RR boxes the request gets routed through. The RR then makes a call to the leader storage node of the partition the request needs to talk to and the leader gets to talk to the other related follower nodes for data replication.



The Partition Metadata Service is a service that holds the leader-follower nodes information for all the data partitions

Tables

	CustID	Customer Information
	145783	{ name:"Bob", city:"London", ...}
	236294	{ name:"Sara", city:"Tampa", ...}
	333363	{ name:"Betty", city:"Madison", ...}
	445104	{ name:"James", city:"Miami", ...}
	523422	{ name:"Alex", city:"London", ...}
	643145	{ name:"Val", city:"Seattle", ...}
	723342	{ name:"Jeff", city:"Toledo", ...}

When you make a table in DynamoDB, you need to tell it your primary hash key for the table like the CustID above.

Hashing

Hash Value	CustID	Customer Information
0x9531	145783	{ name:"Bob", city:"London", ...}
0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}
0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}

DynamoDB then computes a hash for the primary key

Partitioning

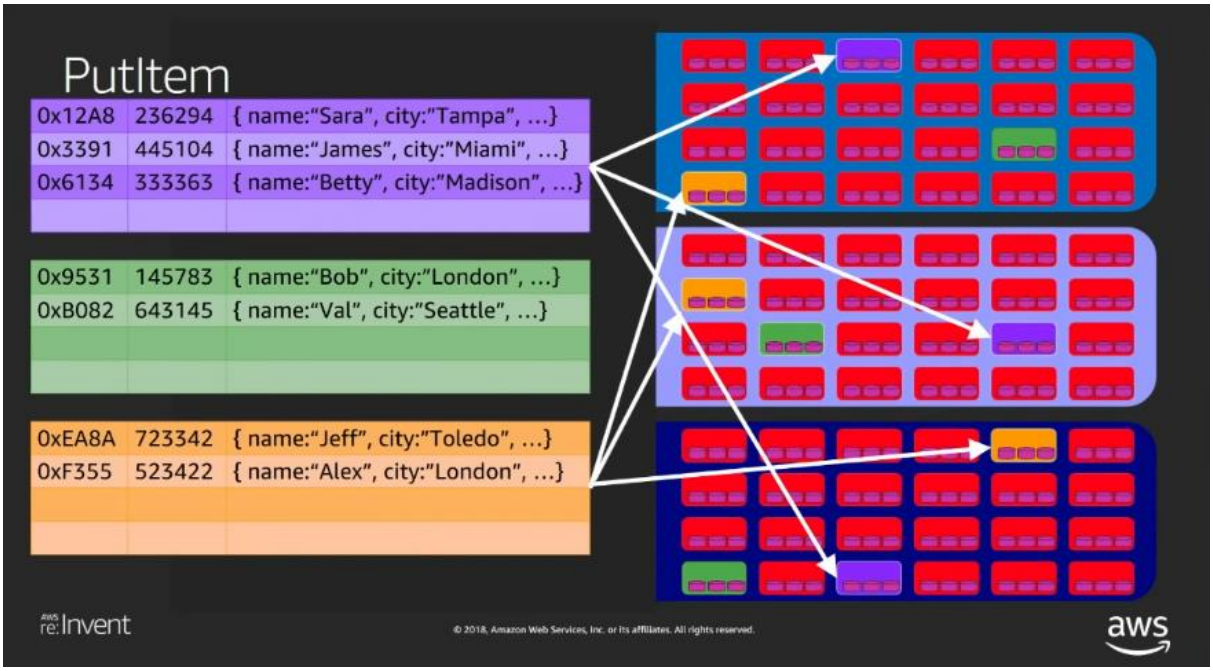
Hash Value	CustID	Customer Information
0x9531	145783	{ name:"Bob", city:"London", ...}
0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}
0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}

0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}

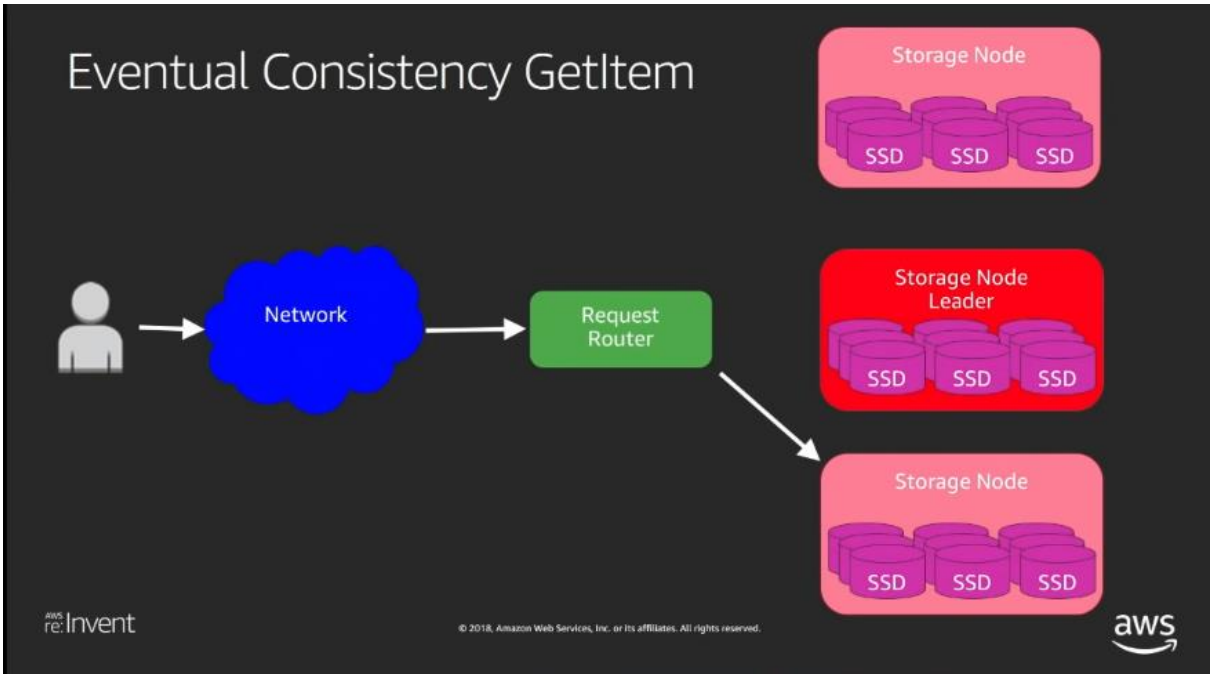
0x9531	145783	{ name:"Bob", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}

0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}

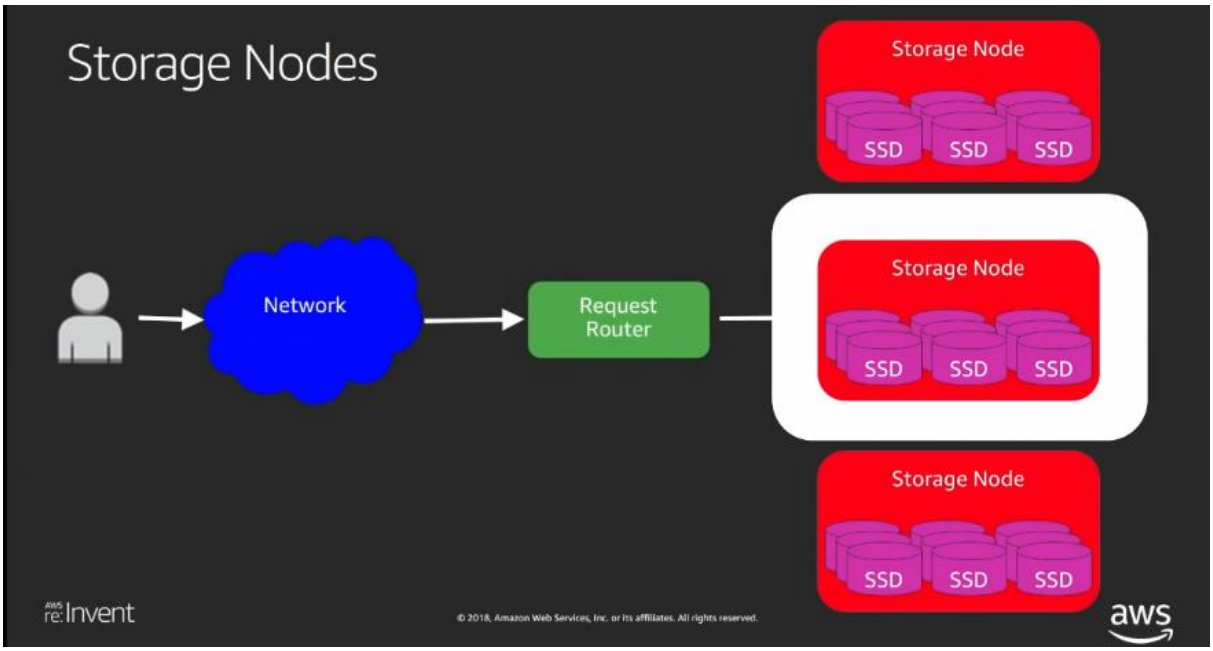
Then uses the hash to sort the data into partitions as above



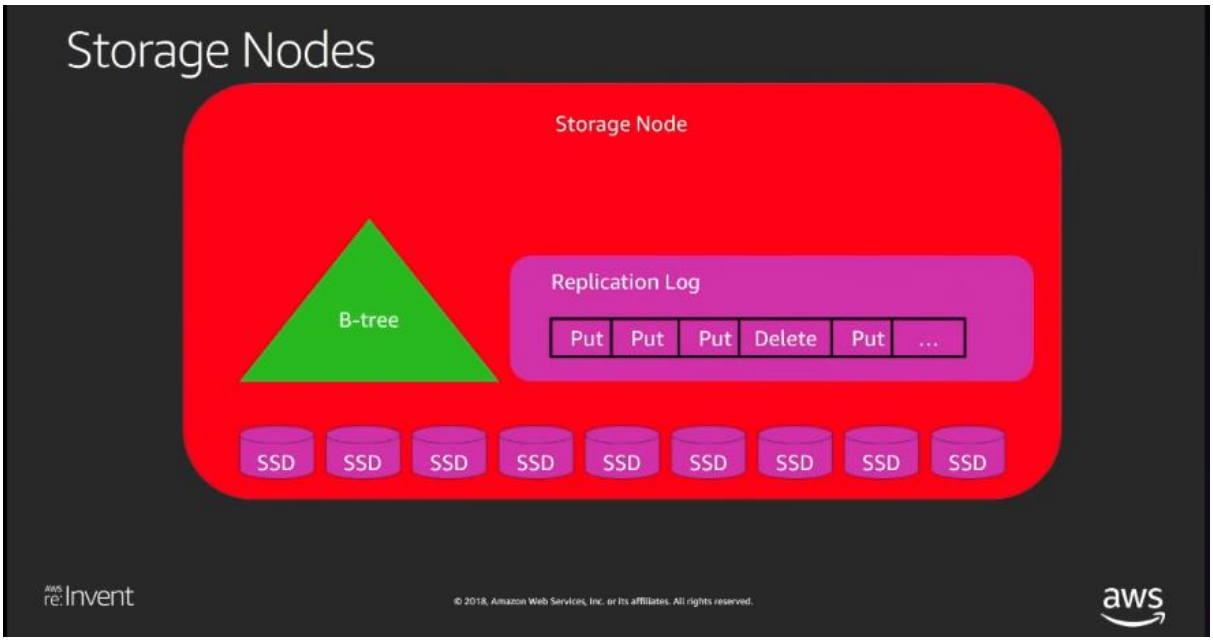
The partitions then get mapped out to the storage nodes for the AZ the customer specifies.



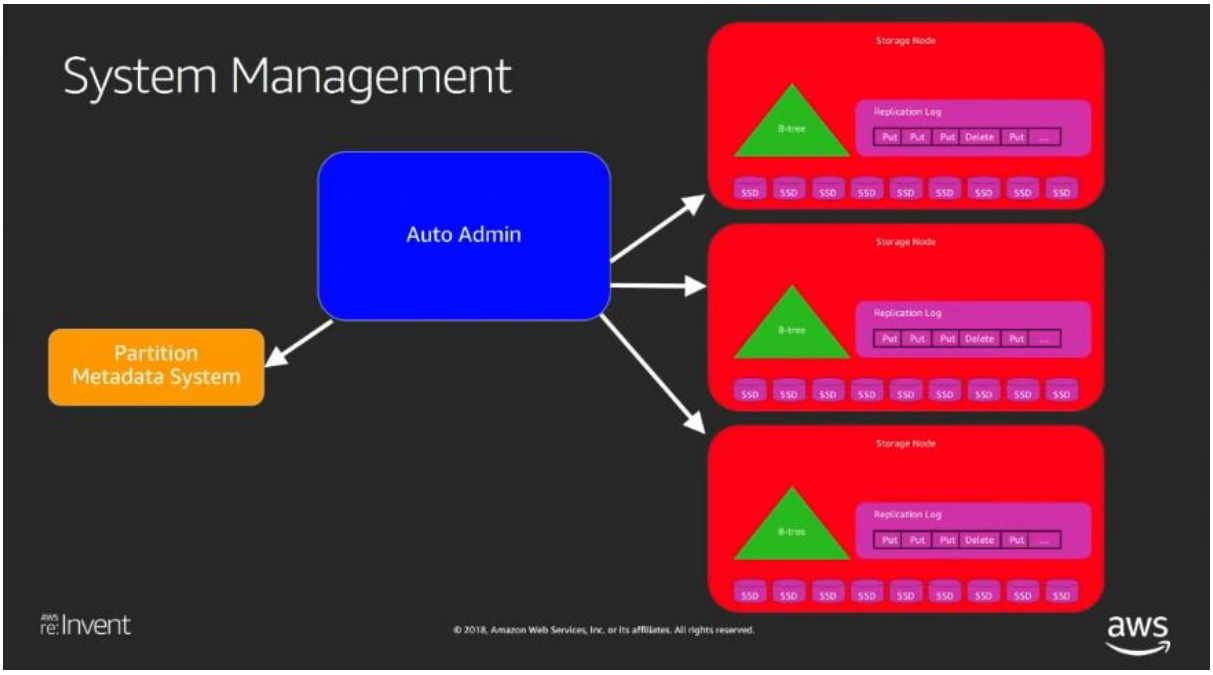
For eventually consistent read requests, we let the RR randomly choose the storage nodes holding the data, this means that you may not get the most recent version of your data in some cases.



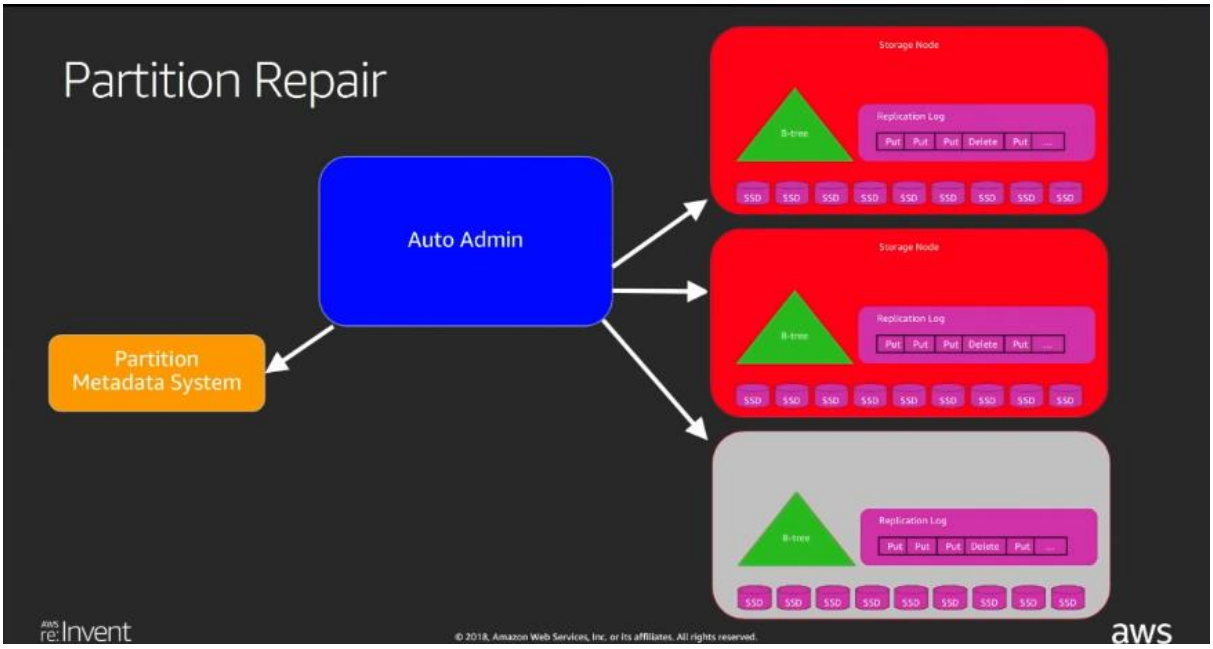
What happens inside the storage nodes?



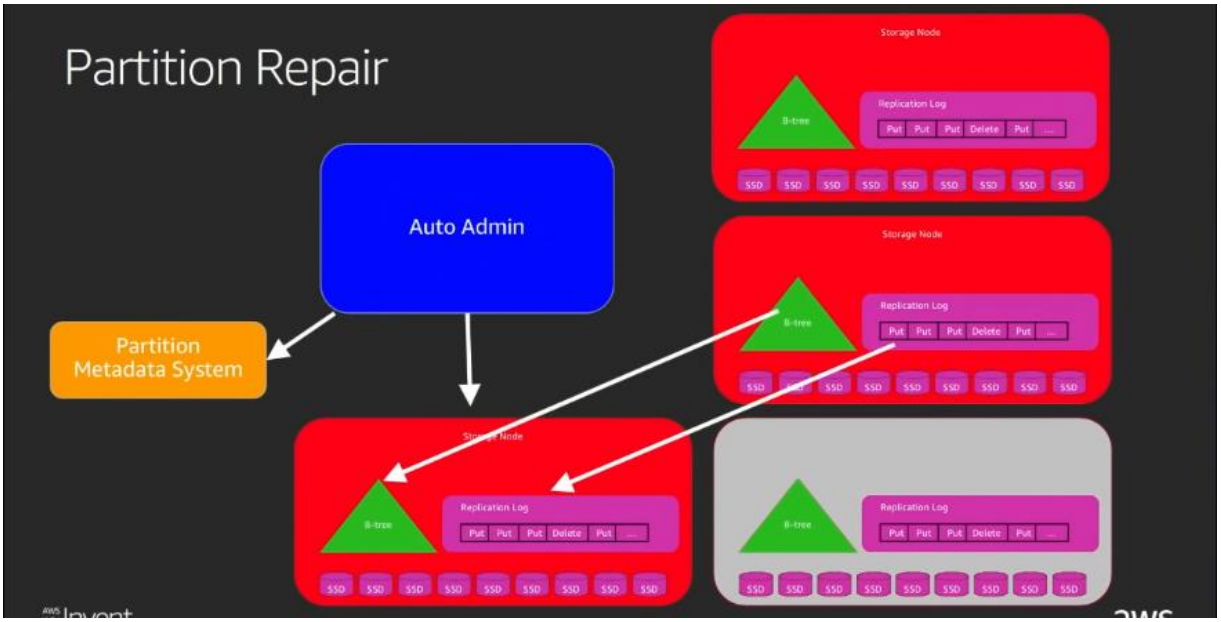
The B-Tree is where we do all the querying and scanning for the user data using the resulting B-Tree index. the replication log is recording every mutation to the data partition.



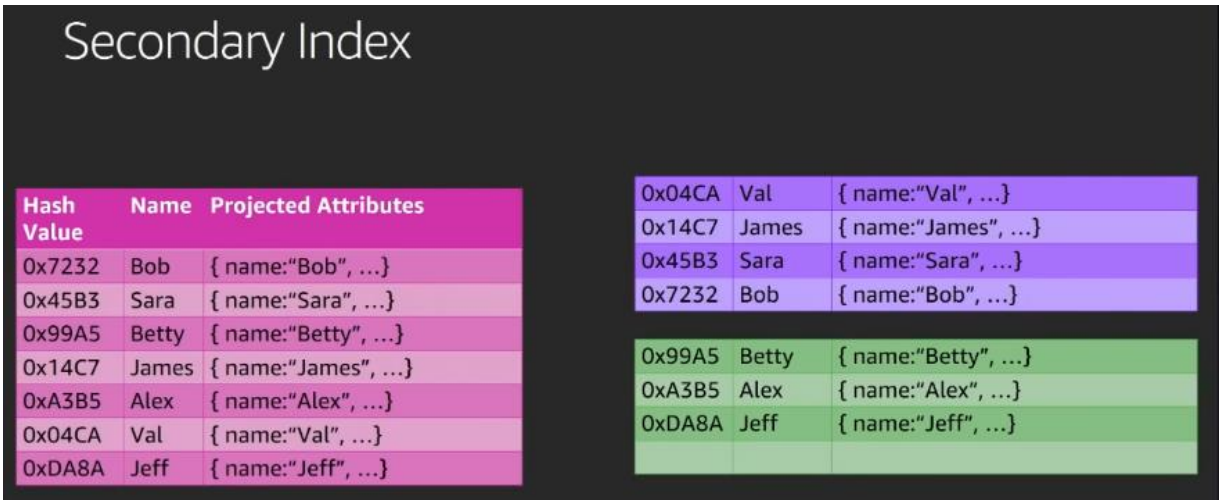
DynamoDB has a component called Auto Admin that does many things, like making sure the PMS is up-to-date with the leader location for all partitions.



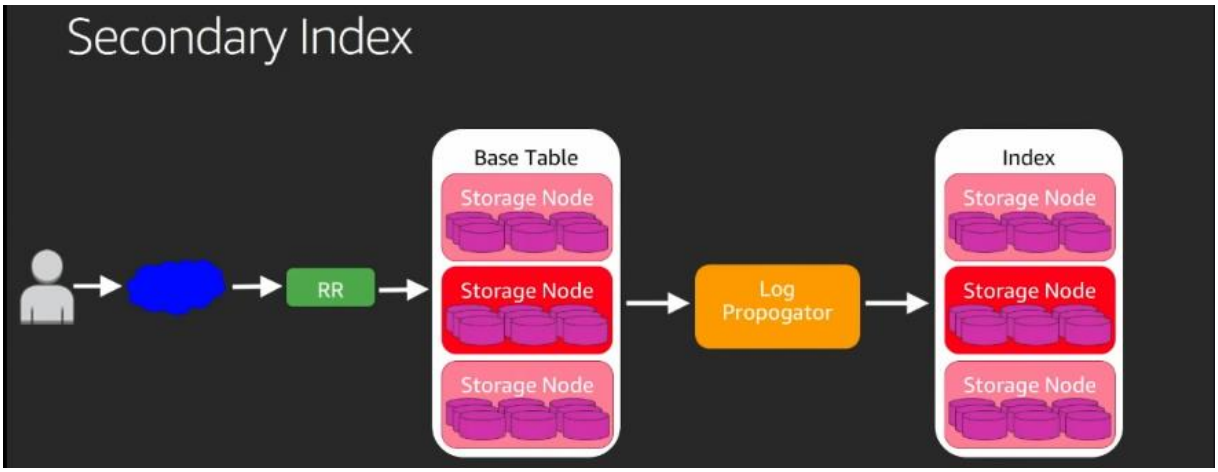
Auto-Admin also does partition repair using another storage node to take over that failed partition as below



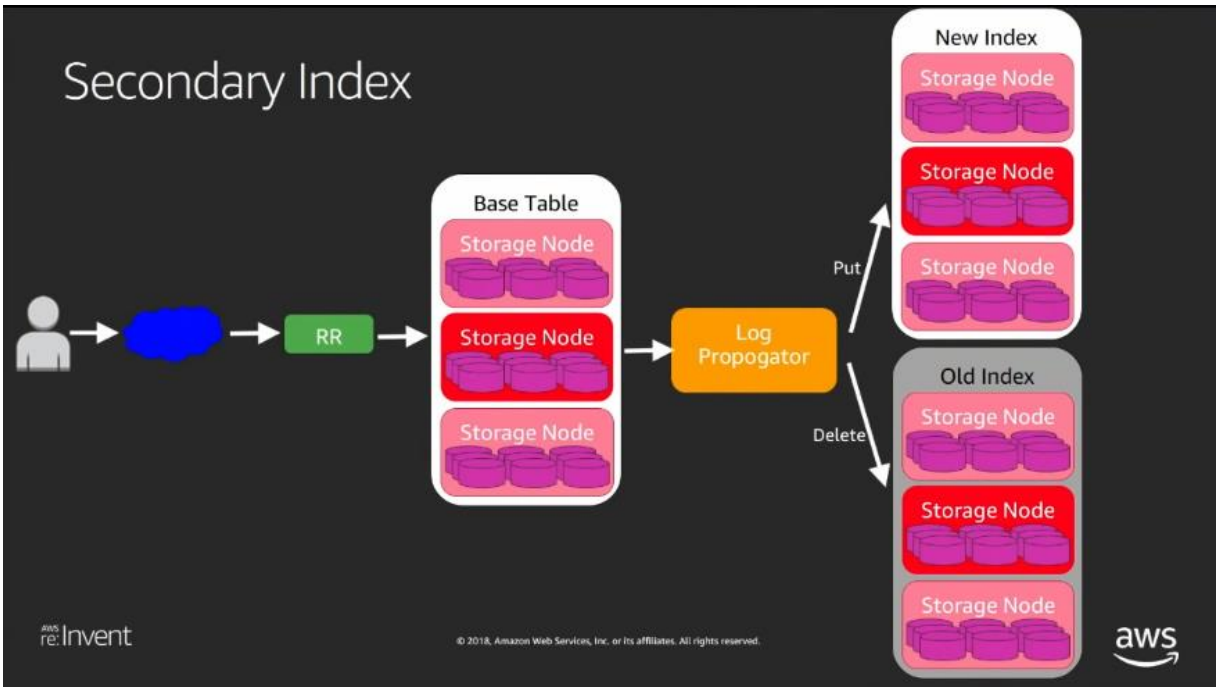
It then starts copying the B-Tree and replication log over to the new storage node, then it makes sure the replication log is applied to the new B-Tree to make sure the new node catches up to the leader.



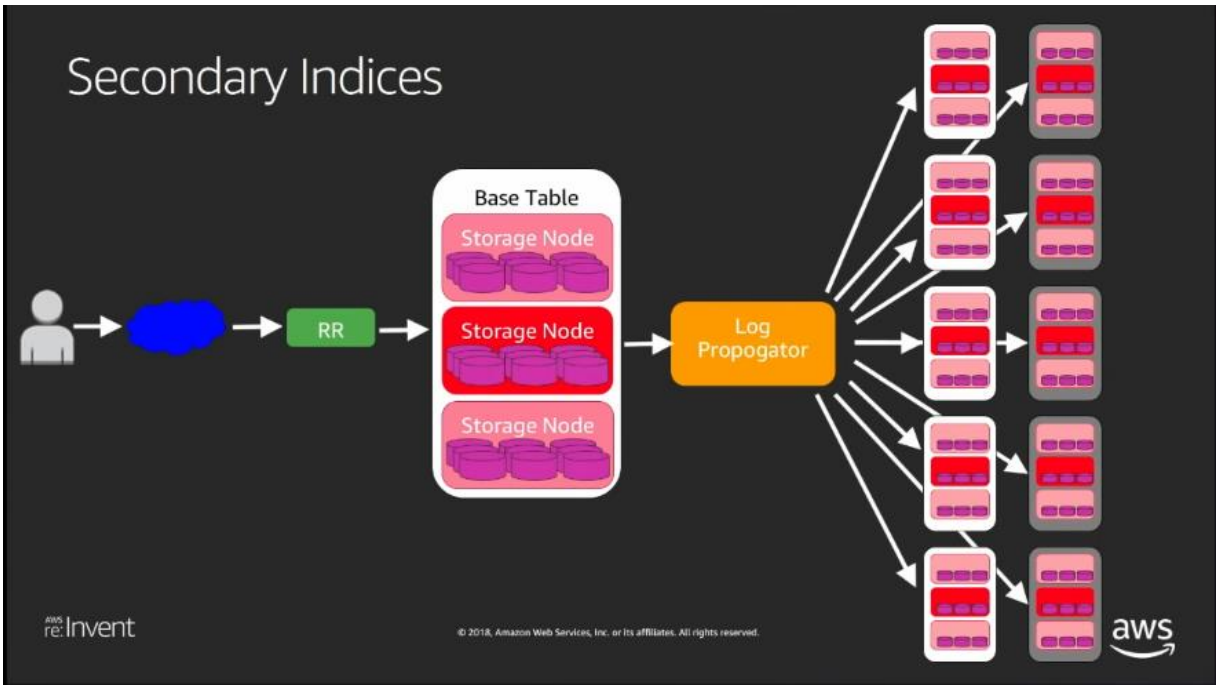
We can also build a secondary index on the name attribute for our item being stored in the table. The SI are independent of what happens with the base table.



The Log Propagator is watching the repl log and the storage nodes and executing a PUT when a secondary index changes on an SI partition



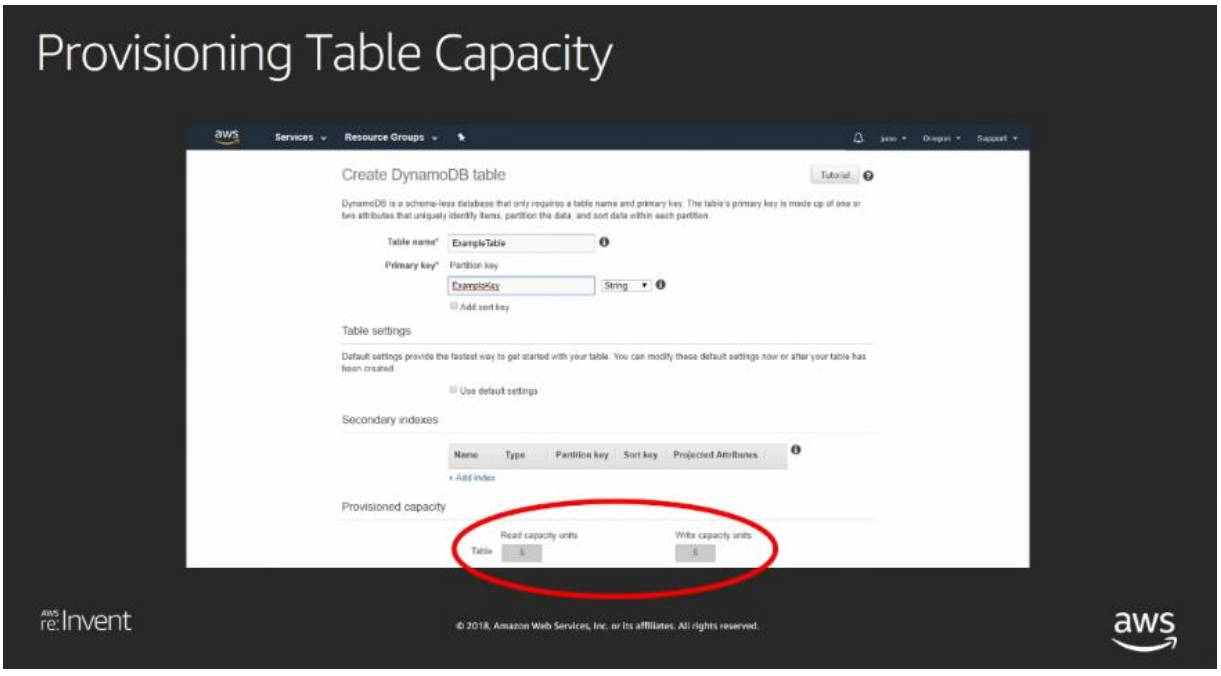
We update the value and rewrite the new value somewhere else



You can have up to 5 secondary indexes, this creates many possible updates and changes for changes

This is like the DBA for DynamoDB that does a lot of things.

This is about provisioning new tables in DynamoDB.



You need to provide the table name, primary key, and the read and write capacity units. 1 read capacity unit, RCU will allow you to read up to a 4k object.

Provisioning Trade-off

Functionality ⟷ $

Provisioning Example

300 Read Capacity Units (RCU)

Ignoring Writes

This is identical for the write side

Provisioning

0x12A8	236294	{ name:"Sara", city:"Tampa", …}
0x3391	445104	{ name:"James", city:"Miami", …}
0x6134	333363	{ name:"Betty", city:"Madison", …}

← 100 RCUs

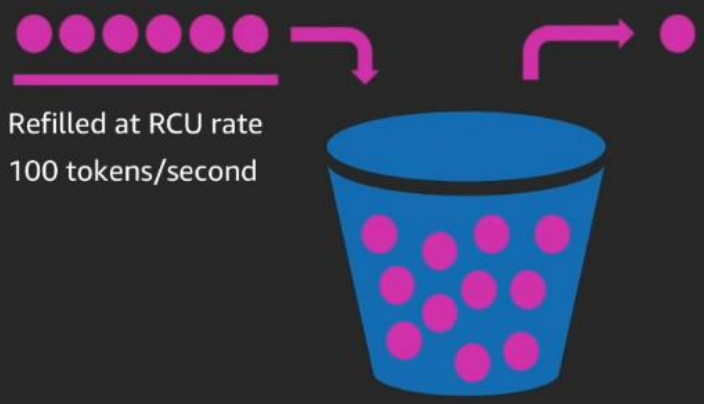
0x9531	145783	{ name:"Bob", city:"London", …}
0xB082	643145	{ name:"Val", city:"Seattle", …}

← 100 RCUs

0xEA8A	723342	{ name:"Jeff", city:"Toledo", …}
0xF355	523422	{ name:"Alex", city:"London", …}

← 100 RCUs

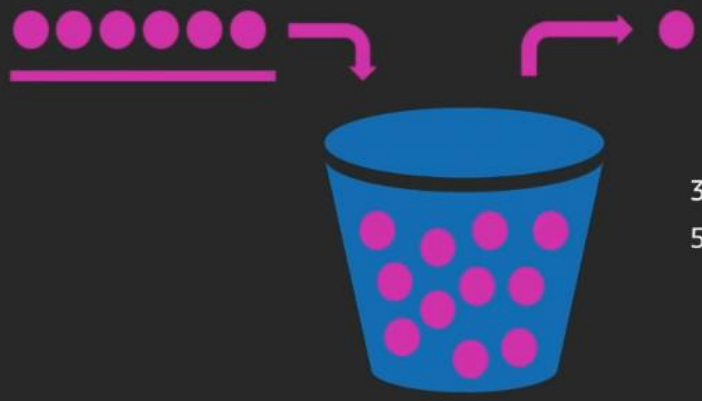
Token Bucket Algorithm



Emptied 1 Token per Request*

*Tokens deducted depends on Item size

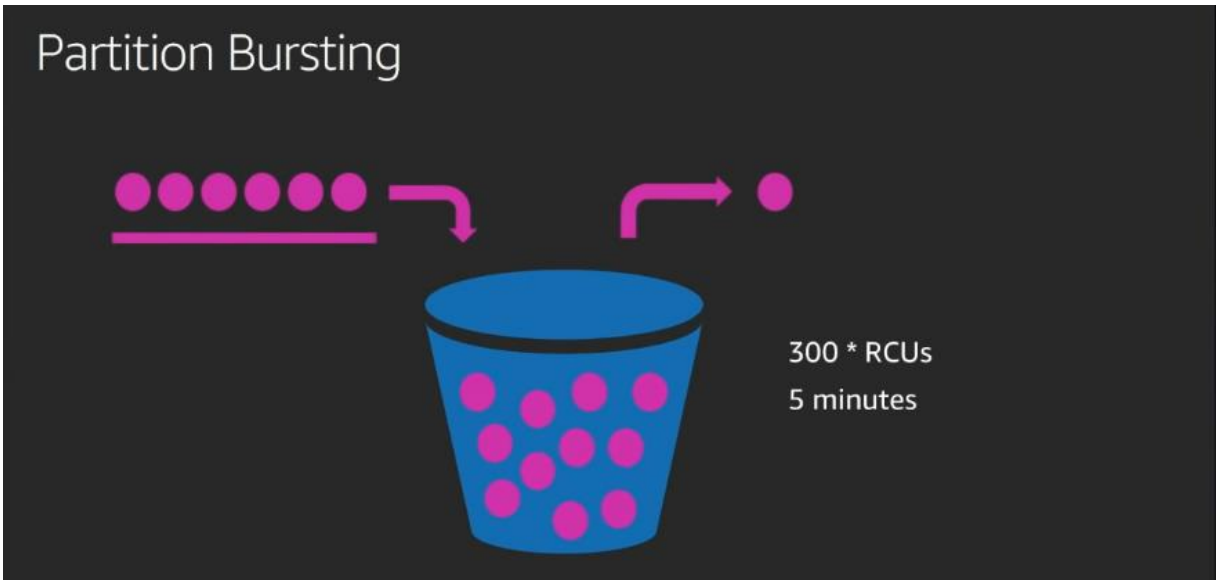
Token Bucket Capacity

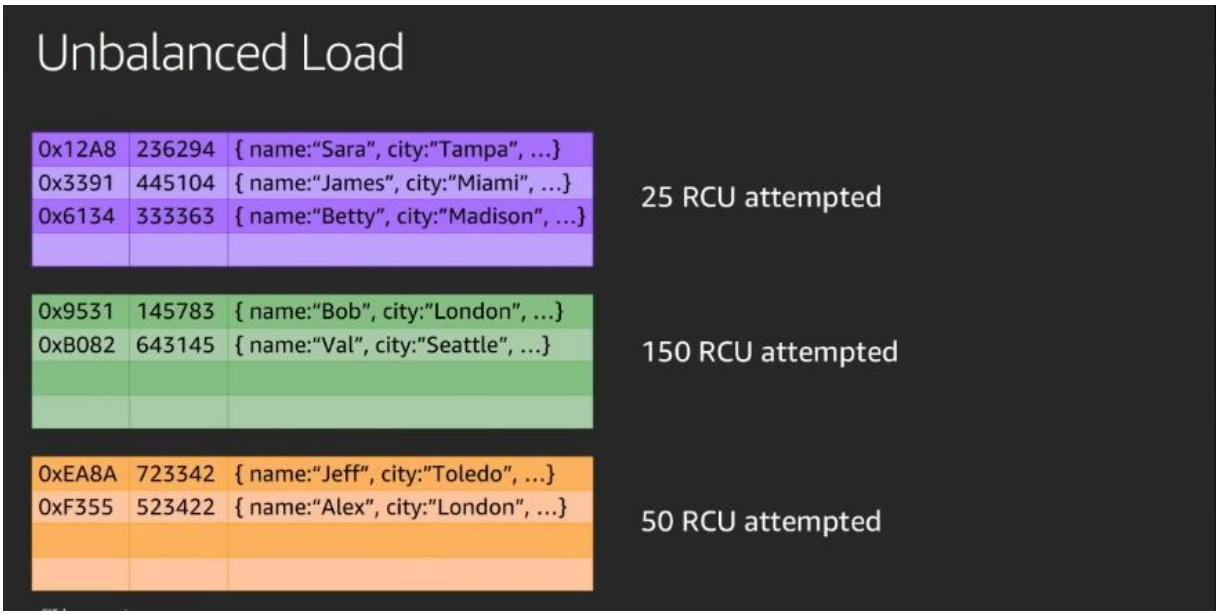


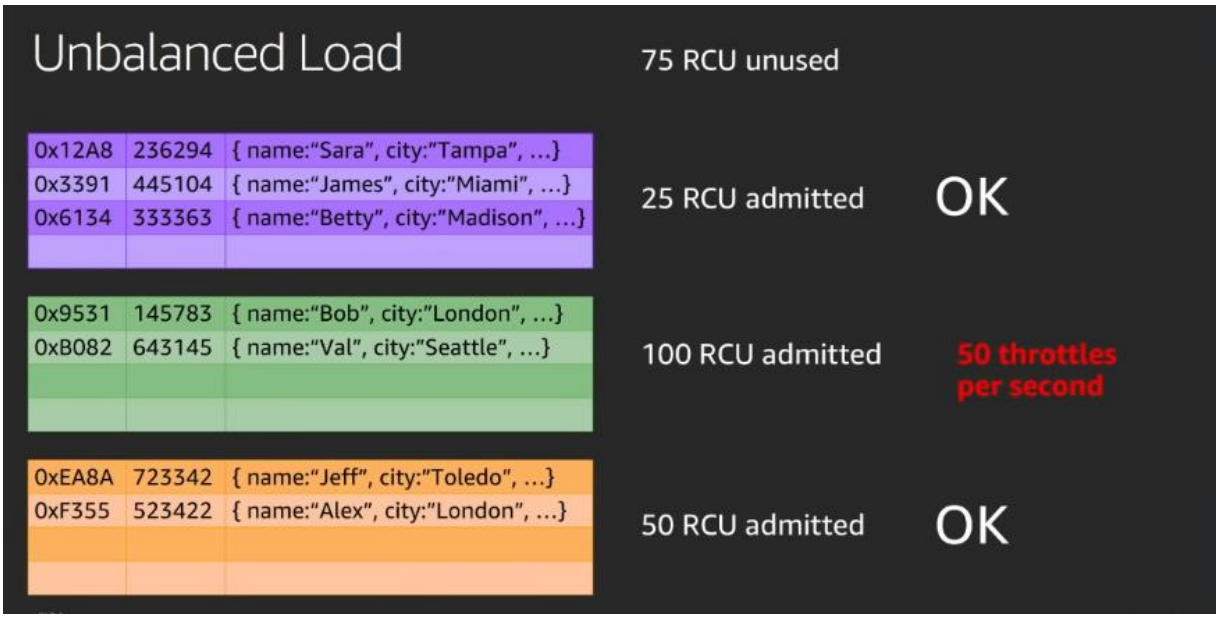
300 * RCUs

5 minutes

Full Token Bucket

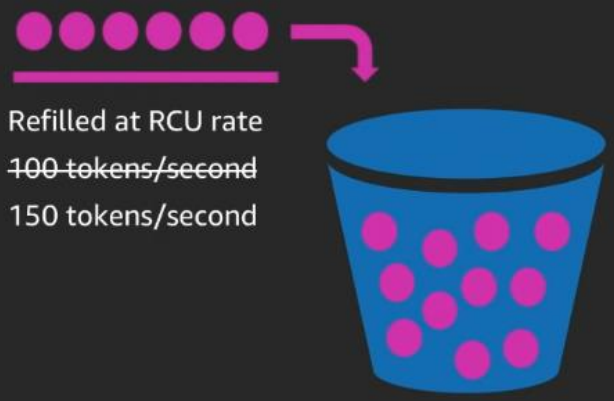






Hot partitions will get throttled as above

Adaptive Capacity

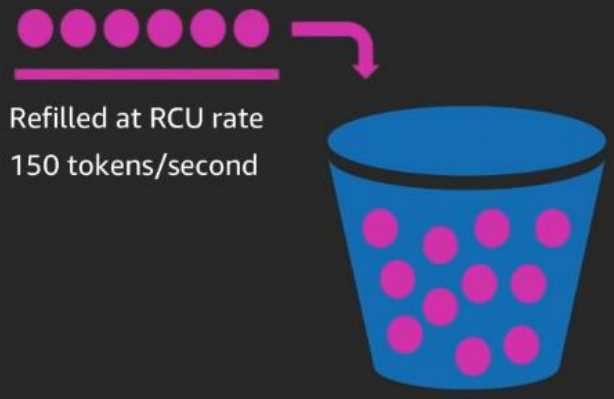


Refilled at RCU rate

~~100 tokens/second~~

150 tokens/second

Adaptive Capacity



Refilled at RCU rate

150 tokens/second

Adaptive Capacity

Multiplier = 1.5

Adaptive Capacity Active

75 RCU unused

0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}

25 RCU admitted OK

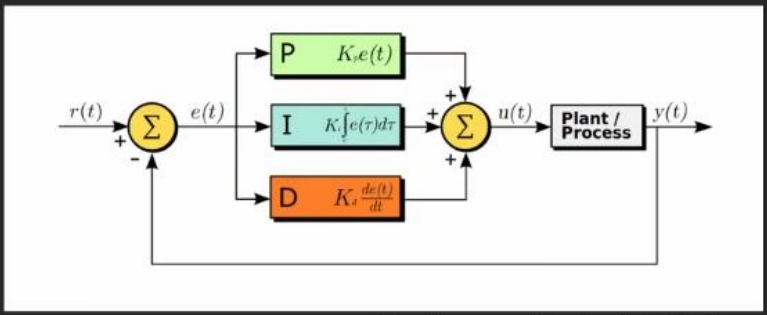
0x9531	145783	{ name:"Bob", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}

150 RCU admitted OK

0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}

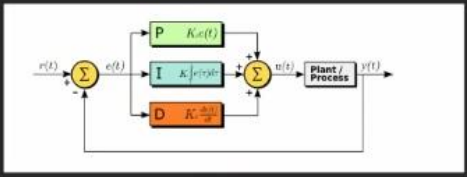
50 RCU admitted OK

PID Controller



https://commons.wikimedia.org/wiki/File:PID.svg

PID Controller



https://commons.wikimedia.org/wiki/File:PID.svg

PID inputs:

- Consumed Capacity
- Provisioned Capacity
- Throttling Rate
- Current Multiplier

PID output:

- New Multiplier

We apply this to the partitions in your table to determine the adaptive capacity to apply at each time

Too Much Allowed

150 RCU over provisioned

0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}

150 RCU admitted

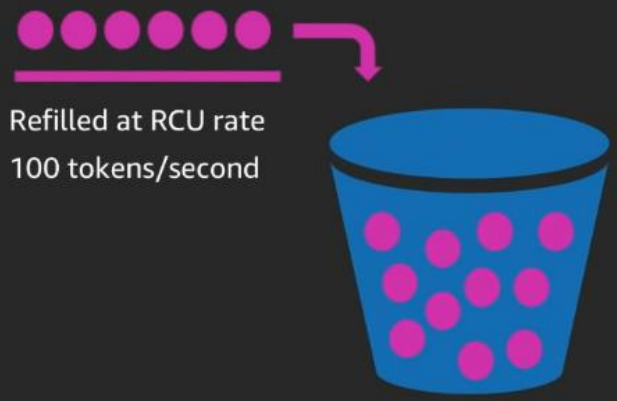
0x9531	145783	{ name:"Bob", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}

150 RCU admitted

0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}

150 RCU admitted

Partition Bursting



Refilled at RCU rate

100 tokens/second

Adaptive Capacity

~~Multiplier = 1.5~~

Multiplier = 1.0

Throttling

0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}

100 RCU admitted 50 Throttled

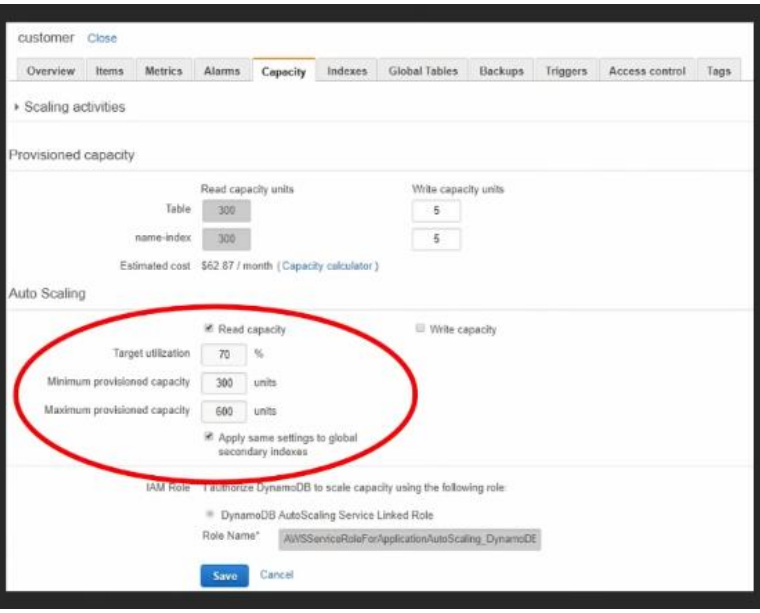
0x9531	145783	{ name:"Bob", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}

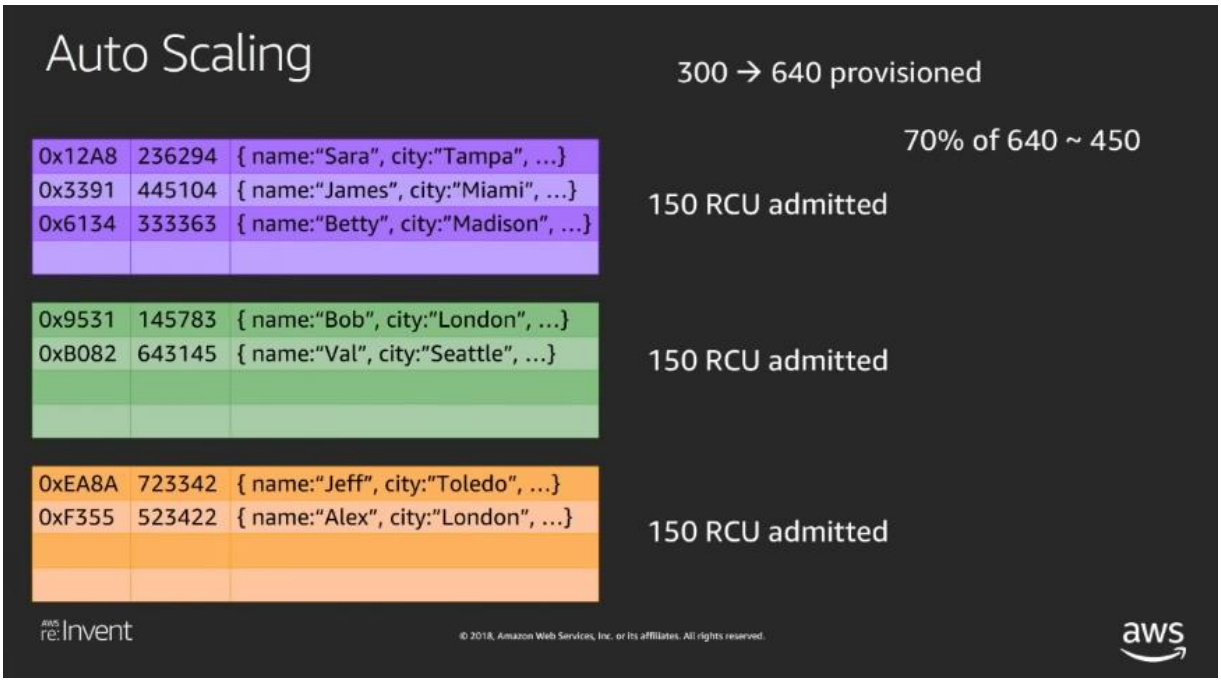
100 RCU admitted 50 Throttled

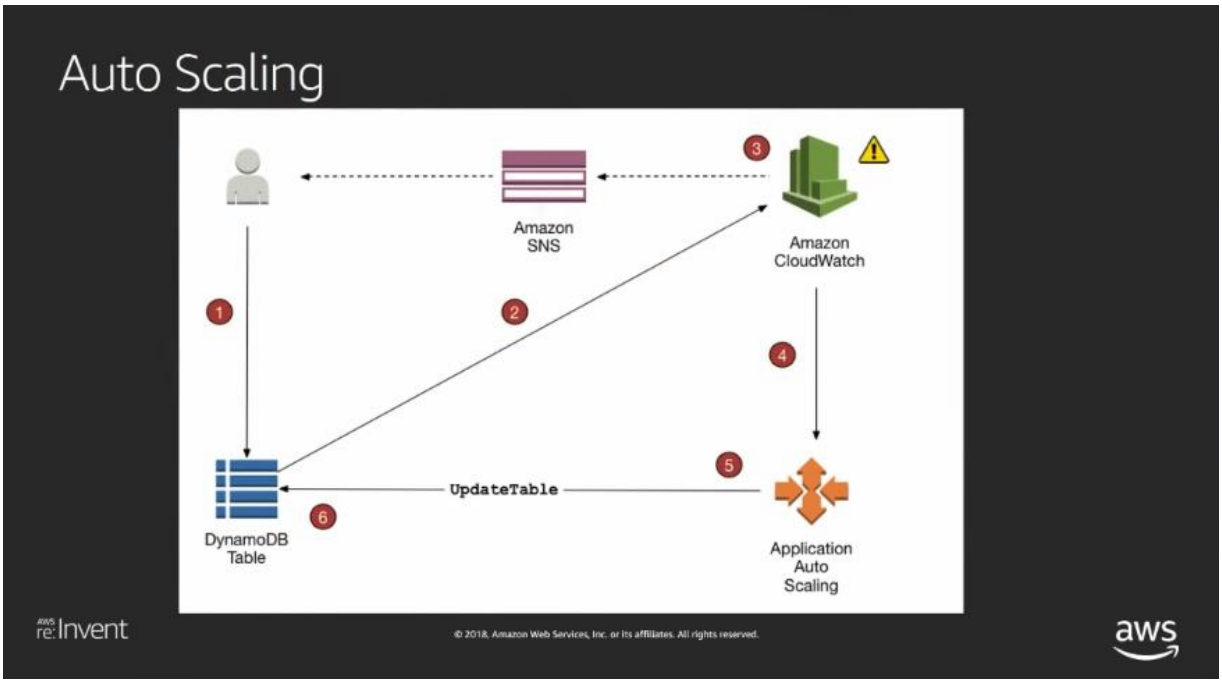
0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}

100 RCU admitted 50 Throttled

Auto Scaling







Auto-Scaling sets 2 alarms per table in CloudWatch that you can use for reads, for the provisioned value and for the target provisioned value.

Auto Scaling

450 → 43 provisioned

70% of 43 ~ 30

0x12A8	236294	{ name:"Sara", city:"Tampa", …}
0x3391	445104	{ name:"James", city:"Miami", …}
0x6134	333363	{ name:"Betty", city:"Madison", …}

10 RCU Consumed

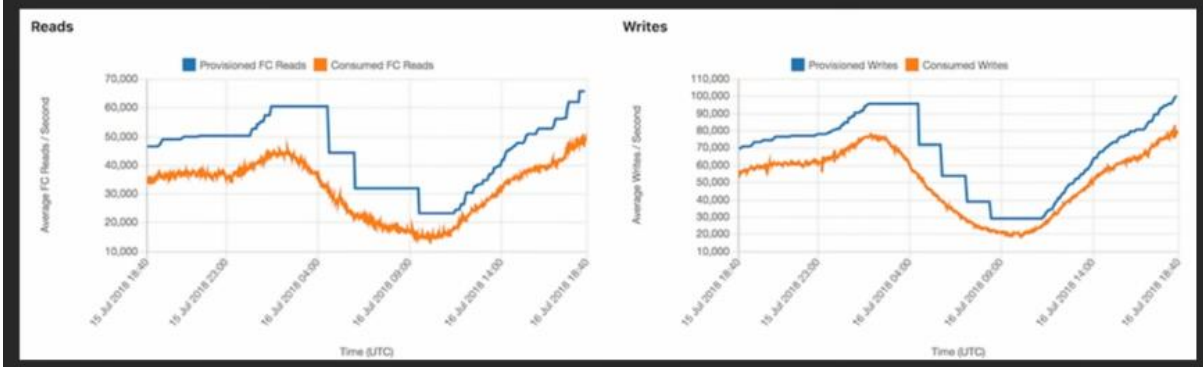
0x9531	145783	{ name:"Bob", city:"London", …}
0xB082	643145	{ name:"Val", city:"Seattle", …}

10 RCU Consumed

0xEA8A	723342	{ name:"Jeff", city:"Toledo", …}
0xF355	523422	{ name:"Alex", city:"London", …}

10 RCU Consumed

Actual Auto-scaled Table



Provisioning Recap

- Balanced Usage
- Imbalance in Time — Bursting
- Imbalance in Key Space — Adaptive Capacity
- Changing Workloads — Auto Scaling

We will continue to iterate

Backup And Restore

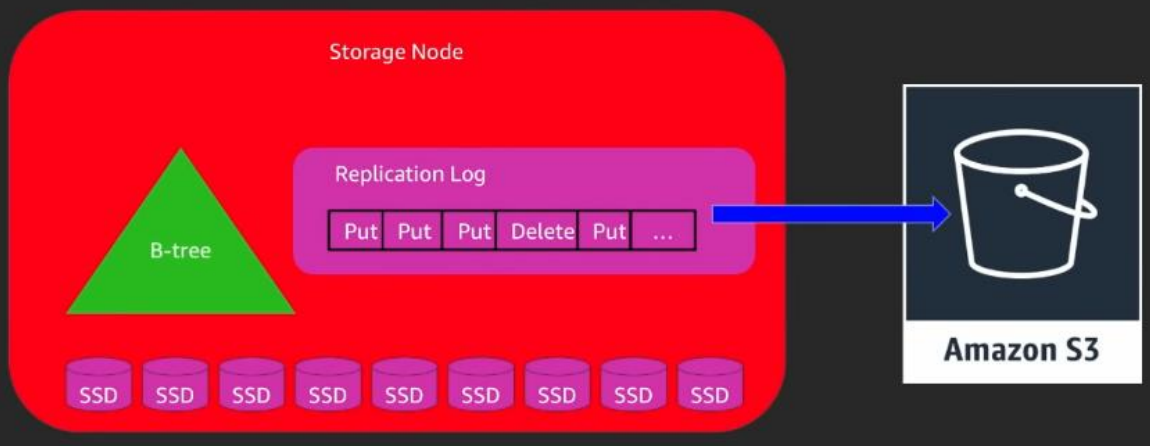
Backup and Restore

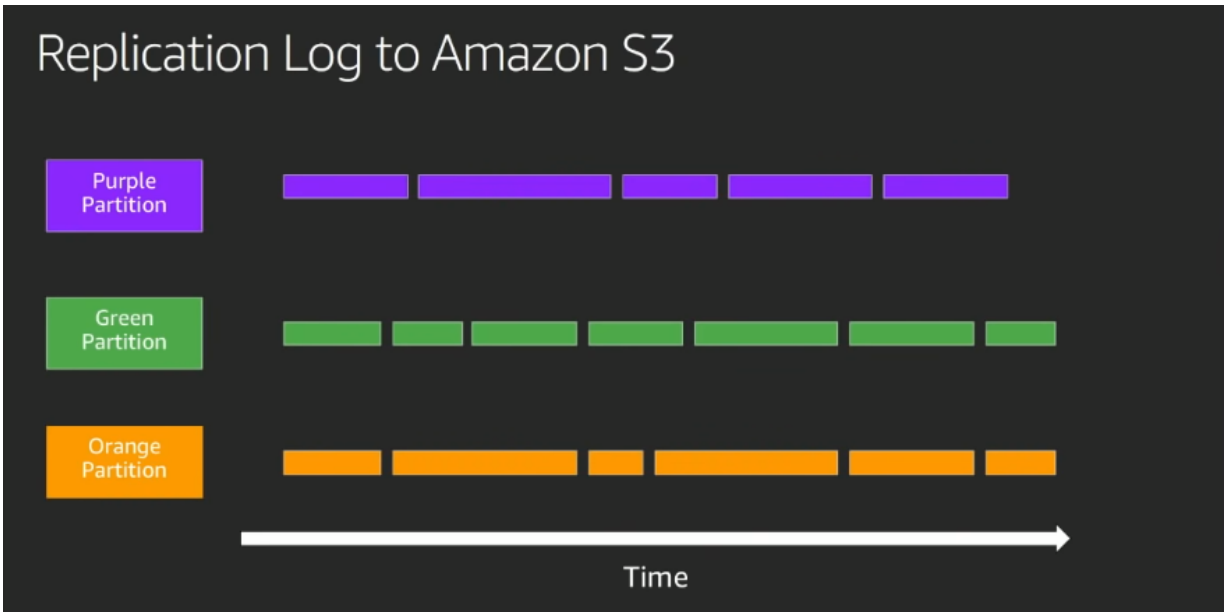
- Point in Time
- On demand Backups

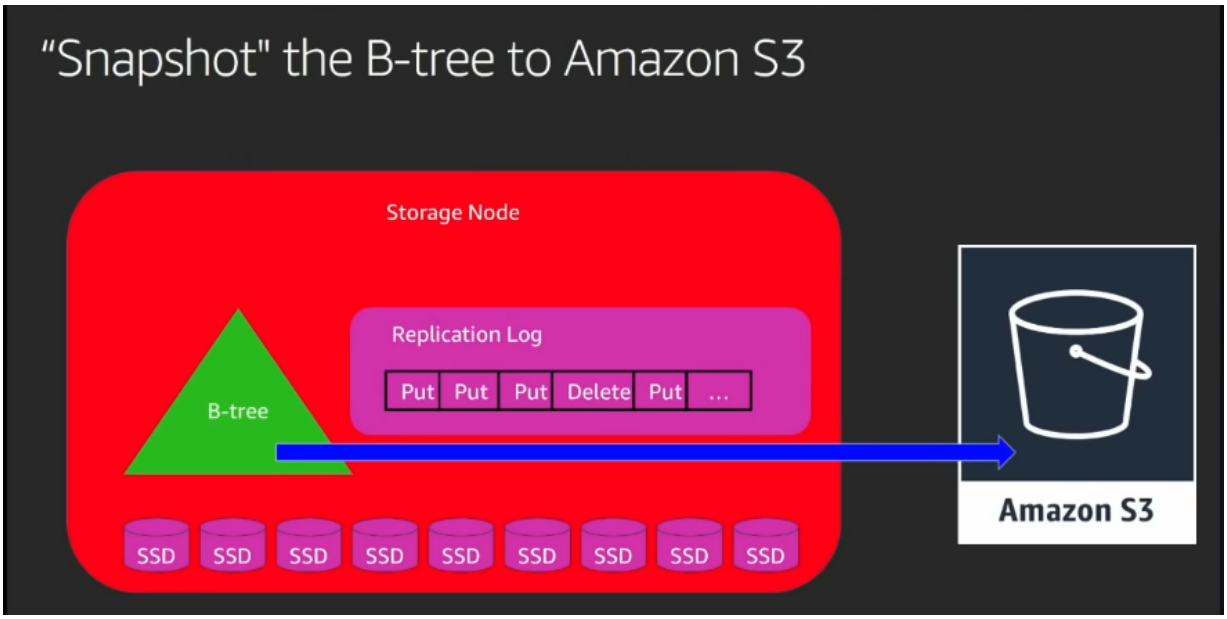
Q: Where to durably store backups?

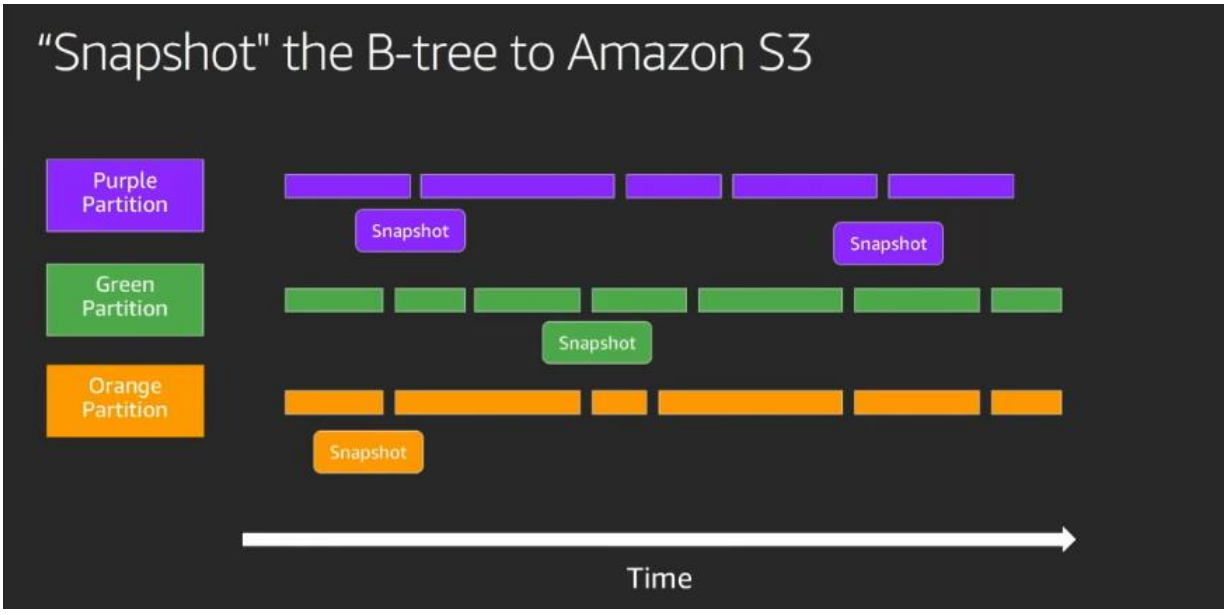
A: Amazon Simple Storage Service (Amazon S3)

Replication Log to Amazon S3

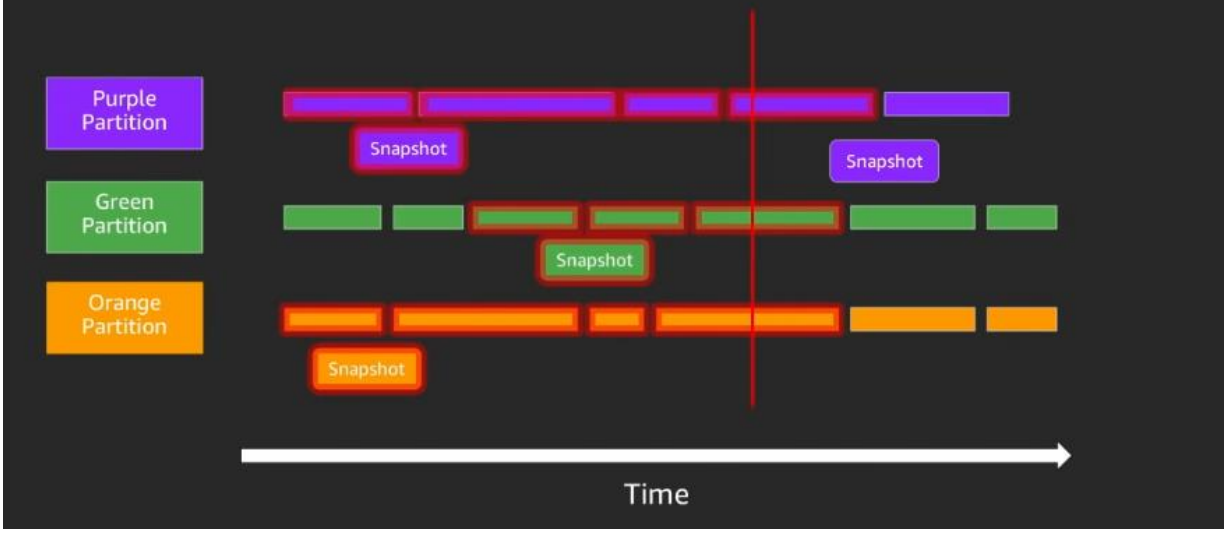



Replication Log to Amazon S3
Purple Partition
Green Partition
Orange Partition
Time

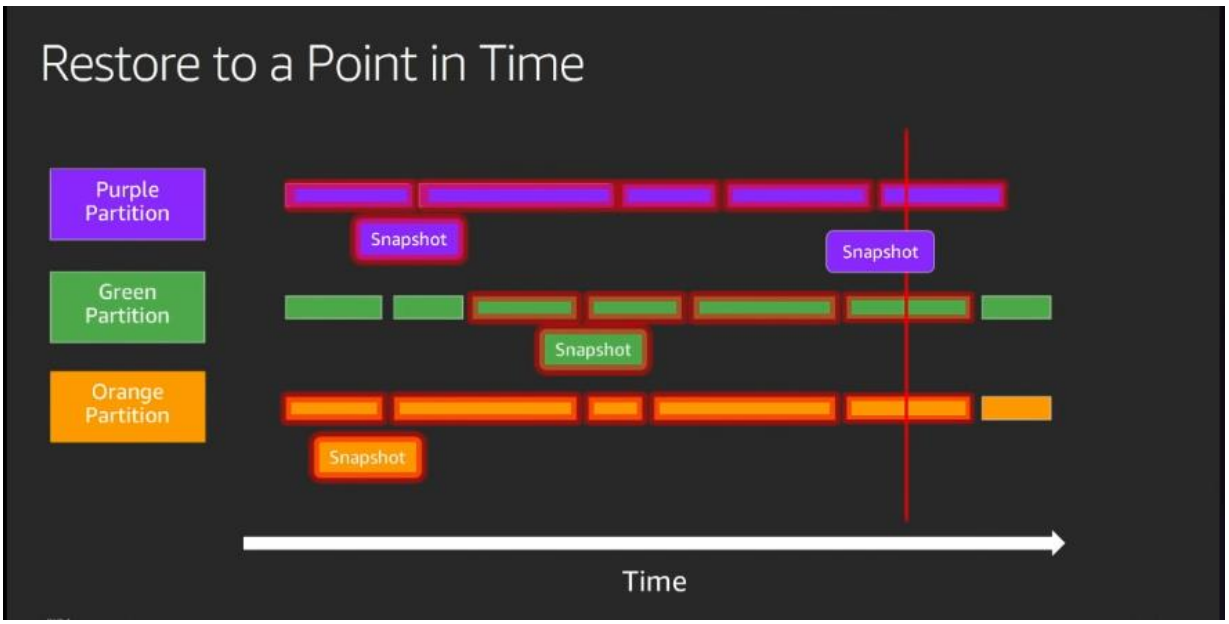

"Snapshot" the B-tree to Amazon S3
Storage Node
Replication Log
Put
Put
Put
Delete
Put
...
B-tree
SSD
SSD
SSD
SSD
SSD
SSD
SSD
SSD
SSD
Amazon S3


"Snapshot" the B-tree to Amazon S3
Purple Partition
Green Partition
Orange Partition
Snapshot
Snapshot
Snapshot
Snapshot
Time

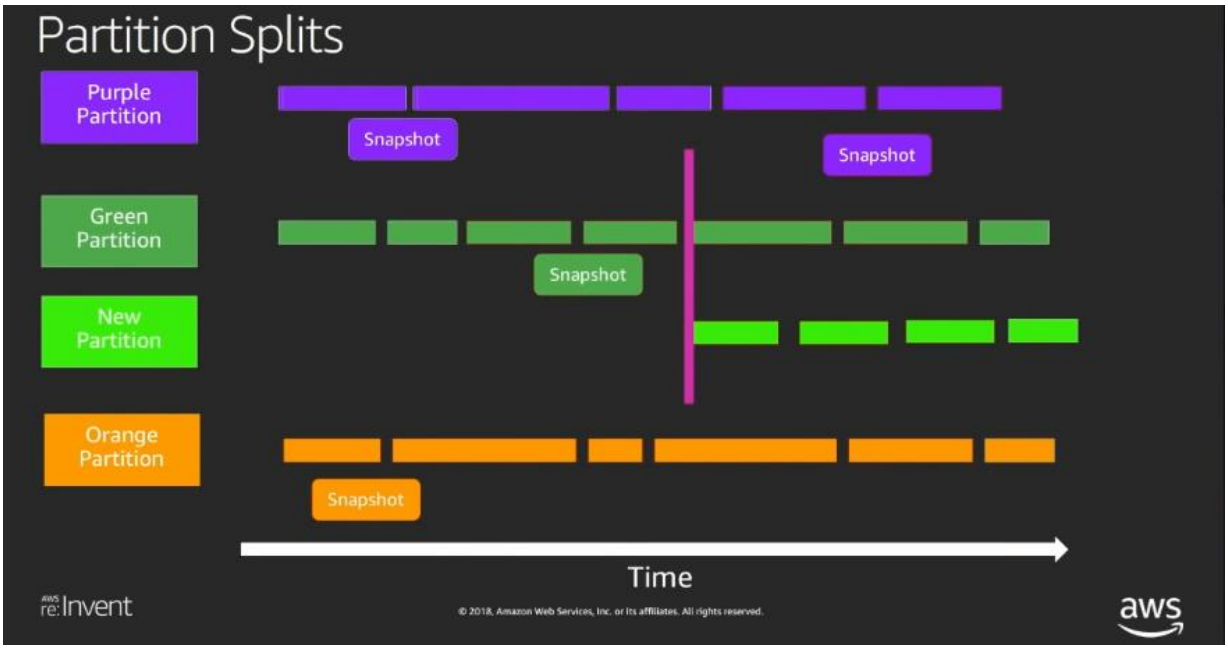
Restore to a Point in Time

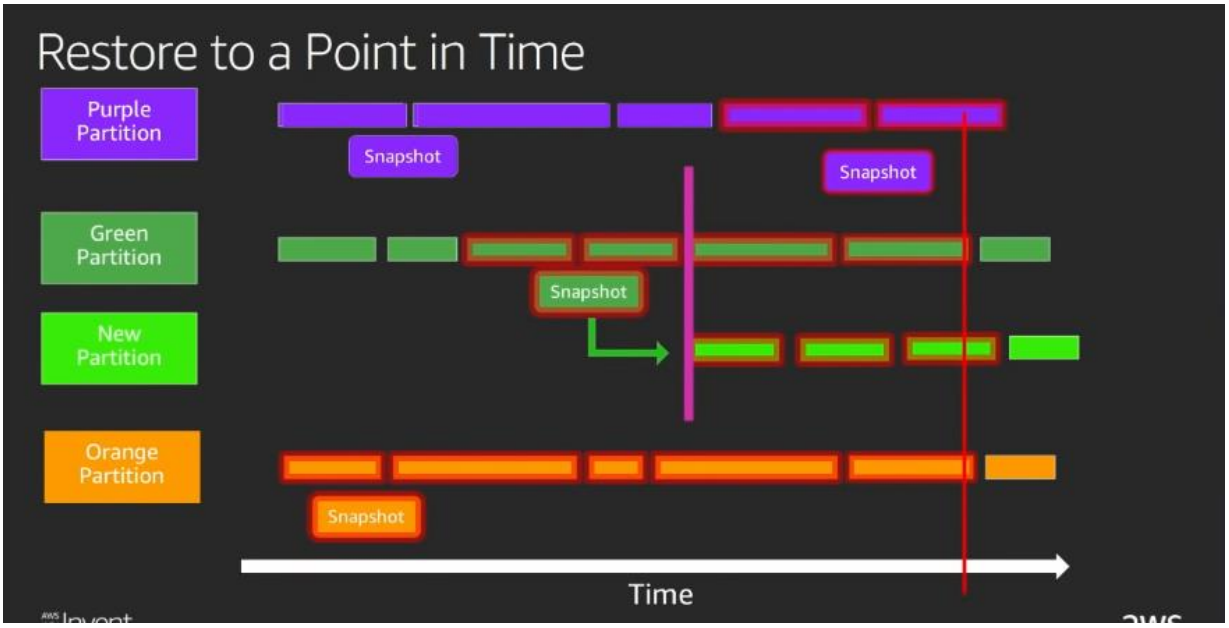


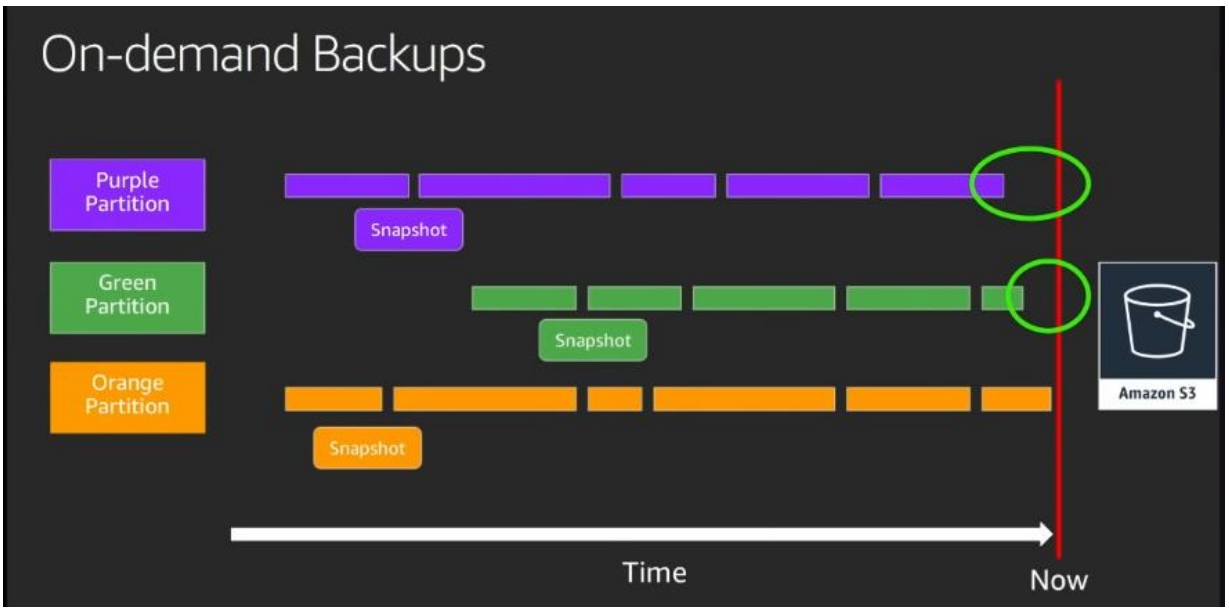
Restore to a Point in Time

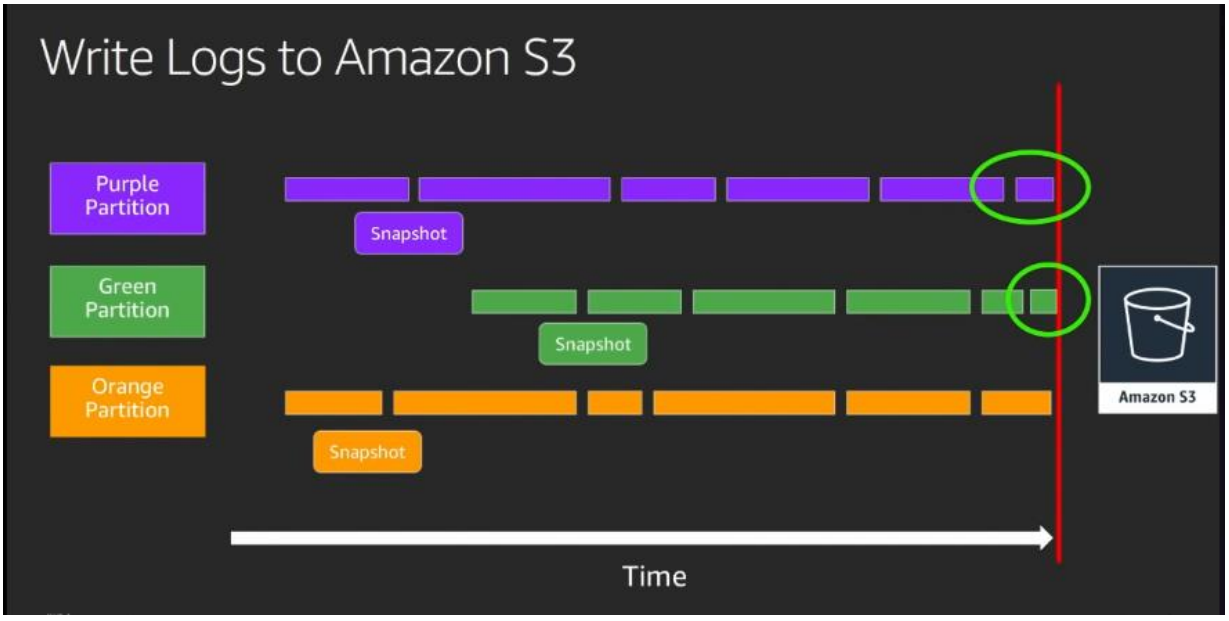


Partition Splits

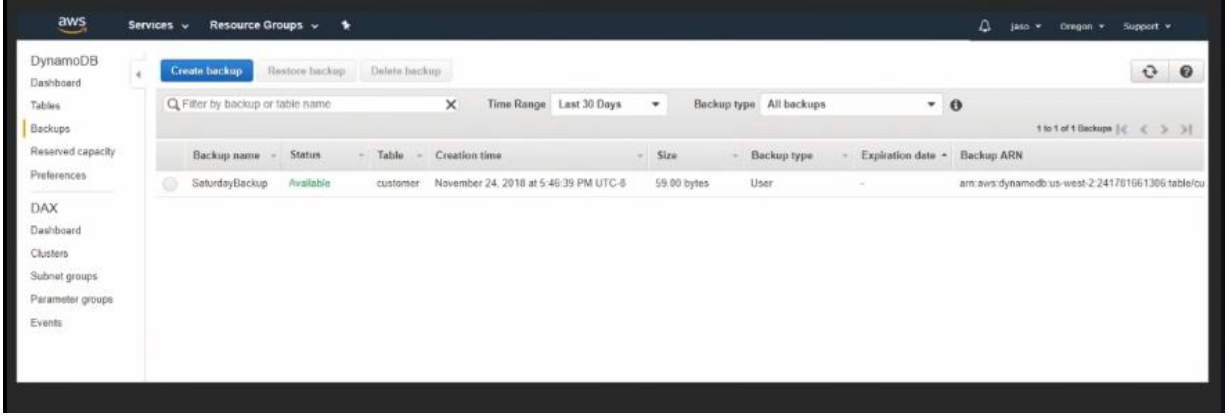



Restore to a Point in Time
Purple Partition
Green Partition
New Partition
Orange Partition
Snapshot
Snapshot
Snapshot
Snapshot
Time

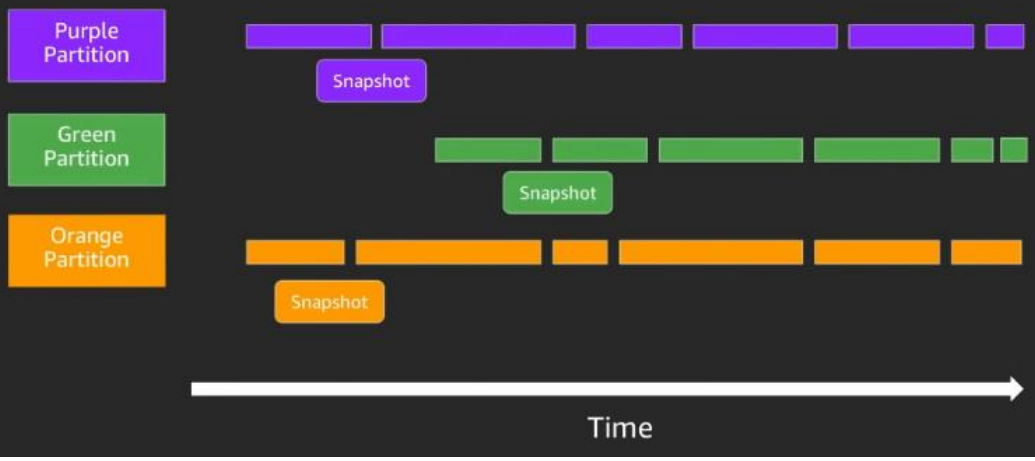

On-demand Backups
Purple Partition
Green Partition
Orange Partition
Snapshot
Snapshot
Snapshot
Amazon S3
Time
Now


Write Logs to Amazon S3
Purple Partition
Green Partition
Orange Partition
Snapshot
Snapshot
Snapshot
Amazon S3
Time

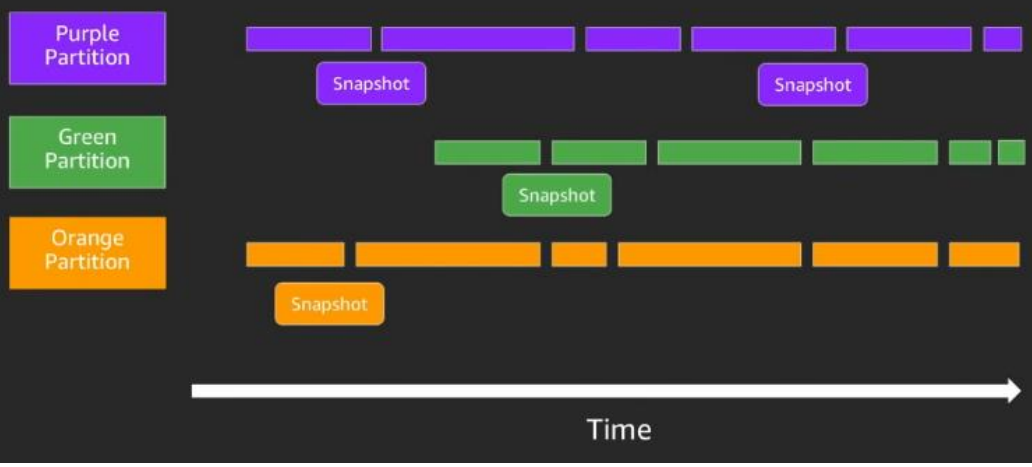
On-demand backup

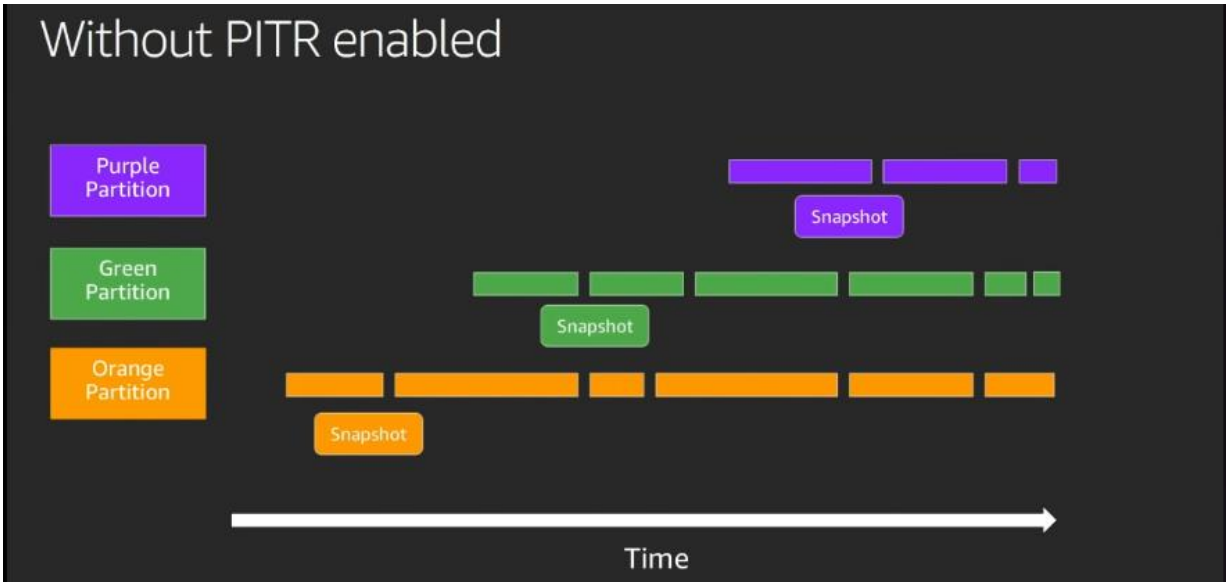


Without PITR enabled

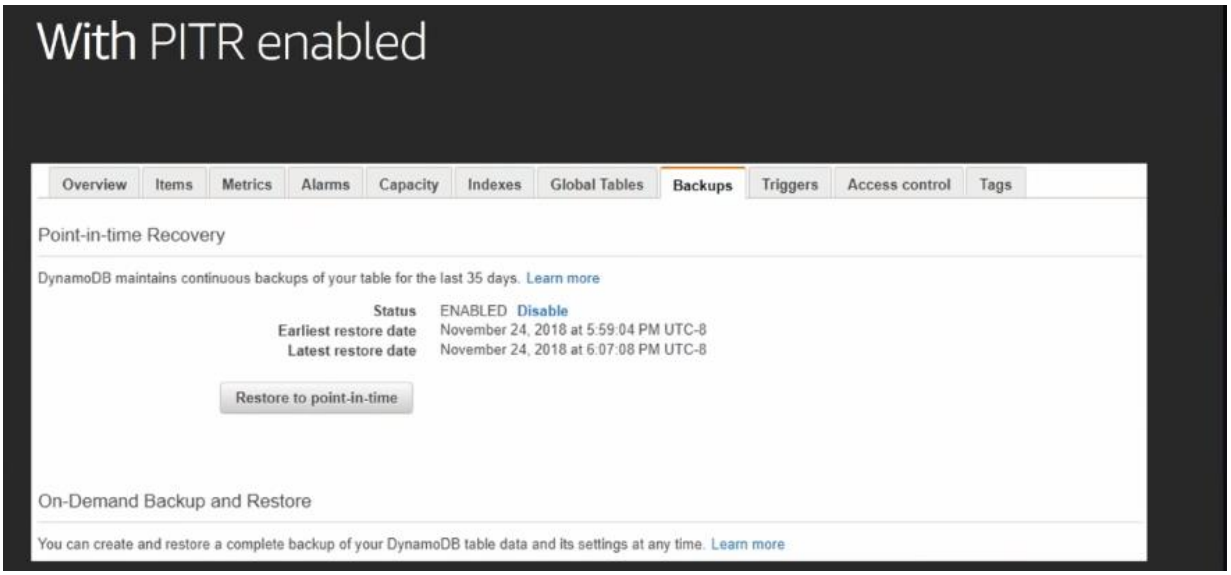


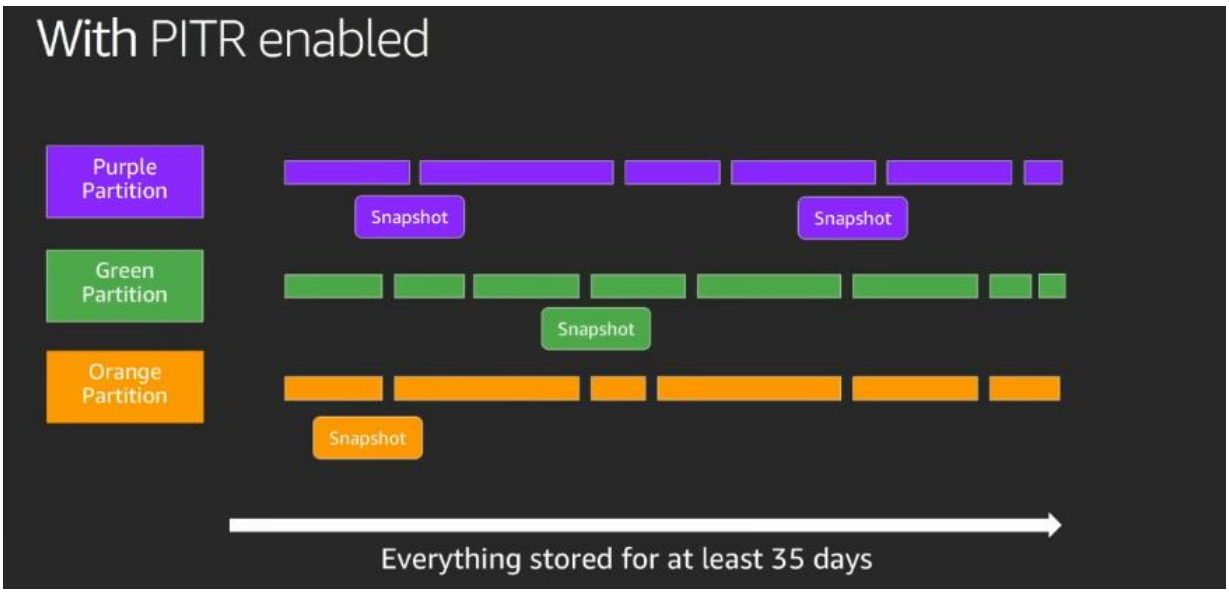
Without PITR enabled



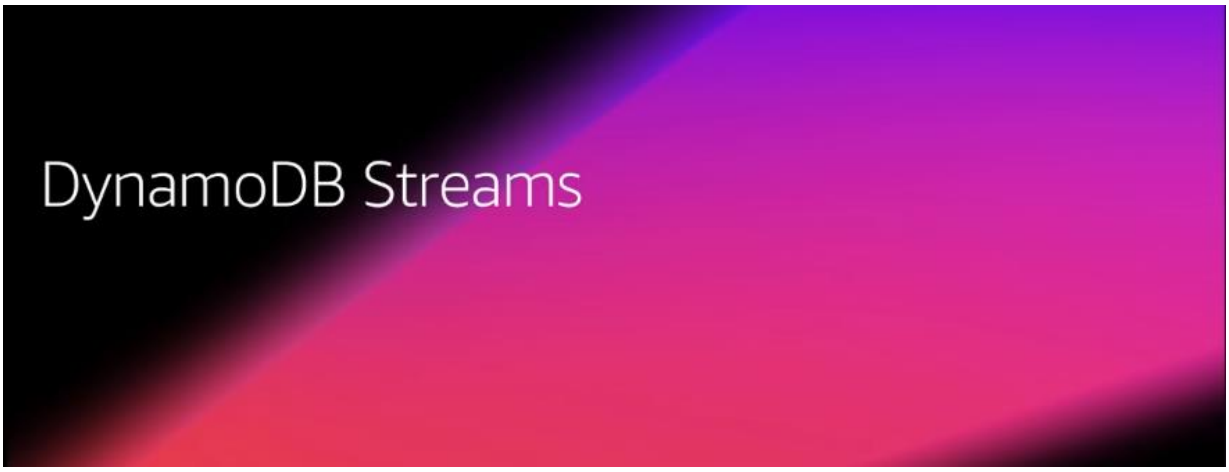


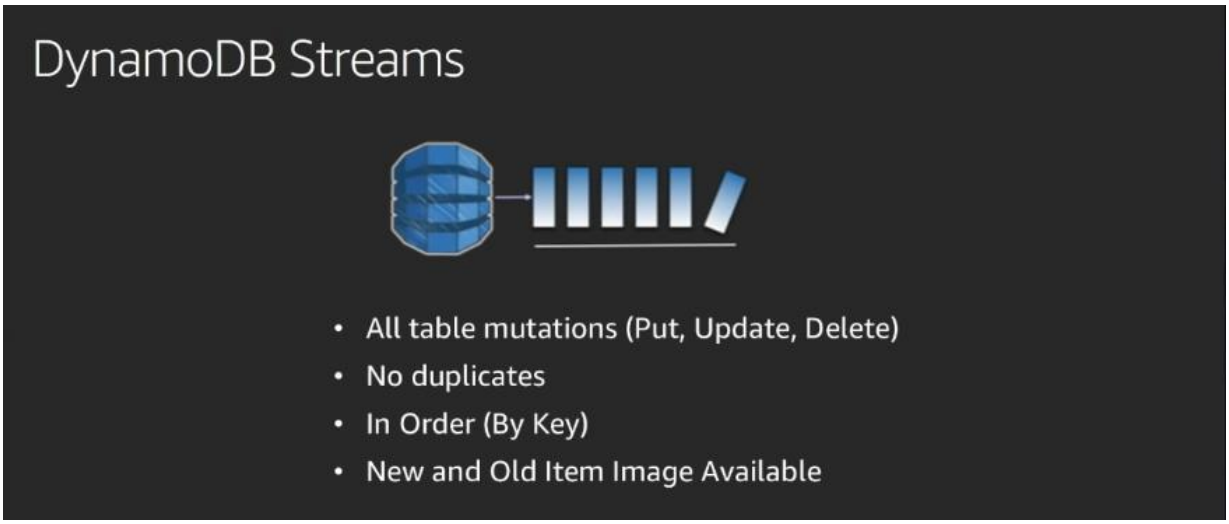
We can delete older backups



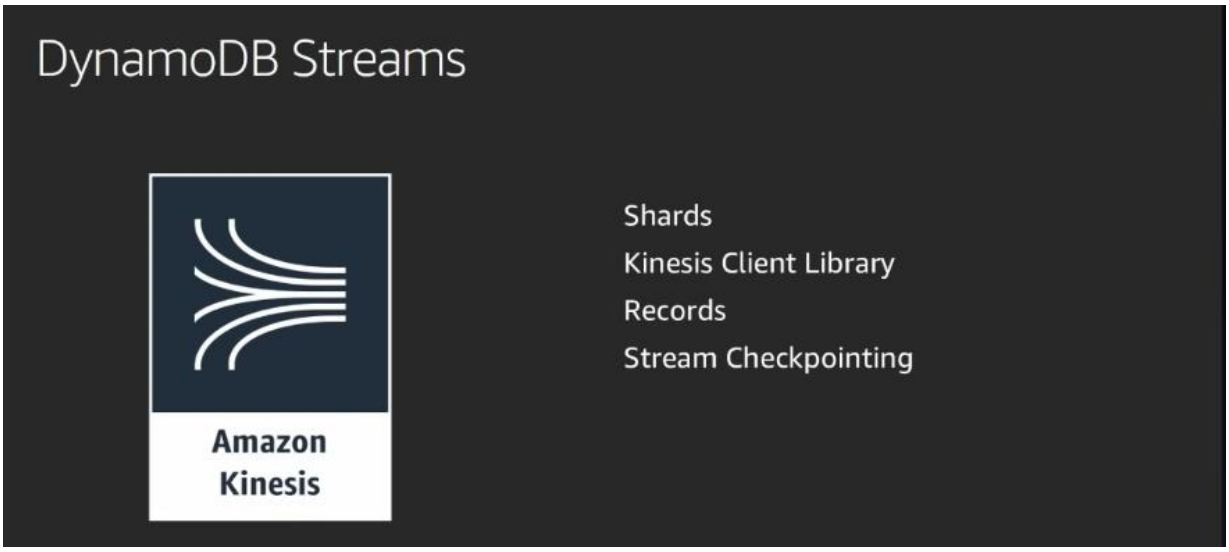


We now have to maintain this data for at least 35 days, that is why we charge you if you turn PITR on.

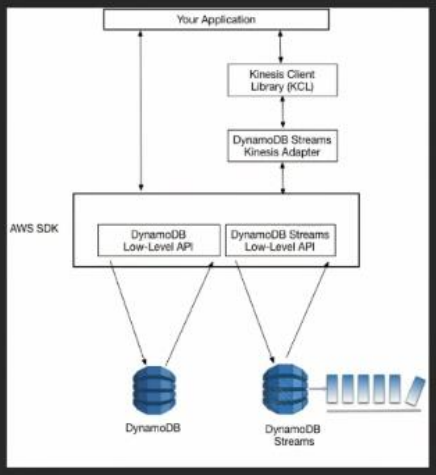




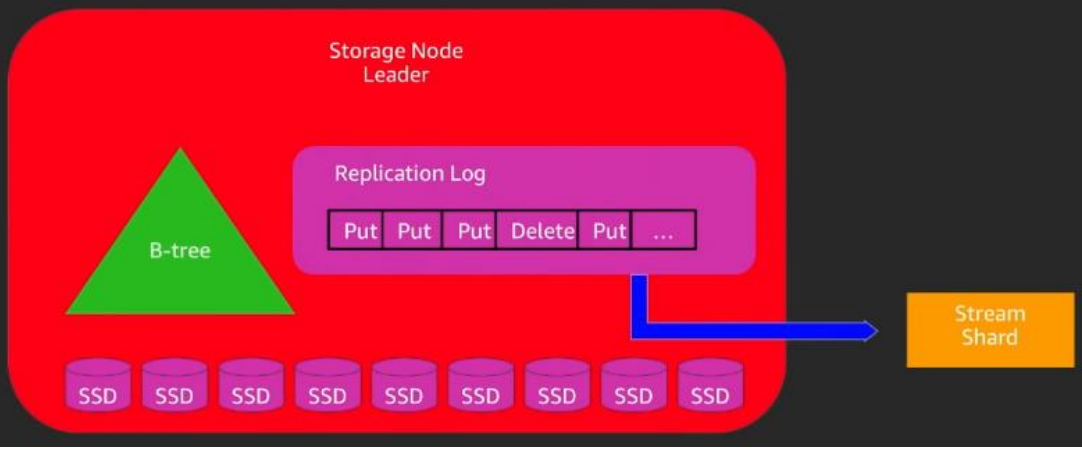
This is a way of getting every mutation to your table.



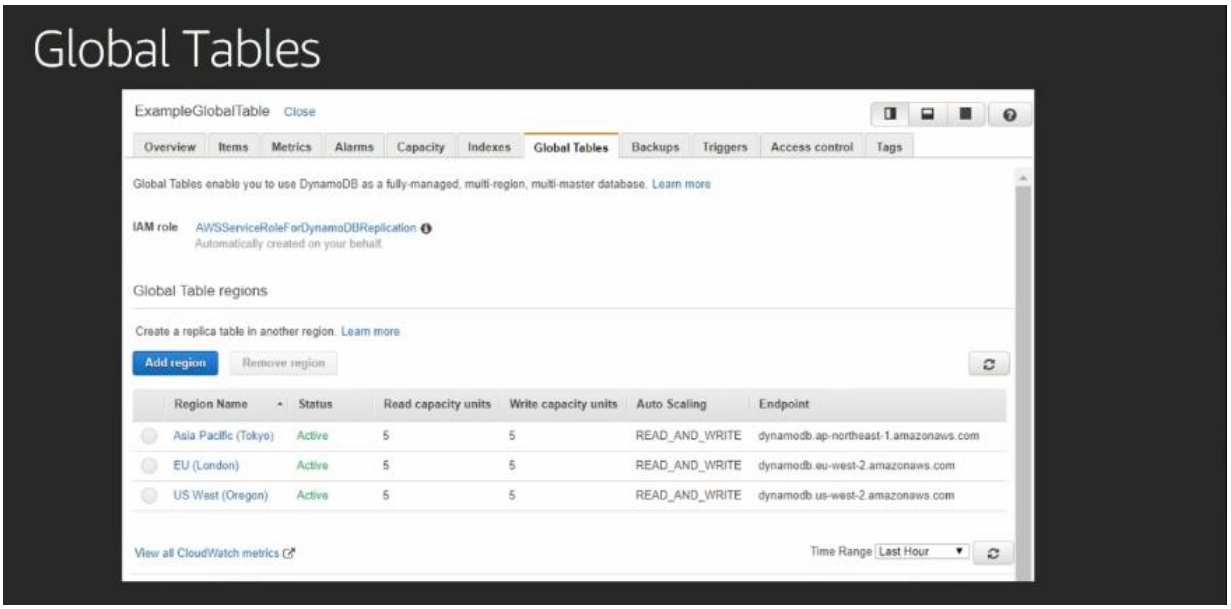
Reading from the stream



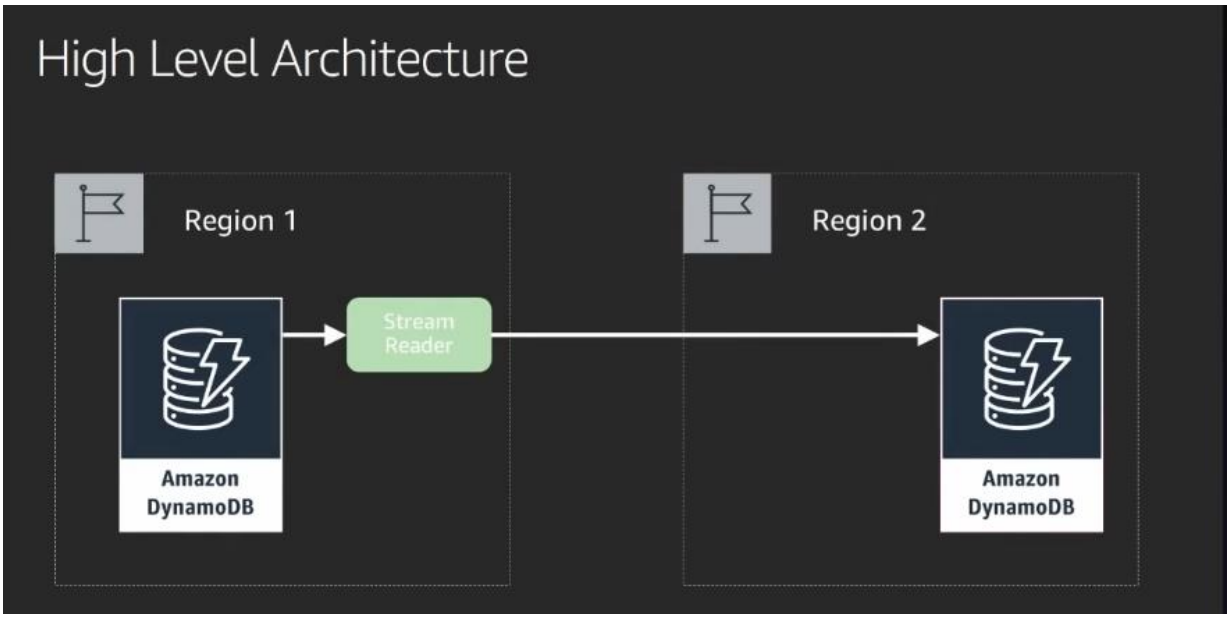
Writing to DynamoDB Streams

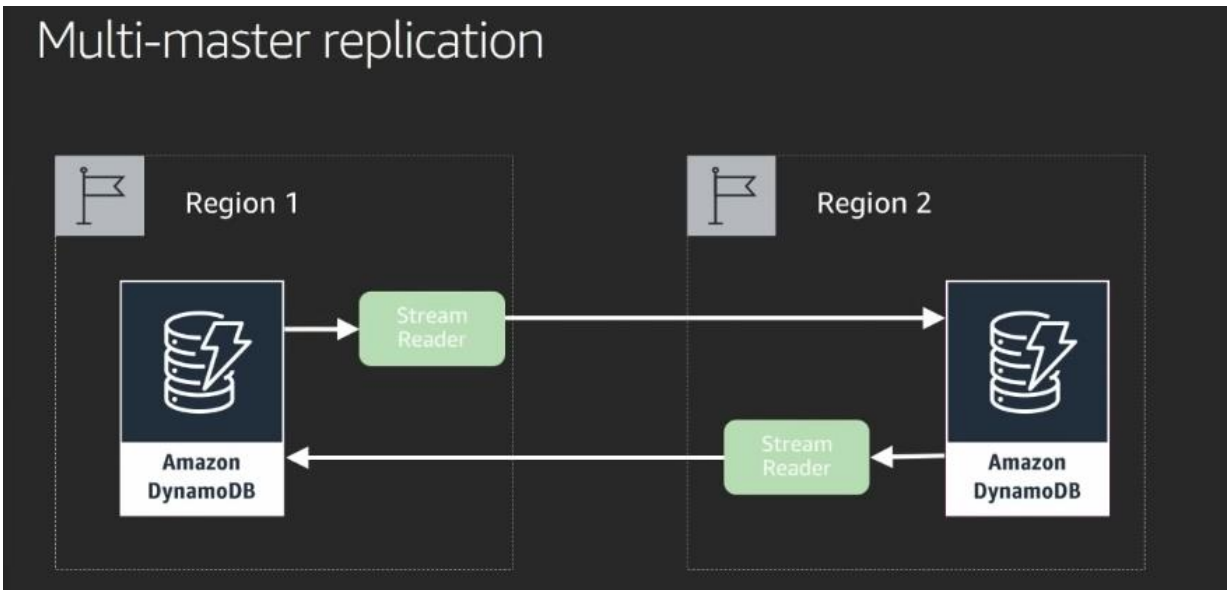


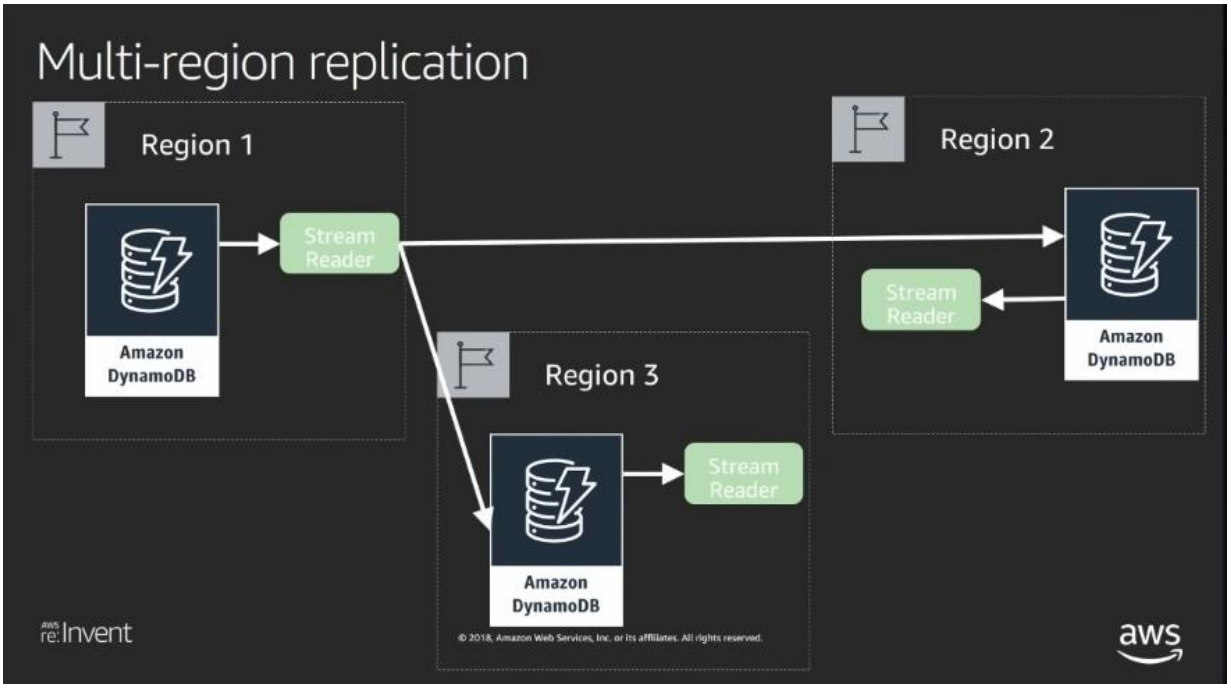
Global Tables

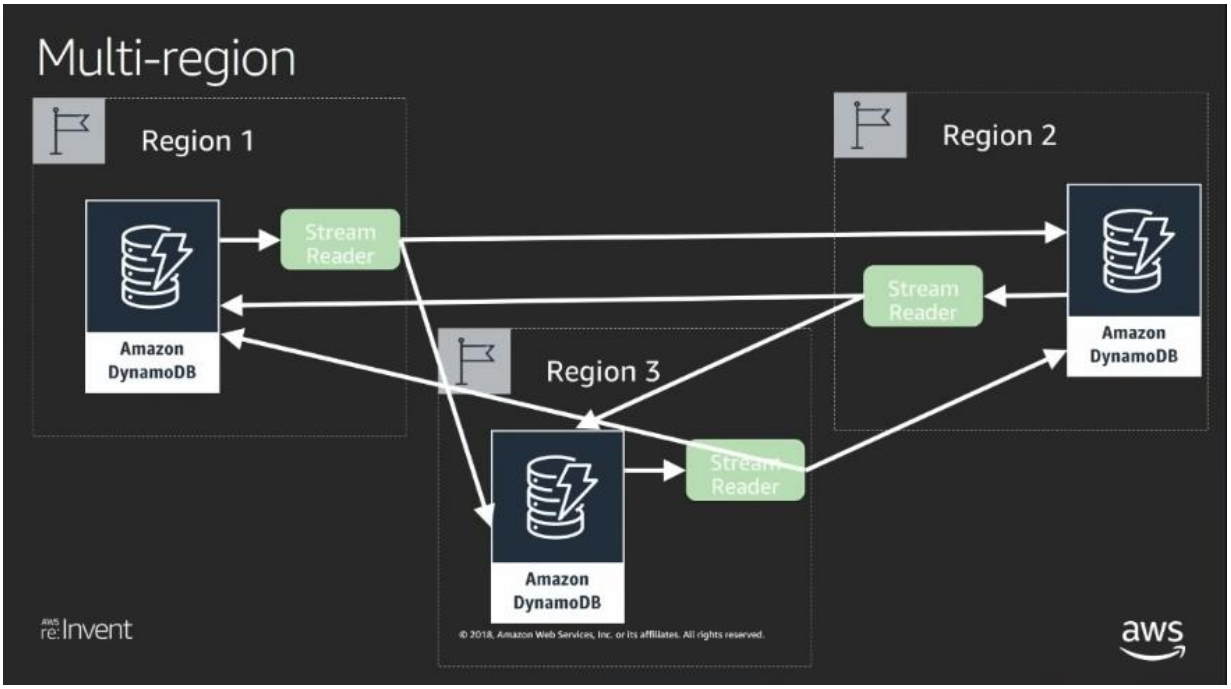


There is an IAM role associated with using Global Tables with DynamoDB

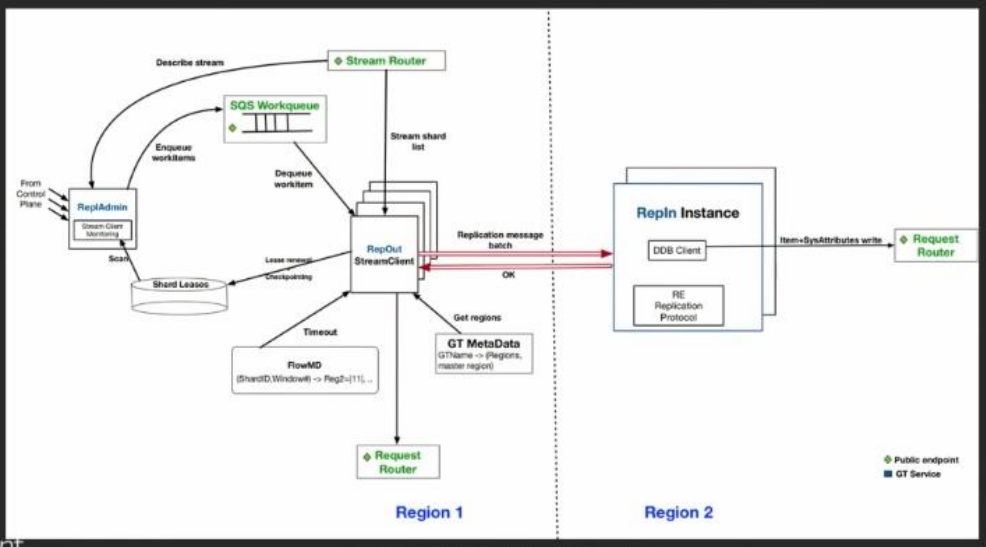




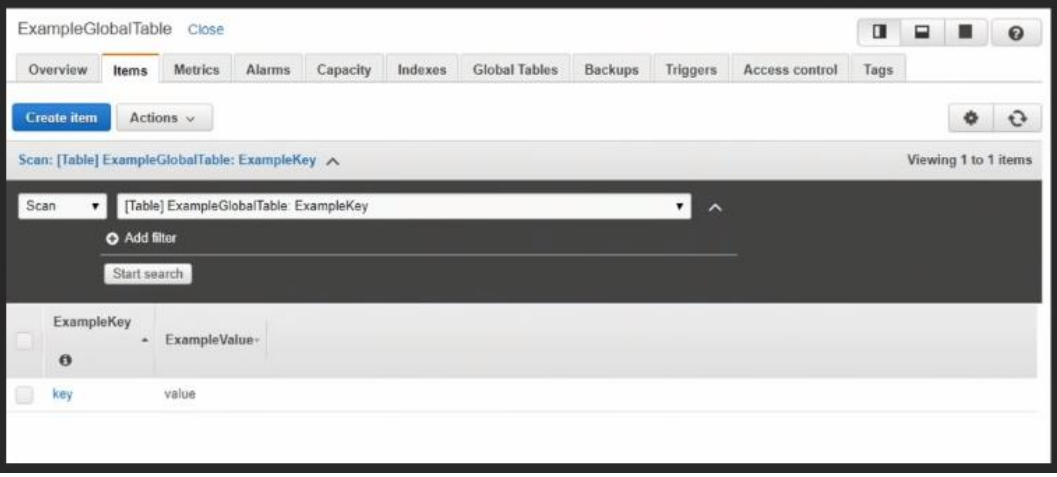

Multi-region replication
Region 1
Region 2
Region 3
Amazon DynamoDB
Stream Reader
Amazon DynamoDB
Stream Reader
Amazon DynamoDB
Stream Reader
aws re:Invent
© 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
aws


Multi-region
Region 1
Region 2
Region 3
Amazon DynamoDB
Stream Reader
Amazon DynamoDB
Stream Reader
Amazon DynamoDB
Stream Reader
aws re:Invent
© 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
aws

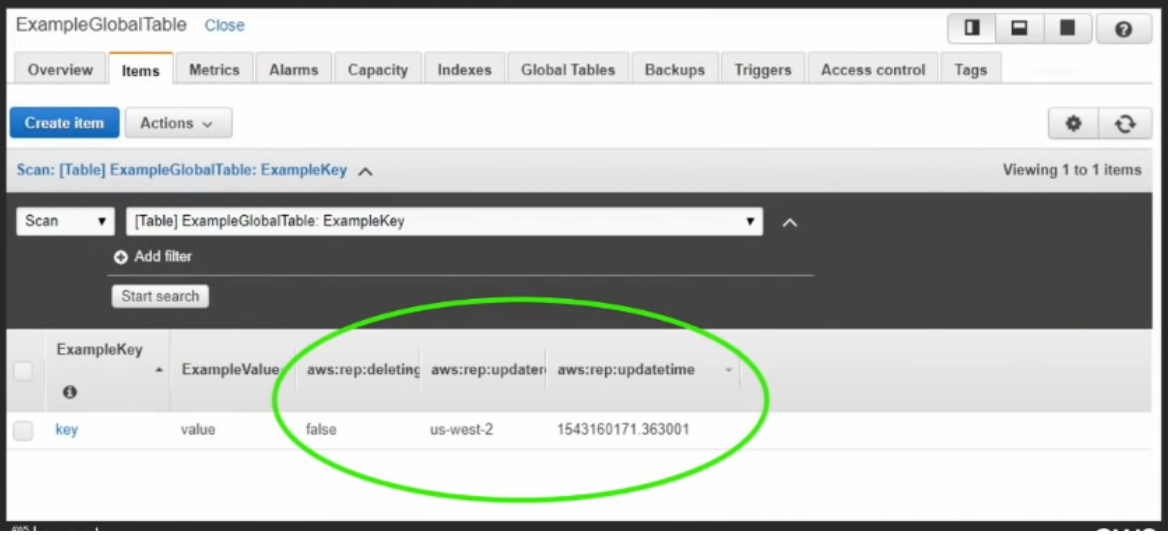
"Real" Architecture



Initial PutItem



After RepOut



Conflict Resolution

Last Writer wins.

- aws:rep:deleting
- aws:rep:updatetime
- aws:rep:updateregion

Areas Covered

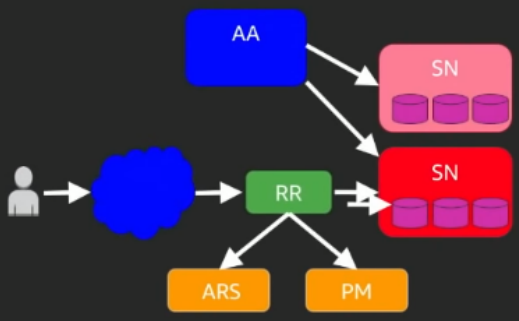
GetItem / PutItem

Auto Scaling

Backup Restore

Streams

Global Tables



Areas Not Covered

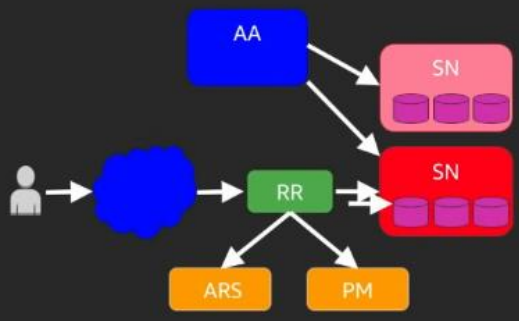
Fleet management

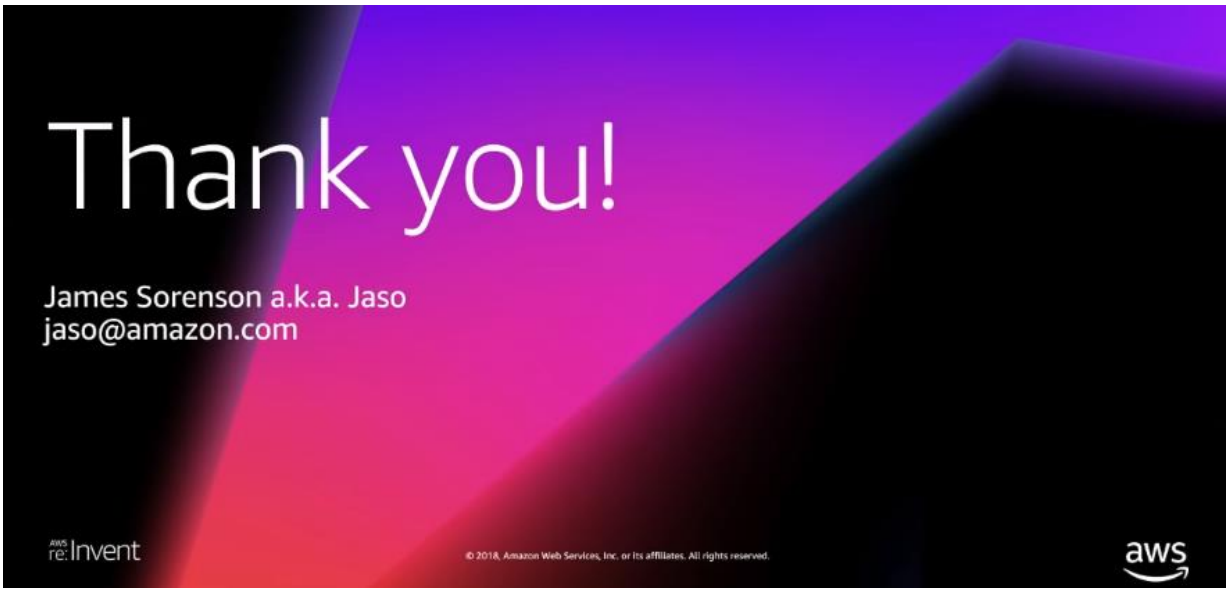
Metering

Monitoring and Alarming

Capacity Planning

...




Thank you!
James Sorenson a.k.a. Jaso
jaso@amazon.com
aws re:Invent
© 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.
aws