

AWS re:INVENT

Optimizing Serverless Application Data Tiers with Amazon DynamoDB

Srini Uppalapati – Vice President,
Consumer Bank Engineering, Capital One
Edin Zulich – NoSQL Solutions Architect, AWS

November 27, 2017

AWS
re:Invent

© 2017 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



As a fully managed database service, **Amazon DynamoDB** is a natural fit for **serverless architectures**. In this session, we dive deep into why and how to use DynamoDB in serverless applications, followed by a real-world use case from CapitalOne.

First, we dive into the relevant DynamoDB features, and how you can use it effectively with AWS Lambda in solutions ranging from web applications to real-time data processing. We show how some of the new features in DynamoDB, such as Auto Scaling and Time to Live (TTL), are particularly useful in serverless architectures, and distill the best practices to help you create effective serverless applications. In the second part, we talk about how CapitalOne migrated billions of transactions to a completely serverless architecture and built a scalable, resilient and fast transaction platform by leveraging DynamoDB, AWS Lambda and other services within the serverless ecosystem.

Agenda

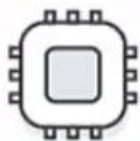
- DynamoDB in serverless applications
 - A bit about Amazon DynamoDB
 - Example: transactions and queries with microservices
 - Example: high volume time series data
- Serverless at CapitalOne
 - Breaking down the monolith
 - Transaction Hub journey

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



AWS Serverless Platform



AWS Lambda



Amazon API Gateway



Amazon S3



Amazon DynamoDB



Amazon SNS & SQS



AWS Step Functions



Amazon Kinesis
Amazon Athena



Tooling

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved. <https://aws.amazon.com/serverless/>



AWS Serverless Platform



What is DynamoDB?

AWS Blog

Prime Day 2017 – Powered by AWS

by [Jeff Barr](#) | on 14 SEP 2017 | in [Case Studies](#) | [Permalink](#) | [Share](#)

NoSQL Database – Amazon DynamoDB requests from Alexa, the Amazon.com sites, and the Amazon fulfillment centers totaled 3.34 trillion, peaking at 12.9 million per second. According to the team, the extreme scale, consistent performance, and high availability of DynamoDB let them meet needs of Prime Day without breaking a sweat.

What is DynamoDB?

- Fully managed NoSQL database service
- Designed for OLTP use cases where you know access patterns
- Designed to be used as an operational database that provides:
 - Extreme scale – now with Auto Scaling
 - Performance – single digit ms latency at any scale
 - High availability and reliability – zero downtime
 - Global availability – available in all AWS regions
 - Serverless experience – no servers and clusters to manage
 - Integration with AWS Lambda and other AWS services



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



DynamoDB: Common Use Cases

Market orders

Tokenization
(PHI, credit cards)

User profiles

Sessions

Shopping cart

Chat messages

IoT sensor data
& device status

Social media feeds

File metadata



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Reference Architecture (Processing Paths)

Building blocks for serverless microservices



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



This pattern is commonly used in mobile backends, web apps, and e-commerce applications.

Reference Architecture (Processing Paths)

Building blocks for serverless microservices



AWS re:Invent

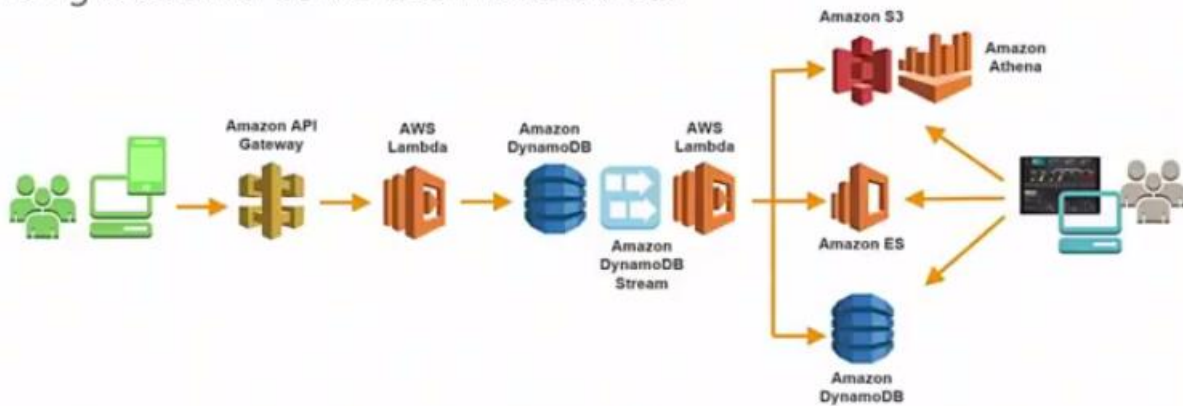
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



You can stream the data to consumers that can then do some processing

Reference Architecture (Processing Paths)

Building blocks for serverless microservices



AWS
re:Invent

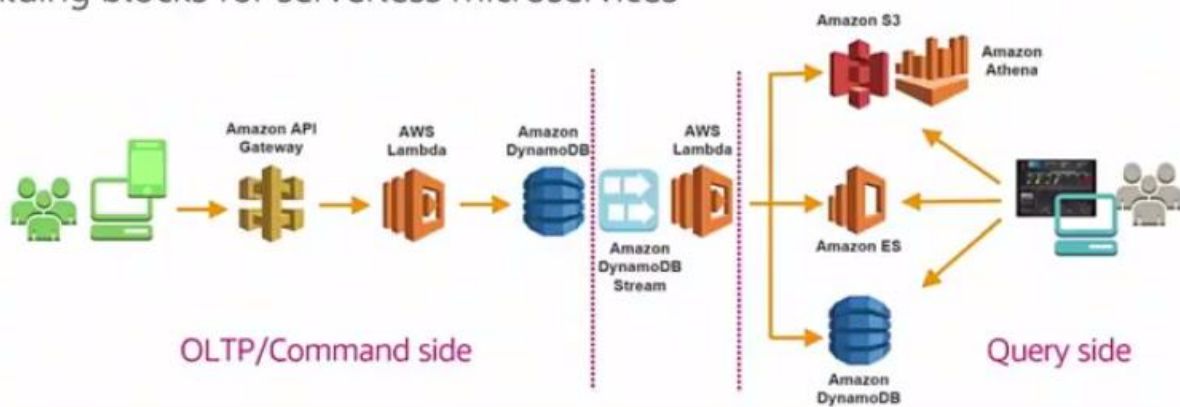
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



You can also send data to different locations using lambdas

Reference Architecture (Processing Paths)

Building blocks for serverless microservices



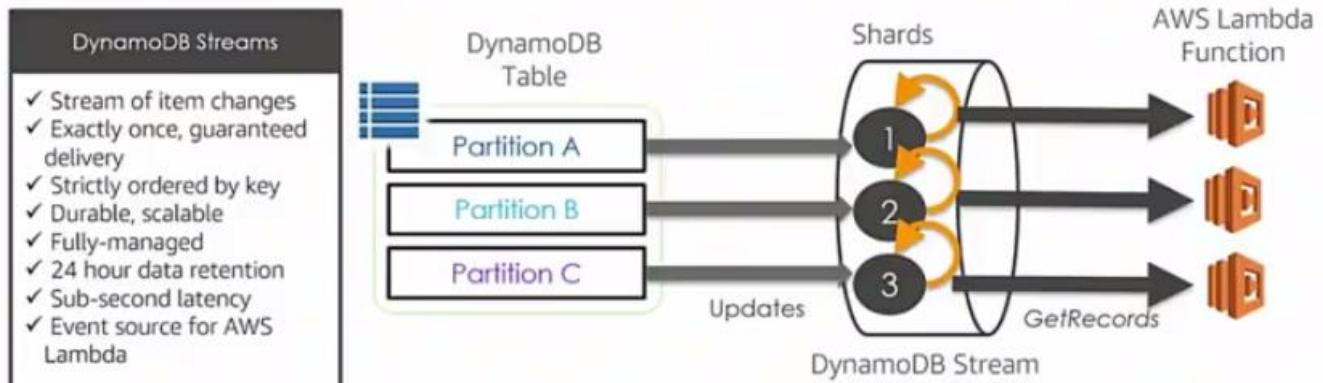
1. API Gateway + Lambda + DynamoDB: core building block
2. DynamoDB Streams + AWS Lambda: building block for reliable event delivery

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



DynamoDB Streams and AWS Lambda



DynamoDB Streams + AWS Lambda = reliable "at least once" event delivery

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

But It's NoSQL! I need SQL...

“

A deep dive on how we were using our existing databases revealed that they were frequently not used for their relational capabilities. About 70 percent of operations were of the key-value kind, where only a primary key was used and a single row would be returned.

Werner Vogels
CTO, Amazon,
On database usage patterns at Amazon

”

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Example: Transactions and Queries with DynamoDB and Microservices

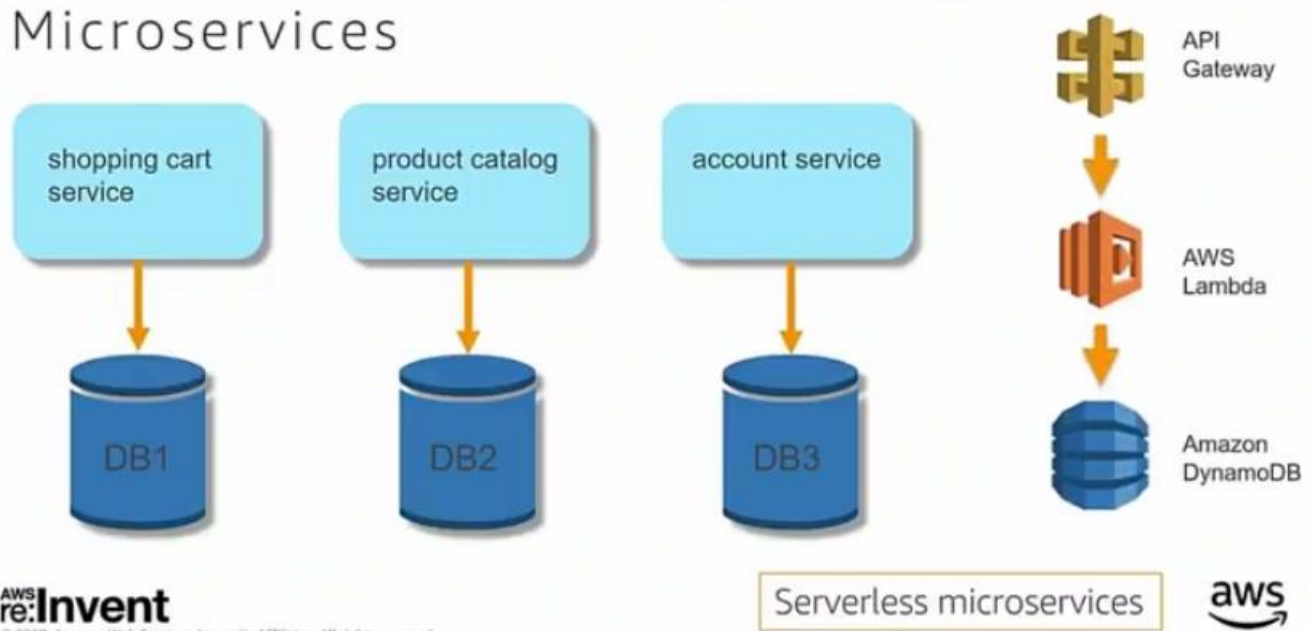
- An e-commerce app
- Relational versus non-relational data model
- Transactions in a distributed microservices architecture
- Enabling querying in a distributed microservices architecture

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Example: E-commerce App Using Microservices



AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Serverless microservices



Shopping Cart Data Model

Relational database



Normalized schema

Non-relational database



De-normalized: aggregates

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



NoSQL database like DynamoDB use a schema-less approach.

Updating Cart Using OCC

1. Get the cart: GetItem

```
{  "TableName": "Cart",  
  "Key": {"CartId": {"N": "2"}}  
}
```



2. Update the cart: conditional PutItem (or UpdateItem)

```
{  "TableName": "Cart",  
  "Item": {  
    "CartID": {"N": "2"},  
    "LastUpdate": {"N": "t4"},  
    "CartItems": {...}  
  },  
  "ConditionExpression": "LastUpdate = :v1",  
  "ExpressionAttributeValues": {":v1": {"N": "t3"}}  
}
```

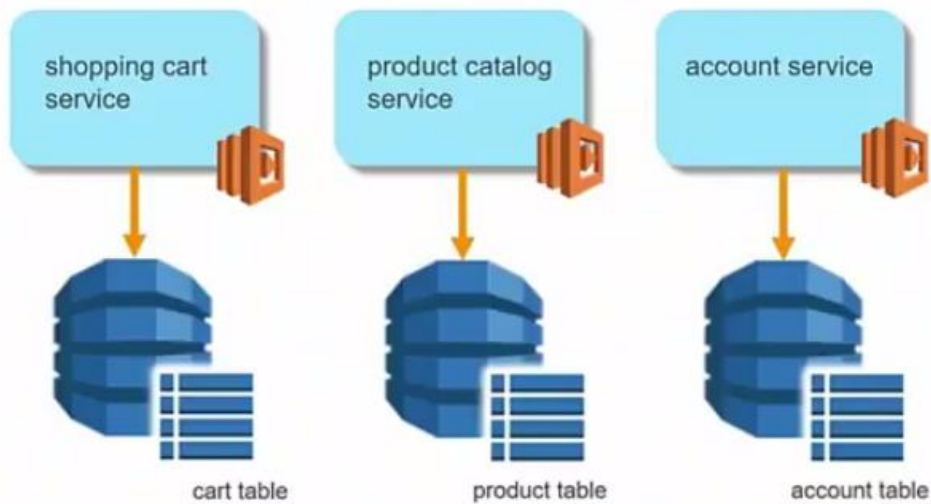
- ✓ Use conditions to implement optimistic concurrency control ensuring data consistency
- ✓ GetItem call can be eventually consistent

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transactions that Span Microservices

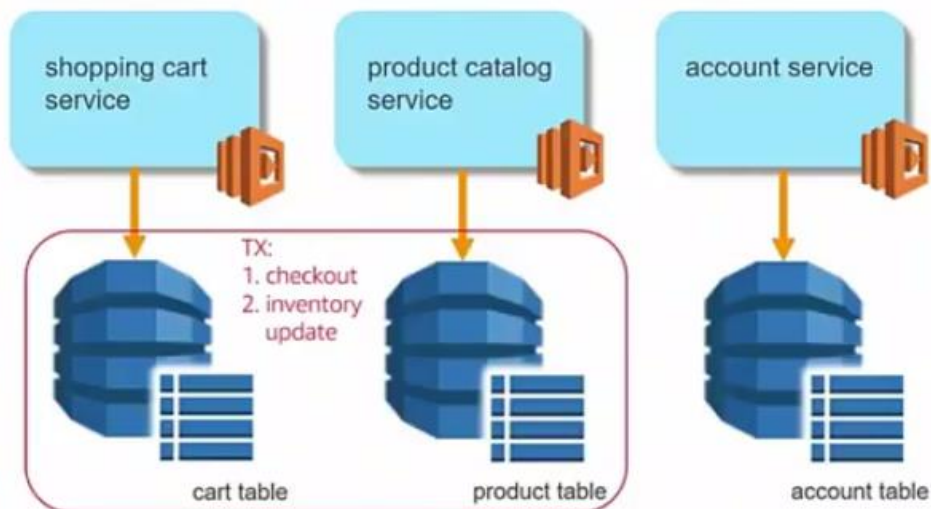


AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transactions that Span Microservices



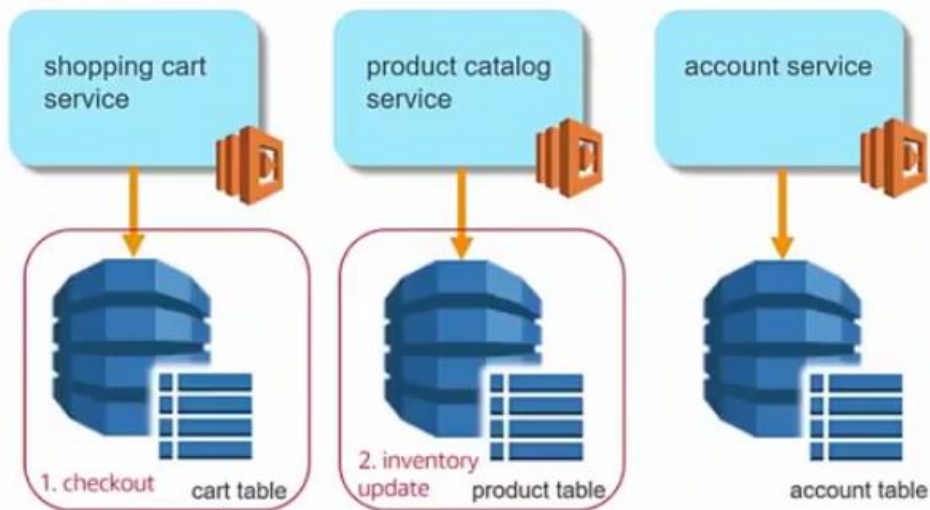
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We can also do this as a 2-phase commit without locking as below

Transactions that Span Microservices

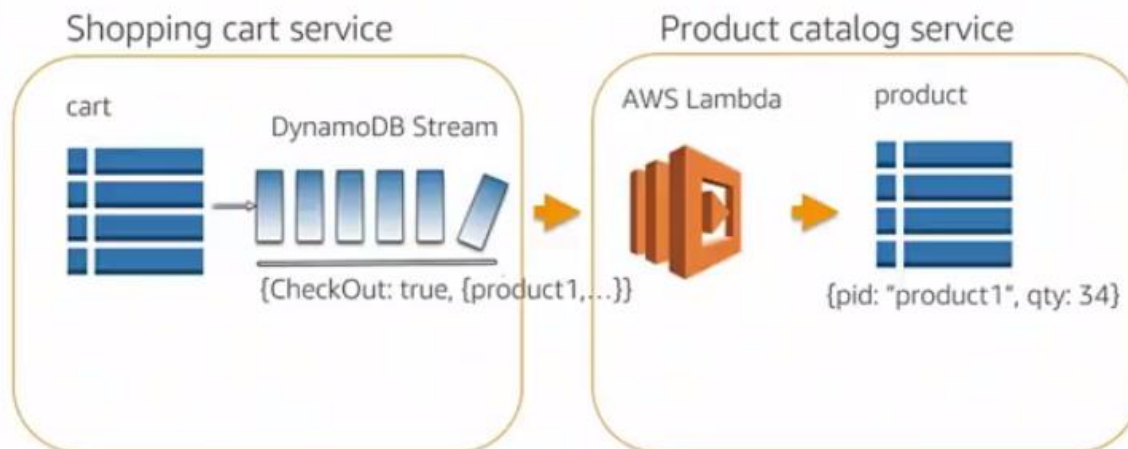


AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transaction: Cart Checkout

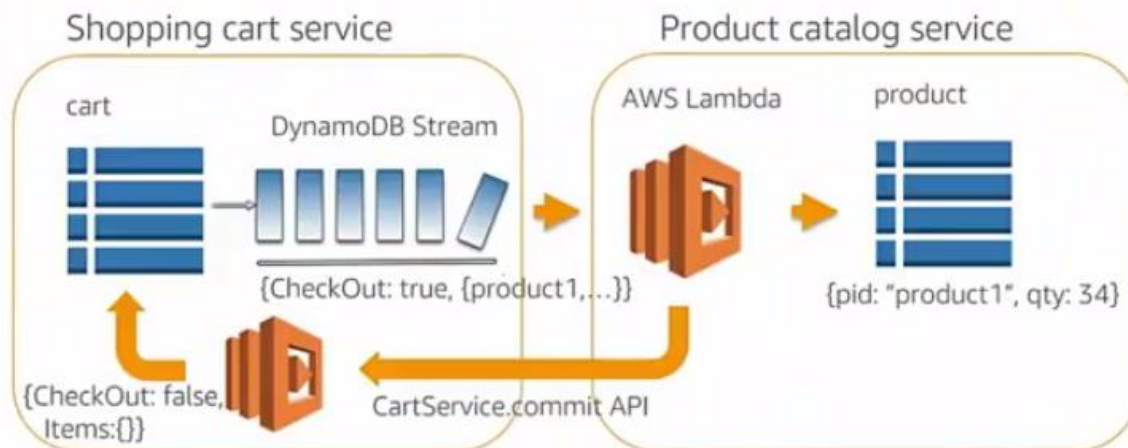


AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transaction: Cart Checkout



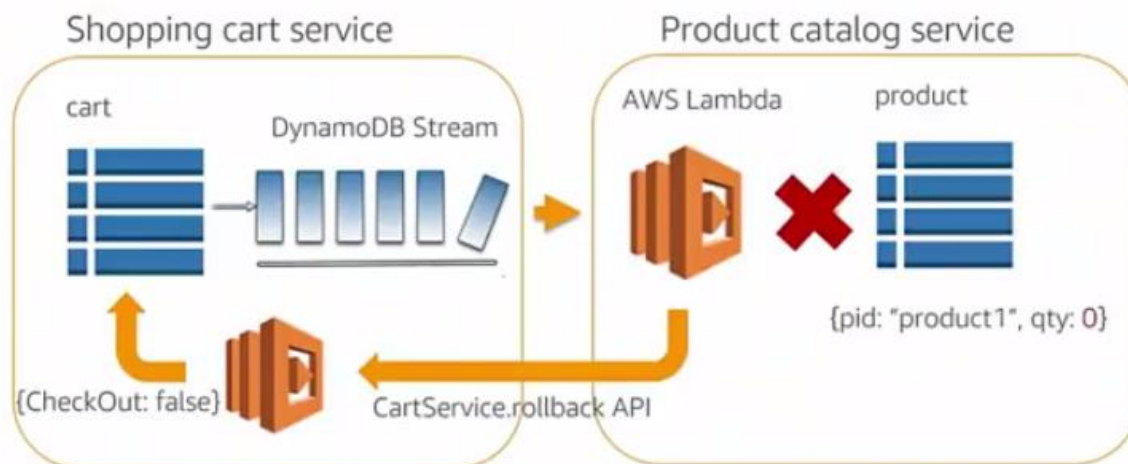
Pattern: Saga using DynamoDB Streams and Lambda

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transaction: Cart Checkout Failure



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

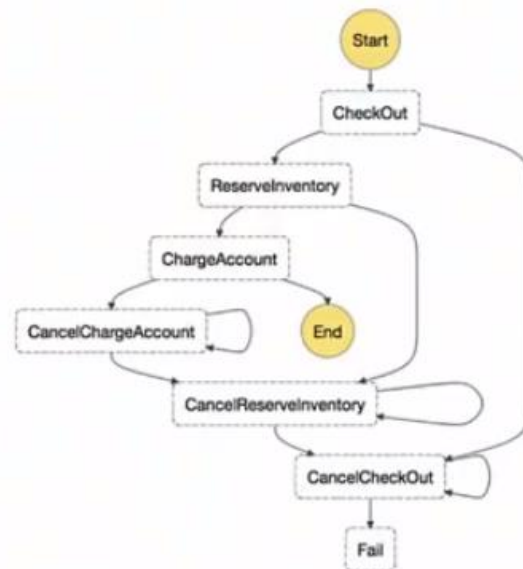


Cart Table Update Scenarios Using OCC

1. Add/remove item:
 1. `get cart => tread = LastUpdate`
 2. `update cart`
 `add/remove IF (LastUpdate = tread AND Checkout = false)`
2. Checkout begin:
 1. `get cart => tread = LastUpdate`
 2. `update cart`
 `SET Checkout = true IF (LastUpdate = tread AND Checkout = false)`
3. Checkout commit:
 1. `get cart => tread = LastUpdate`
 2. `update cart`
 `SET items={}, Checkout = false IF (LastUpdate = tread AND Checkout = true)`
4. Checkout rollback:
 1. `get cart => tread = LastUpdate`
 2. `update cart`
 `SET Checkout = false IF (LastUpdate = tread AND Checkout = true)`

Another Building Block for Transactions: Step Functions

- Define steps as a state machine
- Each step is implemented as a Lambda function
- Lambda functions have to be idempotent
- This approach can be combined with the DynamoDB Streams based approach



Transactions with Microservices: Takeaways

- Requirements:
 1. Service atomically updates database and publishes events
 2. Events delivered at least once
- DynamoDB Streams + AWS Lambda approach satisfies both
- Denormalized schema (aggregates)
- Building blocks: DynamoDB Streams + AWS Lambda; Step Functions
- Asynchronous and eventually consistent
- Each service publishes its rollback method
- Lambda functions have to be idempotent
- Increased total transaction latency



© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Querying in Microservices Architectures

- Challenges
 - Data is segregated in different microservices' databases
 - Operational view of data does not satisfy querying needs
- Solution
 - Separate the operational and querying views
 - Polyglot persistence: use the right database for the job
 - Command Query Responsibility Segregation (CQRS)

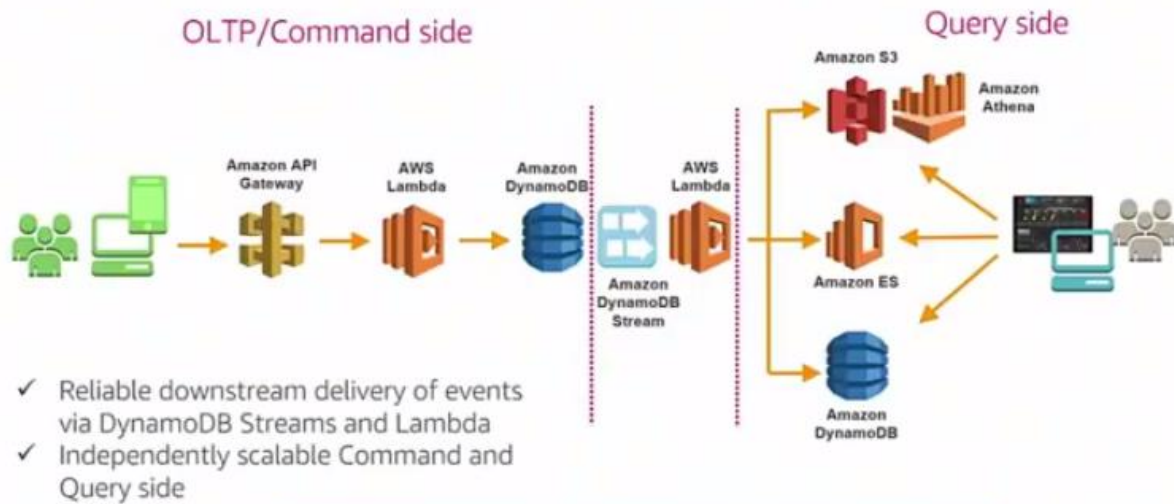


© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We use the CQRS pattern for querying in microservices architecture by separating the 2 parts

Enabling Querying with Microservices



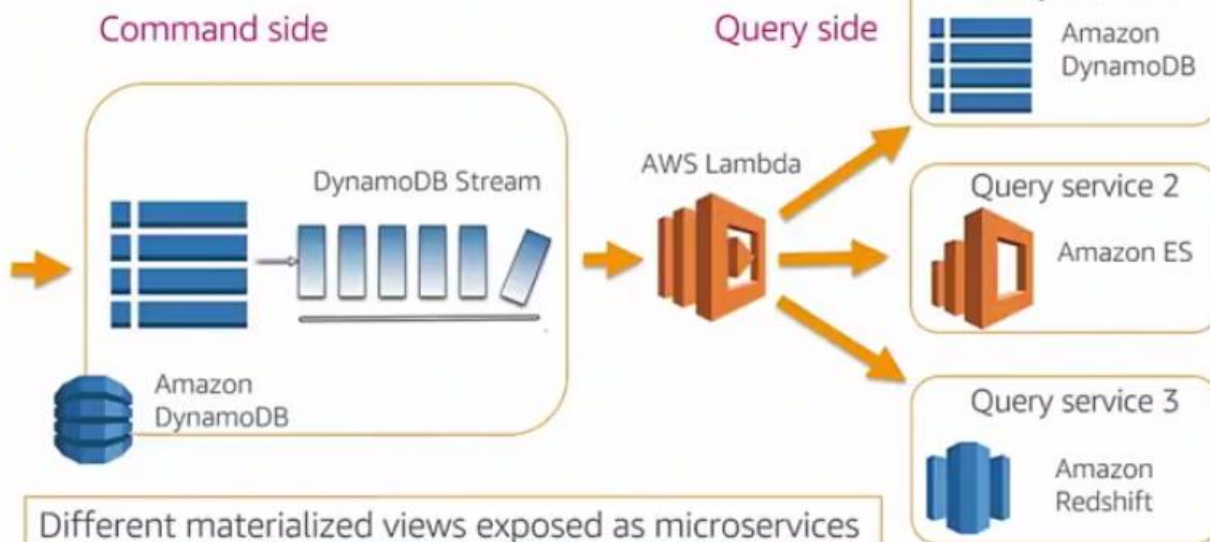
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Command Query Responsibility Segregation



Querying with Microservices



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Multiple Subscribers: Fan-Out



Querying with Microservices: Takeaways

- CQRS provides separation of concerns
 - CRUD operations and queries separated from each other
 - Each optimized for its responsibility
- Each side is independently scalable
- Each view can be exposed as a microservice
- Eventual consistency and longer latency
- Added complexity

Example: High Volume Time Series Data

What we need:

1. High volume ingest
 - 100,000 data points per second
2. Fast access for X days/months
 - By sensor ID + time range
 - Under 10 ms

AWS re:Invent

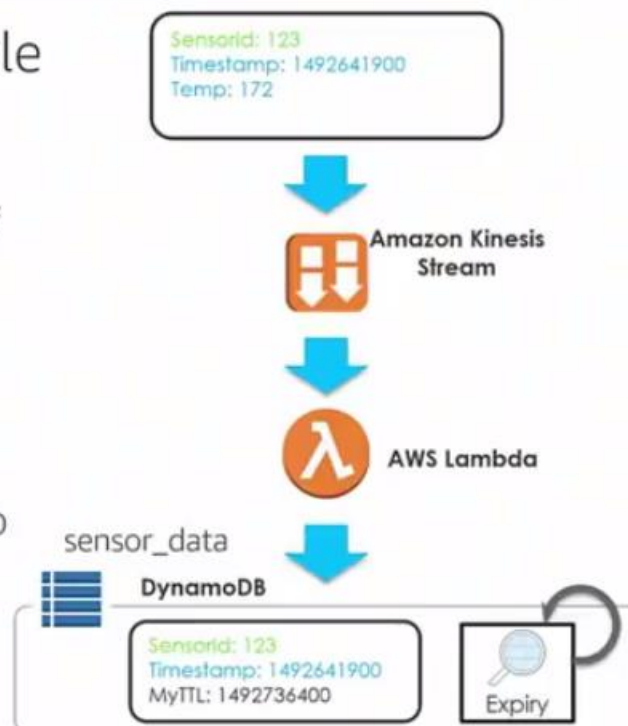
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Time Series Data Example

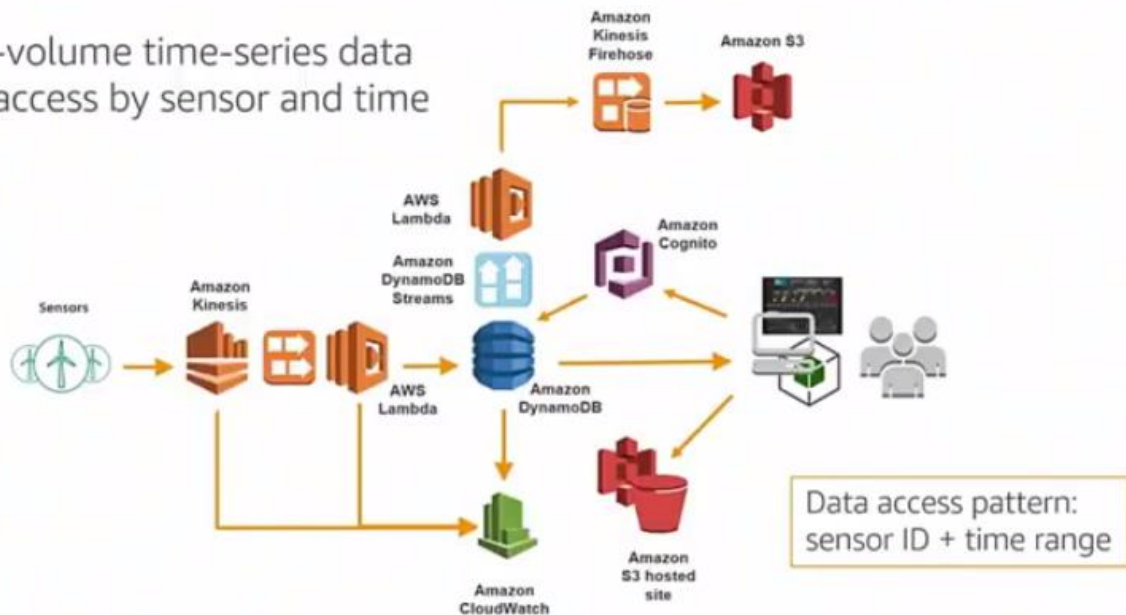
- Sensor data (temp) from 1000's of sensors
- Store for fast access (<10 ms)
- Access by sensor ID + time range
- Store for 30 days

Kinesis, DynamoDB partition key: Sensor ID
DynamoDB sort key: Timestamp



Solution Architecture

High-volume time-series data
Fast access by sensor and time



We use CloudWatch to monitor our services and possibly implement automatic scaling for our Kinesis Streams, Lambda and DynamoDB can scale automatically. DynamoDB expires data after 2 days but can store them in some location like S3 before deleting them. We use AWS Cognito to authenticate the user before granting access.

Cost Considerations

Data rate: 100,000 data points per second

Data storage: 1 month's worth = ~2.5 TB

Estimated cost:

Lambda: \$2K – \$5K per month

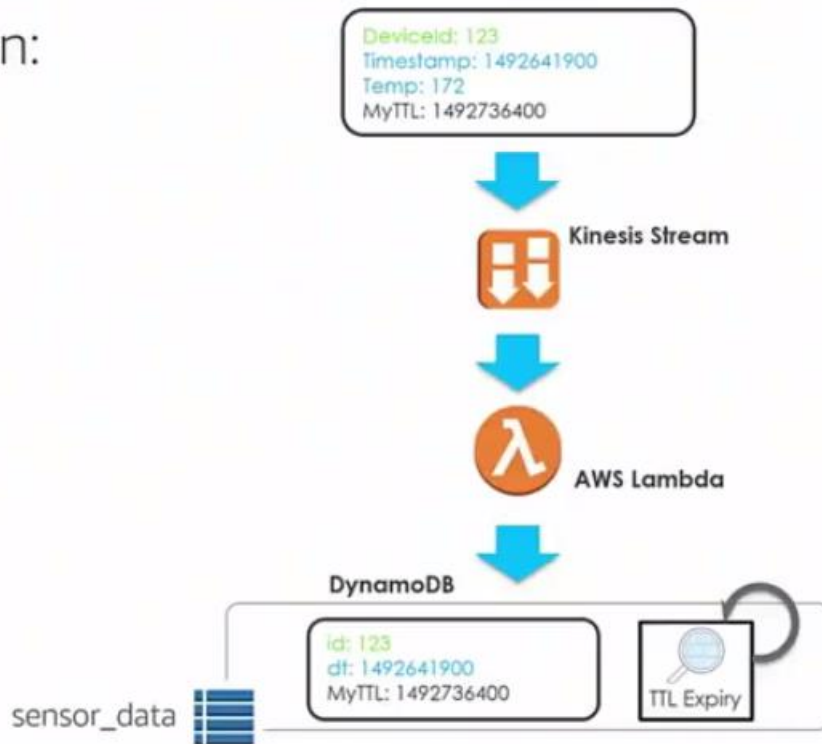
Kinesis: 100,000 records -> 100 shards -> ~\$5K per month

DynamoDB: 100,000 WCU's -> \$50K per month

Where is the problem?

- DynamoDB Write Capacity Unit (WCU) is 1 KB
- Our scenario:
 - Storing data points ~50 B in size
 - Write capacity utilization per write: $50/1024 * 100\% = 4.88\%$

Solution:

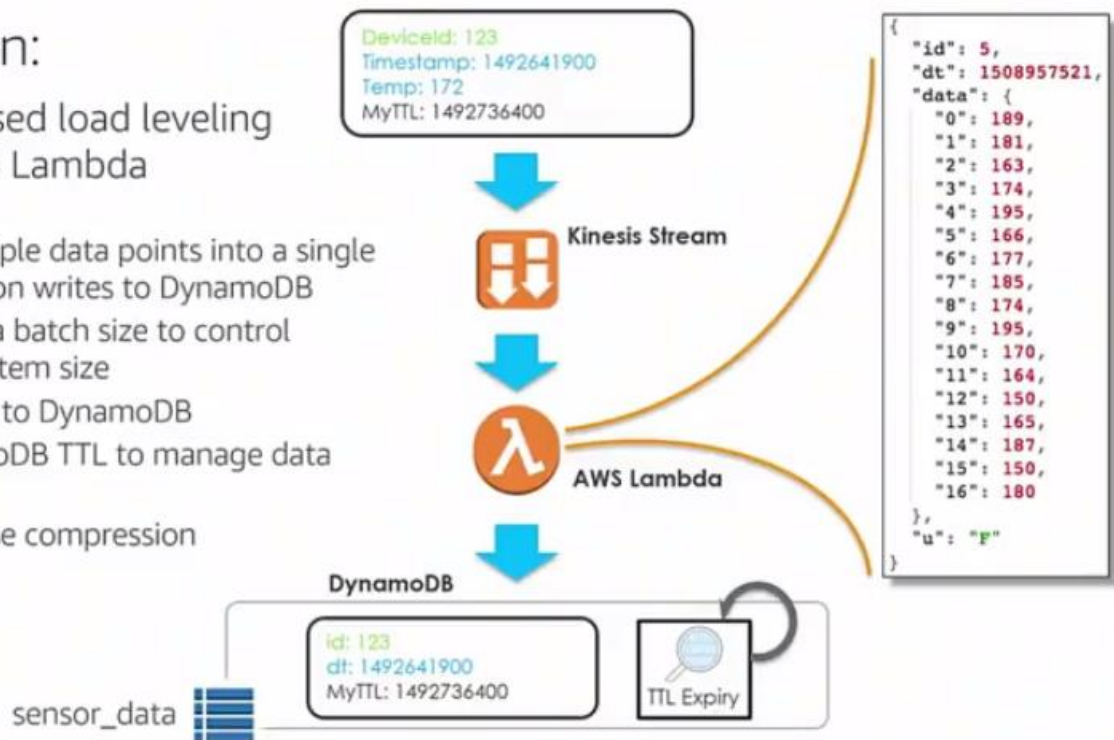


In our architecture, batches of data are delivered from kinesis to our Lambda functions. When we invoke that function every time a single stream of data comes in

Solution:

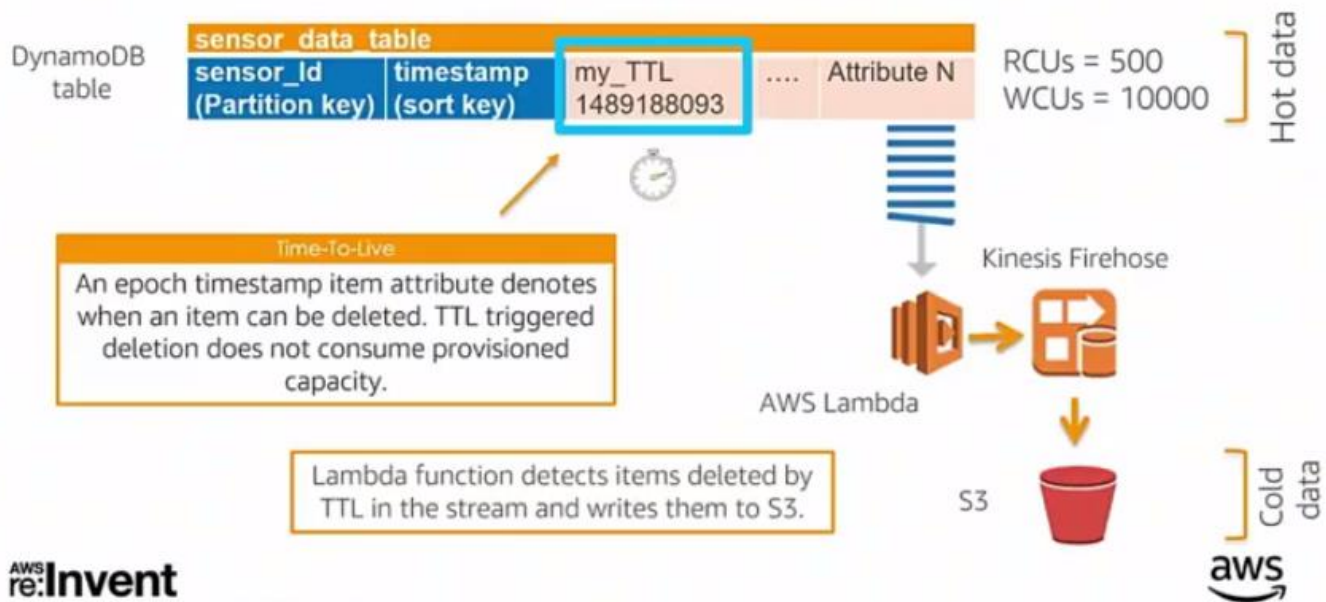
Queue-based load leveling using AWS Lambda

- ✓ Group multiple data points into a single item – save on writes to DynamoDB
- ✓ Use Lambda batch size to control DynamoDB item size
- ✓ Batch-write to DynamoDB
- ✓ Use DynamoDB TTL to manage data lifetime
- ✓ Optional: use compression



We can instead group the events/metrics and store them in a single item that Lambda then gets to process in a single call. We can save a lot of writes this way.

Using TTL to Age Out Cold Data



We can configure the TTL value using a lambda as an environment variable, we can then create a new lambda to detect items by their expiry data and ship them to S3 before deleting

Cost, Revised

Data rate: 100,000 data points per second

Data storage: 1 month's worth = ~2.5 TB

Lambda: \$2K – \$5K per month

Kinesis: 100,000 records -> 100 shards -> ~\$5K per month

DynamoDB: ~~100,000 WCU's -> \$50K per month~~

- Bin data points: 10 per item saves 10x in cost
- 10,000 WCU's -> \$5K per month (10x less...)
- Diff. over 1 year: \$600K vs. \$60K

We can save a lot by storing multiple data points in a single record (DynamoDB item)

Scaling Considerations

Adding more sensors:

- Kinesis: you need to add more shards
- Lambda: scales automatically based on Kinesis shards
- DynamoDB: scales automatically for space and throughput
 - Auto Scaling increases and decreases provisioned capacity as needed
 - For large spikes, provision capacity explicitly
 - Time-to-live (TTL) automatically deletes expired data without consuming provisioned capacity

Adding more events per sensor:

- Lambda function creates "denser" DynamoDB items
 - More data points stored in each DynamoDB item

Monitoring

Kinesis

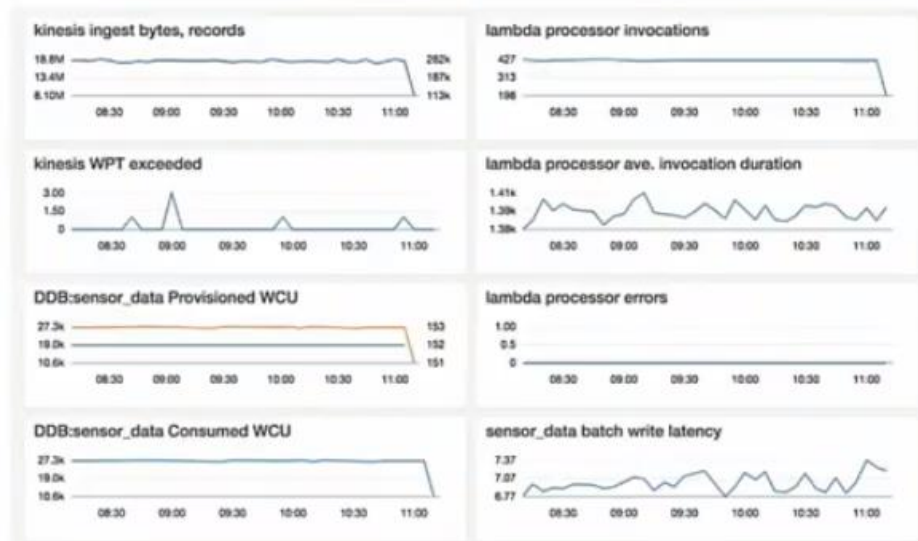
IncomingBytes (stream/shard)
IncomingRecords (stream/shard)
GetRecords.IteratorAgeMilliseconds
ReadProvisionedThroughputExceeded
WriteProvisionedThroughputExceeded

DynamoDB

SuccessfulRequestLatency
Throttling events/errors
Consumed/provisioned capacity
UserErrors
SystemErrors

Lambda

Invocations
Errors
Duration
Throttles
IteratorAge



Time Series Data: Takeaways

DynamoDB:

- Know your data
 - Structure
 - Access patterns
- Know the cost model
 - Writes are the most expensive resource
 - Attribute names are counted in the item size
- Optimization:
 - Binning—store multiple data points in a single item
 - Compression
 - TTL to delete cold data
 - Queue-based load leveling for uneven workload patterns

} Optimize

Lambda:

- Reuse database connections
- Test for optimal batch size, memory/CPU, and execution time

Monitor!

Serving Transactions Serverless

About Capital One Bank

- One of the 10 largest banks in the US based on deposits.
- Serving approximately **45 million** customer accounts across all products
- Online everywhere, with branches or cafes in many major cities



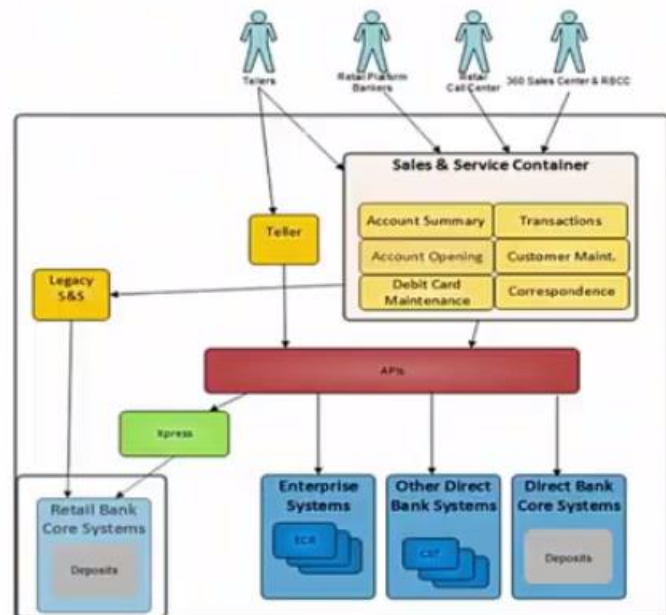
AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



About Capital One Bank

**The retail bank
has historically
been served by
one large
mainframe**



AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



About Capital One Bank

We made significant investments in our mobile experience...

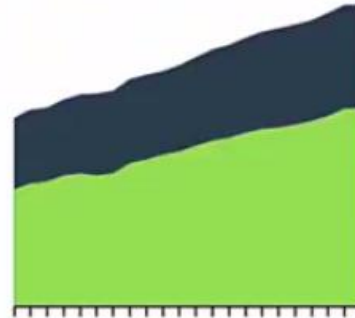


” Best big bank for mobile banking.
- nerdwallet

” 'Industry Star' & 'Best Payment/Banking' at the 8th annual Mobile Excellence Awards
- January 2016

And we've seen a large-and-growing amount of mobile traffic...

Mobile Monthly Unique Visitors



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

About Capital One Bank

Our product team is busy coming up with ways to use transaction data to help customers...



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



About Capital One Bank

All of this drove up server requests, and thus increasing speed and reliability problems

” Phone app is not up to par, laggy most of the time. It is very difficult to manage the app

” The site has either been slow or non-responsive

” It was my first time and it was a little slow for me



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



What did we have to solve?

- Select an architecture that could solve our needs
- Creating a one-time copy of all our transactions data in the cloud (18+ months)
- Handling results of batch processing (both overnight and throughout the day)
- Staying completely in sync with real time transactions as they happen
- Creating a new API in the cloud to expose the transactional data in the cloud
- Building a secure and resilient cloud infrastructure

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



What did we decide to do?

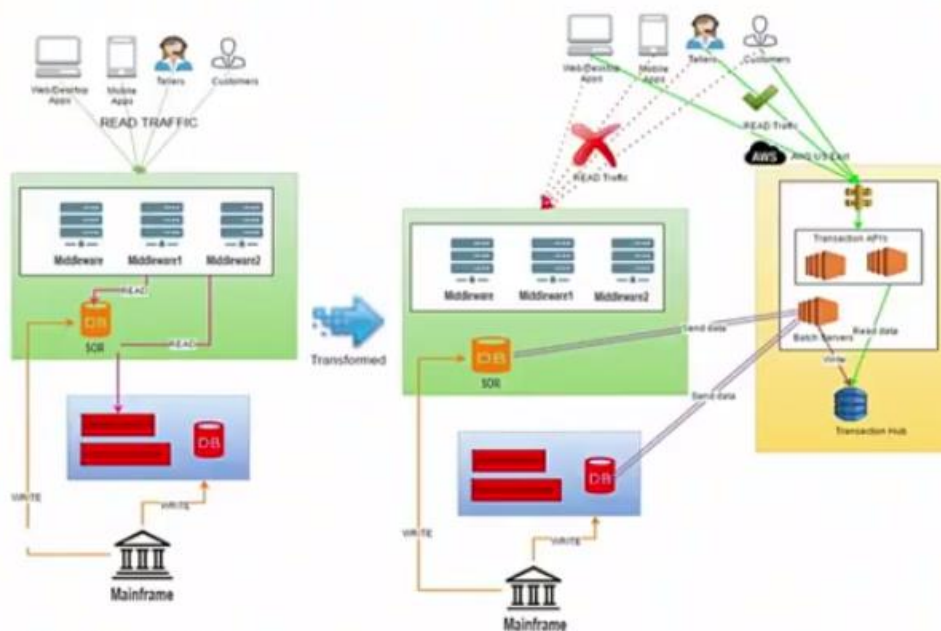
- Build a small team of engineers
- Make them responsible to solve the scalability/reliability problems, with a lean towards serverless event driven architecture
- Hold that team accountable for everything
 - Speed
 - Data quality
 - Data security

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Our CQRS Implementation



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

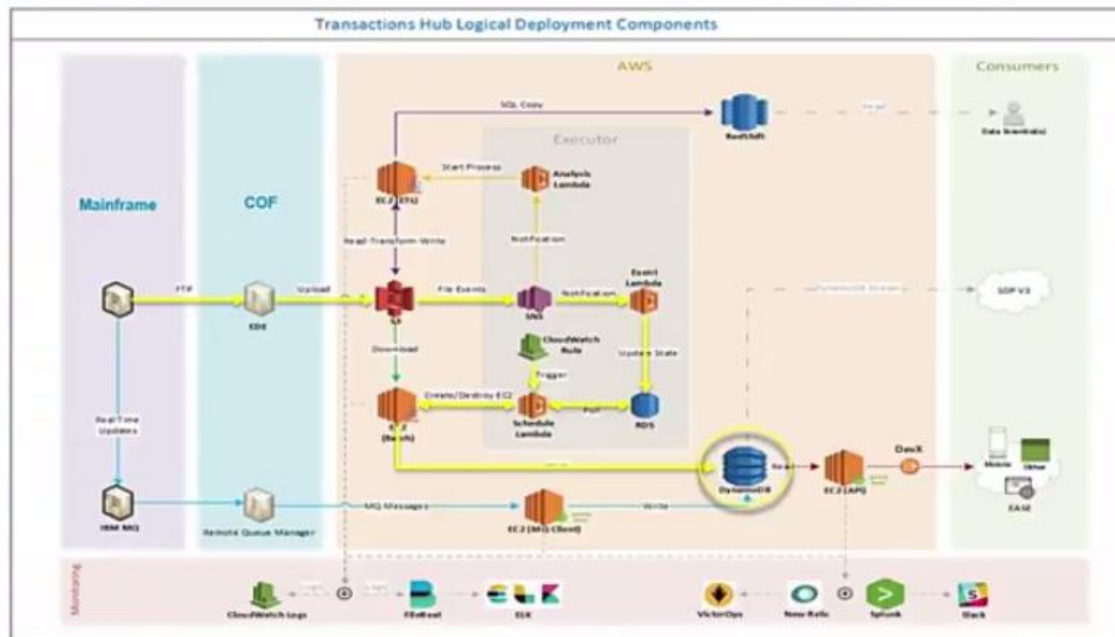


CQRS uses two commands, one for Writes and another for Reads.

[illegible][illegible]

All the files are encrypted in S3

Transactions Hub—Logical Architecture

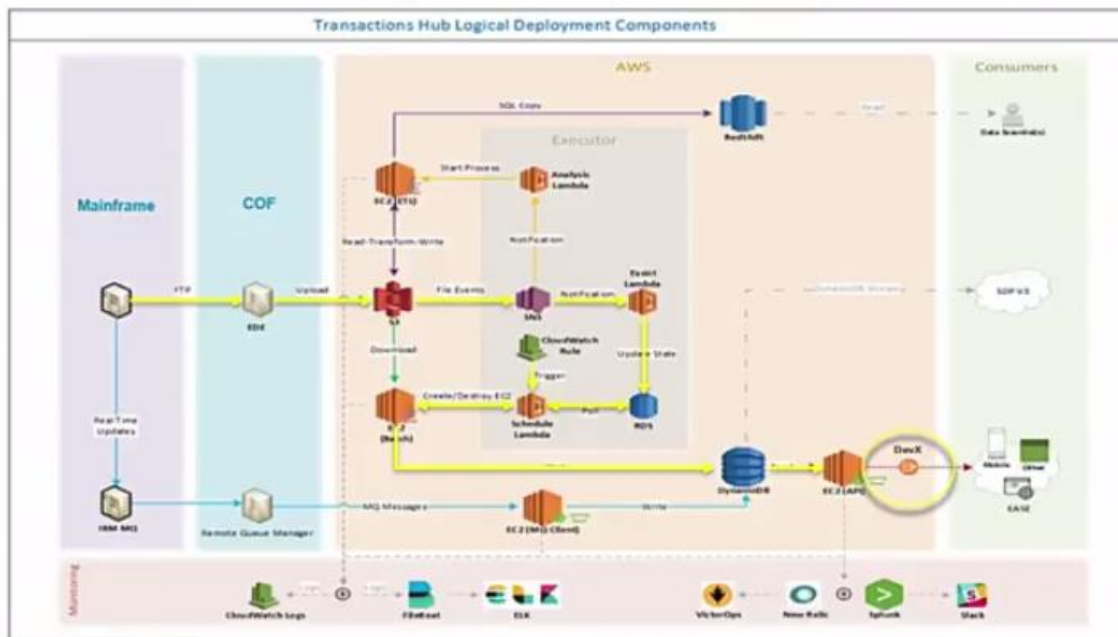


AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Transactions Hub—Logical Architecture



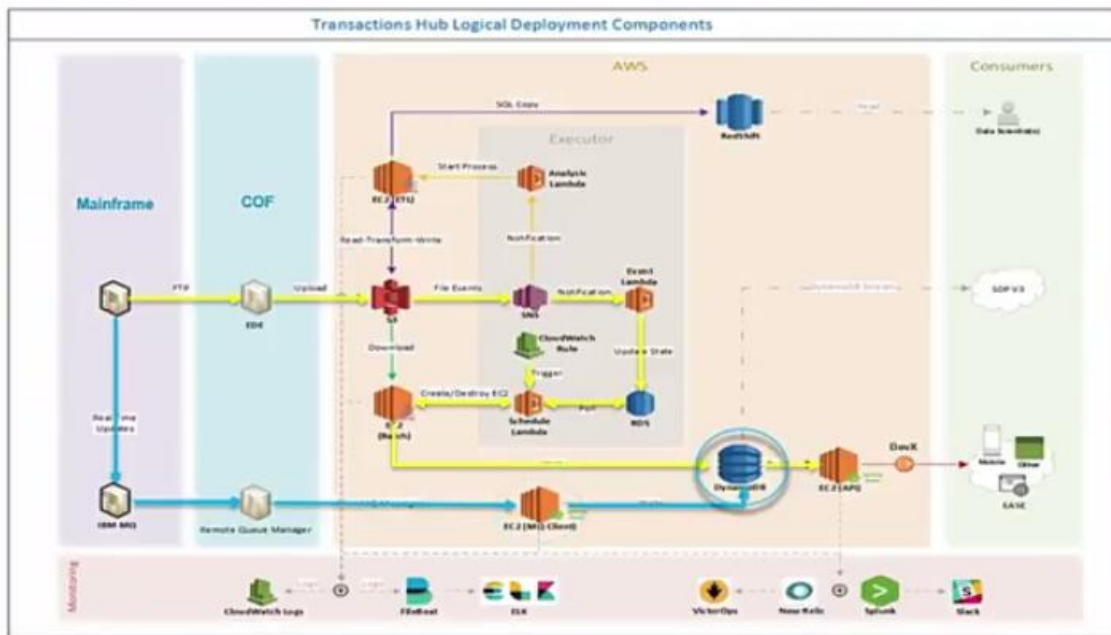
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We can now serve our channels off the data in DynamoDB using the new `getData()` API.

Transactions Hub—Logical Architecture















AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



The spring boot batch applications running on EC2 instances reads transactions as they happen by subscribing to various messaging topics, encrypts them, and then inserts them into DynamoDB.

Services We Used

- **Amazon S3** 
 - Used to store the file feeds coming in from the Core Systems, also used for archiving these files for now.
- **Amazon SNS** 
 - notification to humans
 - notification to Lambdas to signal file arrival
- **Lambda** 
 - Event based architecture where compute resources are created only when needed
 - Orchestration layer for batch jobs (DAILY, MEMO, and HISTORICAL)
- **Amazon CloudWatch rule** 
 - used to check for batch work
- **Amazon RDS** 
 - Used to persist the state of batch jobs
- **EC2/ASG/ELB** 
 - Used to run spring-batch jobs, spring-boot containers for MQ Listener and APIs.
- **DynamoDB** 
 - Used to store the transaction data
- **CloudWatch** 
 - Used to collect application logs, server logs, application metrics, etc.
- **IAM/KMS** 
 - Used to manage encryption keys

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



New Solution Is Much Faster



- Avg. Response time: 54.53 ms
- Total requests sent - 568956
- 10% of total requests had 77 ms response time
- 5% of total requests had 96 ms response time
- Only 1% of total requests had 345 ms response time
- Avg. TPS - ~ 145.85

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Full Suite of Business Monitoring



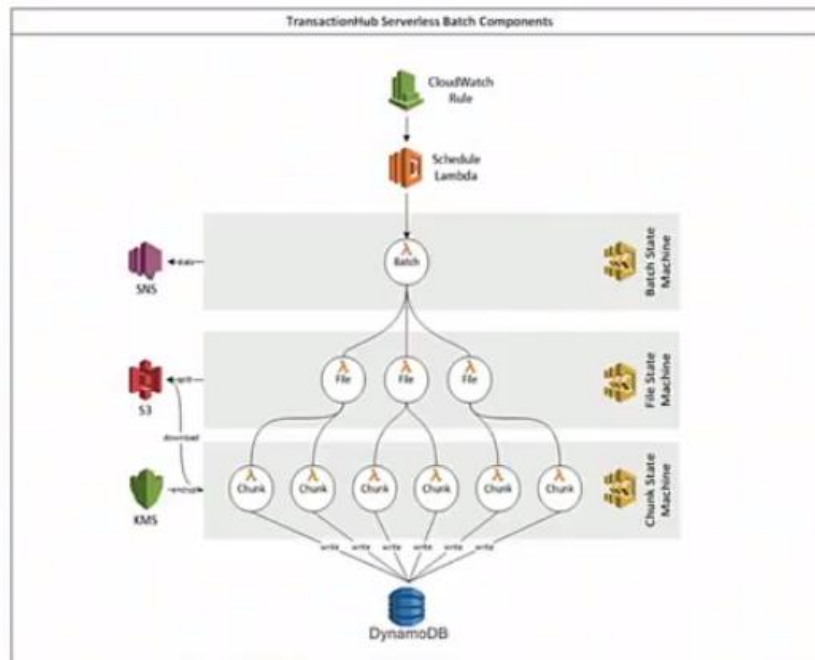
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



There is an ELK stack for logging and monitoring using dashboards for display

Transactions Hub Batch—Serverless



AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We are now replacing the Spring Boot applications running spring batch with lambda **step functions** on 3 levels, the **batch level**, the **file level**, and the **Chunk level** running step functions running as lambda functions to seamlessly read and ingest millions of records into DynamoDB.

Transaction Hub—Expanding Ecosystem

Statements

Generating email alerts for millions of customers using serverless Lambda Functions

Business APIs as Lambda Functions

Making statements available for our customers across channels via Lambda Functions

Second Look

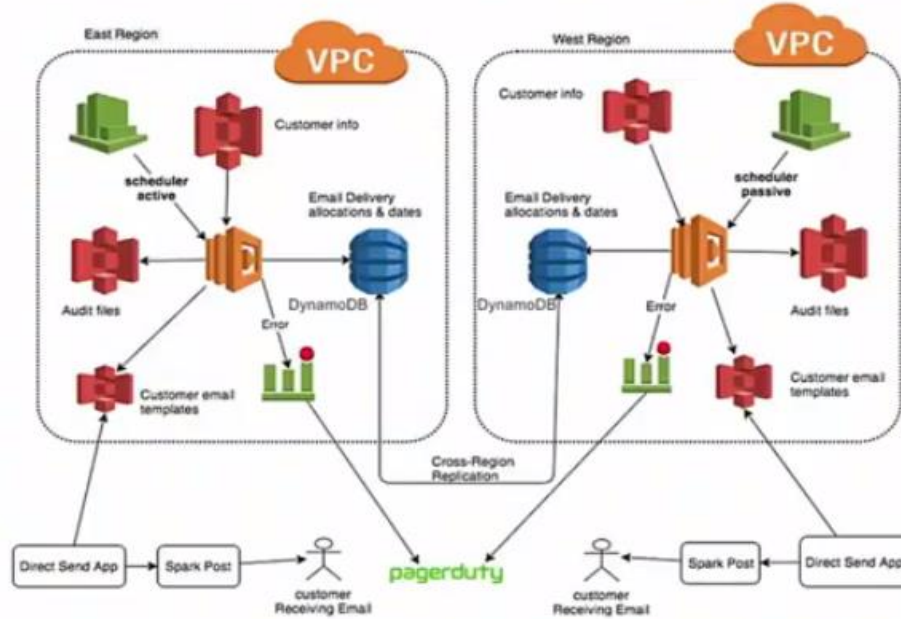
Leveraging DynamoDB streams to alert customers for certain transactions

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Statements: Serverless Email Delivery



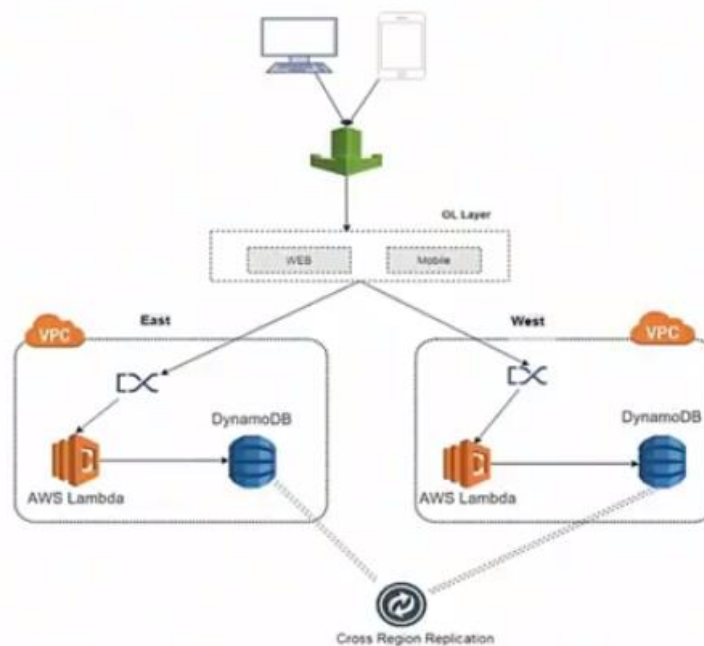
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



We are using a seamless lambda function that reads the list of customers to send information to based on their alert dates, the lambda then reads the data of the customers from S3 bucket, chooses the appropriate email templates, generates the email statements as an S3 file, then sends the S3 files to an email processing application that sends the emails to the customers notifying them of availability of new alerts and statements in their bank inboxes and on their digital channels.

Statements API as Lambda



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



This use case is where we are exposing our **getStatements()** API that runs on an EC2 instance to be replaced by a lambda function. This API provides the previous statements for the various channels via an API Gateway and lambda functions.

Second Look

- We have enabled streams on our DynamoDB transactions table
- AWS Lambda detects the new records as it polls the stream and executes the Lambda function
- From this lambda function we invoke the lambda functions for a feature called second look
- Second Look Lambda function alerts the customer for certain type of Transactions
- We are evaluating approximately 10 million transactions per day to check if we need to send alerts



AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Lessons Learned and Best Practices

- Know your data
- Always test at scale
- Know your access patterns upfront
- Know your indexes
- VPC endpoint
- Encryption
- Container reuse for Lambda

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



AWS re:Invent

Thank you!

We appreciate your feedback!

Check out:

DAT304 – DynamoDB – What's new
Wed, 3:15pm, Veronese 2405

AWS
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

