Modernizing the Amazon Database Infrastructure

Migrating from Oracle to AWS

March 8 2021

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Contents

Overview ... 1
Challenges with using Oracle databases ... 1
  Complex database engineering required to scale ... 1
  Complex, expensive, and error-prone database administration ... 2
  Inefficient and complex hardware provisioning ... 2
AWS Services ... 2
  Purpose-built databases ... 2
  Other AWS Services used in implementation ... 4
  Picking the right database ... 4
Challenges during migration ... 5
  Diverse application architectures inherited ... 5
  Distributed and geographically dispersed teams ... 6
  Interconnected and highly interdependent services ... 6
  Gap in skills ... 6
  Competing initiatives ... 7
People, processes, and tools ... 7
  People ... 7
  Processes and mechanisms ... 8
  Tools ... 11
Common migration patterns and strategies ... 11
  Migrating to Amazon DynamoDB – FLASH ... 11
  Migration to Amazon DynamoDB – Items and Offers ... 18
  Migrating to Aurora for PostgreSQL – Amazon Fulfillment Technologies (AFT) ... 25
  Migrating to Amazon Aurora – buyer fraud detection ... 32
Organization-wide benefits ... 36
Post-migration operating model ... 37

Distributed ownership of databases 37
Career growth 37
Contributors 38
Document revisions 38

Abstract

This whitepaper is intended to be read by existing and potential customers interested in migrating their application databases from Oracle to open-source databases hosted on AWS. Specifically, the paper is for customers interested in migrating their Oracle databases used by Online Transactional Processing (OLTP) applications to Amazon DynamoDB, Amazon Aurora, or open-source engines running on Amazon RDS.

The whitepaper draws upon the experience of Amazon engineers who recently migrated thousands of Oracle application databases to Amazon Web Services (AWS) as part of a large-scale refactoring program. The whitepaper begins with an overview of Amazon's scale and the complexity of its service-oriented architecture and the challenges of operating these services on on-premises Oracle databases. It covers the breadth of database services offered by AWS and their benefits. The paper discusses existing application designs, the challenges encountered when moving them to AWS, the migration strategies employed, and the benefits of the migration. Finally, it shares important lessons learned during the migration process and the post-migration operating model.

The whitepaper is targeted at senior leaders at enterprises, IT decision makers, software developers, database engineers, program managers, and solutions architects who are executing or considering a similar transformation of their enterprise. The reader is expected to have a basic understanding of application architectures, databases, and AWS.

Overview

The Amazon consumer facing business builds and operates thousands of services to support its hundreds of millions of customers. These services enable customers to accomplish a range of tasks such as browsing the Amazon.com website, placing orders, submitting payment information, subscribing to services, and initiating returns. The services also enable employees to perform activities such as optimizing inventory in fulfillment centers, scheduling customer deliveries, reporting and managing expenses, performing financial accounting, and analyzing data. Amazon engineers ensure that all services operate at very high availability, especially those that impact the customer experience. Customer facing services are expected to operate at over 99.90% availability leaving them with a very small margin for downtime.

In the past, Amazon consumer businesses operated data centers and managed their databases distinct from AWS. Prior to 2018, these services used Oracle databases for their persistence layer which amounted to over 6,000 Oracle databases operating on 20,000 CPU cores. These databases were hosted in tens of data centers on-premises, occupied thousands of square feet of space, and cost millions of dollars to maintain. In 2017, Amazon consumer facing entities embarked on a journey to migrate the persistence layer of all these services from Oracle to open-source or license-free alternatives on AWS. This migration was completed to leverage the cost effectiveness, scale, and reliability of AWS and also to break free from the challenges of using Oracle databases on-premises.

Challenges with using Oracle databases

Amazon recently started facing a growing number of challenges with using Oracle databases to scale its services. This section briefly describes three of the most critical challenges faced.

Complex database engineering required to scale

Engineers spent hundreds of hours each year trying to scale the Oracle databases horizontally to keep pace with the rapid growth in service throughputs and data volumes. Engineers used database shards to handle the additional service throughputs and manage the growing data volumes but in doing so increased the database administration workloads. The design and implementation of these shards were complex engineering exercises with new shards taking months to implement and test.

Several services required hundreds of these shards to handle the required throughput placing an exceptionally high administrative burden on database engineers and database administrators.

Complex, expensive, and error-prone database administration

The second challenge was dealing with complicated, expensive, and error-prone database administration. Database engineers spent hundreds of hours each month monitoring database performance, upgrading software, performing database backups, and patching the operating system (OS) for each instance and shard. This activity was tedious, and it had the potential to cause downtime and trigger a cascade of failures.

Inefficient and complex hardware provisioning

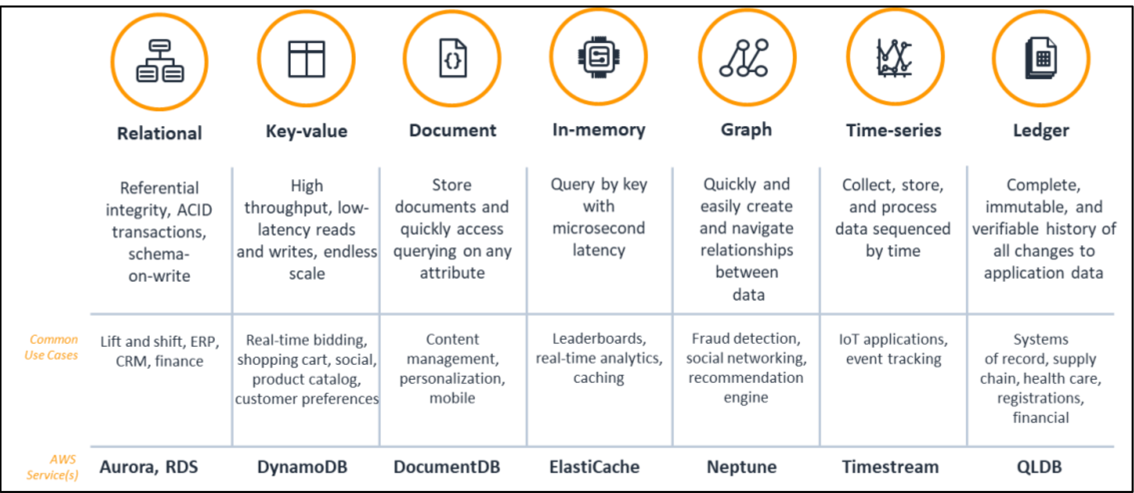
The third challenge was dealing with complex and inefficient hardware provisioning. Each year database engineers and the infrastructure management team expended substantial time forecasting demand and planning hardware capacity to meet it. After forecasting, engineers spent hundreds of hours purchasing, installing, and testing the hardware in multiple data centers around the world. Additionally, teams had to maintain a sufficiently large pool of spare infrastructure to fix any hardware issues and perform preventive maintenance. These challenges coupled with the high licensing costs were just some of the compelling reasons for the Amazon consumer and digital business to migrate the persistence layer of all its services to cloud native or open-source databases hosted on AWS.

AWS Services

This section provides an overview of the key AWS database Services used by Amazon engineers to host the persistence layer of their services. It also briefly describes other important AWS Services used by Amazon engineers as part of this transition.

Purpose-built databases

Amazon expects all its services be globally available, operate with microsecond to millisecond latency, handle millions of requests per second, operate with near zero downtime, cost only what is needed, and be managed efficiently. AWS services meet these requirements by offering a range of purpose-built databases thereby allowing Amazon engineers to focus on innovating for their customers.



	Relational	Key-value	Document	In-memory	Graph	Time-series	Ledger
	Referential integrity, ACID transactions, schema-on-write	High throughput, low-latency reads and writes, endless scale	Store documents and quickly access querying on any attribute	Query by key with microsecond latency	Quickly and easily create and navigate relationships between data	Collect, store, and process data sequenced by time	Complete, immutable, and verifiable history of all changes to application data
Common Use Cases	Lift and shift, ERP, CRM, finance	Real-time bidding, shopping cart, social, product catalog, customer preferences	Content management, personalization, mobile	Leaderboards, real-time analytics, caching	Fraud detection, social networking, recommendation engine	IoT applications, event tracking	Systems of record, supply chain, health care, registrations, financial
AWS Service(s)	Aurora, RDS	DynamoDB	DocumentDB	ElastiCache	Neptune	Timestream	QLDB

Range of databases offered by AWS

Amazon's engineers relied on three key database services to host the persistence layer of their services—**Amazon DynamoDB**, **Amazon Aurora**, and **Amazon RDS** for MySQL or PostgreSQL.

Amazon DynamoDB

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It is a fully managed, multi-region database with built-in security, backup and restore, and in-memory caching for internet-scale applications. Amazon DynamoDB service can handle trillions of requests per day and easily support over double-digit millions of requests per second across its entire backplane. You can start small or large and DynamoDB will automatically scale capacity up and down as needed.

Amazon Aurora

Amazon Aurora is a MySQL and PostgreSQL compatible relational database built for the cloud that combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open-source databases. Amazon Aurora is up to five times faster than standard MySQL databases and three times faster than standard PostgreSQL databases. It provides the security, availability, and reliability of commercial databases at 1/10th the cost.

Amazon Relational Database Service (Amazon RDS) for MySQL or PostgreSQL

Amazon RDS is a database management service that makes it easier to set up, operate, and scale a relational database in the cloud. It provides cost-efficient, resizable capacity for an industry-standard relational database and manages common database administration tasks.

Other AWS Services used in implementation

Amazon engineers also the following additional services in the implementation:

Amazon Simple Storage Service (Amazon S3): An object storage service that offers industry-leading scalability, data availability, security, and performance.

AWS Database Migration Service: A service that helps customers migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. The AWS Database Migration Service can migrate data to and from most widely used commercial and open-source databases.

Amazon Elastic Compute Cloud (Amazon EC2): A web service that provides secure, resizable compute capacity in the cloud designed to make web-scale cloud computing easier.

Amazon EMR: A service that provides a managed Apache Hadoop framework that makes it easy, fast, and cost-effective to process vast amounts of data across dynamically scalable Amazon EC2 instances.

AWS Glue: A fully managed extract, transform, and load (ETL) service that makes it easy for customers to prepare and load their data for analytics.

Picking the right database

Due to the wide range of purpose-built databases offered by AWS, each team could pick the most appropriate database based on scale, complexity, and features of its service. This approach was in stark contrast to the earlier use of Oracle databases where the service was modified to use a monolithic database layer. The following section describes the decision-making process used to pick the right persistence layer for a service.

Amazon engineers ran preliminary analysis on their database query and usage patterns and discovered that 70% of their workloads used single key-value operations that had little use for the relational features that their Oracle databases were offering. The access pattern for another 20% of the workloads was limited to a single table. Only 10% of the workloads used features of relational databases by accessing data across multiple keys. This discovery implied that most services were better served through a NoSQL store such as Amazon DynamoDB. Amazon DynamoDB offers superior performance at high throughputs and consumes less storage for sparse or semi-structured data sets than relational databases. Given the benefits of using Amazon DynamoDB, engineers running critical, high-throughput services decided to migrate their persistence layer to it.

Business units running services that use relatively static schemas, perform complex table lookups, and experience high service throughputs picked Amazon Aurora. Amazon Aurora provides the security, availability, and reliability of commercial databases at a fraction of their cost; and is fully managed by Amazon Relational Database Service (Amazon RDS) which automates tasks like hardware provisioning, database setup, patching, and backups.

Lastly, business units using operational data stores that had moderate read and write traffic, and relied on the features of relational databases selected Amazon RDS for PostgreSQL or MySQL for their persistence layer. Amazon RDS offers the choice of on-demand pricing with no up-front or long-term commitments or Reserved Instance pricing at lower rates—flexibility that was not previously available with Oracle. Amazon RDS freed up these business units to focus on operating their services at scale without incurring excessive administrative overhead.

Challenges during migration

The following section highlights key challenges faced by Amazon during the transformation journey. It also discusses mechanisms employed to successfully overcome these challenges and their outcomes.

Diverse application architectures inherited

Since its inception, Amazon has been defined by a culture of decentralized ownership that offered engineers the freedom to make design decisions that would deliver value to their customers. This freedom proliferated a wide range of design patterns and frameworks across teams. In parallel, the rapid expansion of the capabilities of AWS

allowed the more recent services to launch cloud-native designs. Another source of diversity was infrastructure management and its impact on service architectures. Teams needing granular control of their database hardware operated autonomous data centers whereas others relied on shared resources. This created the possibility of teams operating different versions of Oracle in a multitude of configurations. This diversity defied standard, repeatable migration patterns from Oracle to AWS databases. The architecture of each service had to be evaluated and the most appropriate approach to migration had to be determined.

Distributed and geographically dispersed teams

Amazon operates in a range of customer business segments in multiple geographies which operate independently. Managing the migration program across this distributed workforce posed challenges including effectively communicating the program vision and mission, driving goal alignment with business and technical leaders across these businesses, defining and setting acceptable yet ambitious goals for each business units, coordinating across a dozen time zones, and dealing with conflicts.

Interconnected and highly interdependent services

As described in the overview section, Amazon operates a vast set of microservices that are interconnected and use common databases. To illustrate this point, the item main databases maintain information about items sold on the Amazon website including item description, item quantity, and item price. This database, its replicas, and the service were frequently accessed by dozens of other microservices and ETLs. A single service losing access to the database could trigger a cascade of customer issues leading to unforeseen consequences. Migrating interdependent and interconnected services and their underlying databases required finely coordinated movement between teams.

Gap in skills

As Amazon engineers used Oracle databases, they developed expertise over the years in operating, maintaining, and optimizing them. As most of these databases were hosted on-premises, the engineers also gained experience in maintaining these data centers and managing specialty hardware. Most service teams shared databases that were managed by a shared pool of database engineers and the migration to AWS was a paradigm shift for them as they did not have expertise in:

- Open-source database technologies such as PostgreSQL or MySQL

- AWS native databases such as Amazon DynamoDB or Amazon Aurora
- NoSQL data modeling, data access patterns, and how to use them effectively
- Designing and building services that are cloud native

Competing initiatives

Lastly, each business unit was grappling with competing initiatives. In certain situations, competing priorities created resource conflicts that required intervention from the senior leadership.

People, processes, and tools

The previous section discussed a few of the many challenges facing Amazon during the migration journey. To circumvent these challenges, Amazon's leadership decided to invest significant time and resources to build a team, establish processes and mechanisms, and develop tooling to accelerate the intended outcomes. The following three sections discuss how three levers—people, processes, and tools—were engaged to drive the project forward.

People

One of the pillars of success was founding the Center of Excellence (CoE). The CoE was staffed with experienced enterprise program managers who led enterprise-wide initiatives at Amazon in the past. The leadership team ensured that these program managers had a combination of technical knowledge and program management capabilities. This unique combination of skills ensured that the program managers could converse fluently with software developers and database engineers about the benefits of application architectures and also engage with business leaders across geographies and business units to resolve conflicts and ensure alignment.

Key objectives

The key objectives of the CoE were:

- Define the overall program vision, mission and goals
- Define the goals for business units and service teams
- Define critical milestones for each service team and tracking progress against them

- Ensure business units receive resources and support from their leadership
- Manage exceptions and project delays
- Uncover technical and business risks, exposing them, and identifying mitigation strategies
- Monitor the health of the program and preparing progress reports for senior leadership
- Engage with the information security audit teams at Amazon to ensure that all AWS services meet data protection requirements
- Publish configurations for each AWS service that meets these data protection requirements; and perform audits of all deployments
- Schedule training for software developers and database engineers by leveraging SMEs from a variety of subject areas
- Identify patterns in issues across teams and engage with AWS product teams to find solutions
- Consolidate product feature requests across teams and engage with AWS product teams to prioritize them

Processes and mechanisms

This section elaborates on the processes and mechanisms established by the CoE and their impact on the outcome of the project.

Goal setting and leadership review

The program managers in the CoE realized early in the project that the migration would require attention from senior leaders. To enable them to track progress, manage delays, and mitigate risks the program managers established a monthly project review cadence. They used the review meeting to highlight systemic risks, recurrent issues, and progress. This visibility provided the leadership an opportunity to take remedial action when necessary. The CoE also ensured that all business segments prioritized the migration.

Establishing a hub-and-spoke model

Due to the large number of services, teams, and geographical locations that were part of the project, the CoE realized that it would be arduous and cumbersome to individually

track the status of each migration. Therefore, they established a hub-and-spoke model where service teams nominated a team member, typically a technical program manager, who acted as the spoke and the CoE program managers were the hub.

The spokes were responsible for:

- Preparing project plans for their teams
- Submitting these project plans to the CoE and receiving validation
- Tracking progress against this plan and reporting it
- Reporting major delays or issues
- Seeking assistance from the CoE to address recurrent issues

The hubs were responsible for:

- Validating the project plans of individual teams for accuracy and completeness
- Preparing and maintaining a unified database/service ramp down plan
- Maintaining open communications with each spoke to uncover recurrent issues
- Assisting service teams that require help
- Preparing project reports for leadership and escalate systemic risks

Training and guidance

A key objective for the CoE was to ensure that Amazon engineers were comfortable moving their services to AWS. To achieve this, it was essential to train these teams on open source and AWS native databases, and cloud-based design patterns. The CoE achieved this by

- Scheduling training sessions on open-source and AWS native databases
- Live streaming training sessions for employees situated in different time zones
- Scheduling design review sessions and workshops between subject matter experts and service teams facing roadblocks
- Scheduling tech talks with AWS product managers on future roadmaps
- Connecting teams encountering similar challenges through informal channels to encourage them to share knowledge

- Documenting frequently encountered challenges and solutions in a central repository

Establishing product feedback cycles with AWS

In the spirit of customer obsession, AWS constantly sought feedback from Amazon engineers. This feedback mechanism was instrumental in helping AWS rapidly test and release features to support internet scale workloads. This feedback mechanism also enabled AWS to launch product features essential for its other customers operating similar sized workloads.

Establishing positive reinforcement

In large scale enterprise projects, engineers and teams can get overwhelmed by the volume and complexity of work. To ensure that teams make regular progress towards goals, it is important to promote and reinforce positive behaviors, recognize teams, and celebrate their progress. The CoE established multiple mechanisms to achieve this, including the following initiatives:

- Broadcasting the success of teams that met program milestones and goals
- Opening communication channels between software developers, databases engineers, and program managers to share ideas and learnings
- Ensuring that the leaders on all teams were recognized for making progress

Risk management and issue tracking

Enterprise scale projects involving large numbers of teams across geographies are bound to face issues and setbacks. The CoE discovered that managing these setbacks effectively was crucial to project success. The following key mechanisms were used by the CoE to manage issues and setbacks:

- Diving deep into issues faced by teams to identify root cause of issues
- Support these teams with the right resources and expertise by leverage AWS support
- Ensure setbacks receive leadership visibility for remedial action
- Documenting these patterns in issues and their solutions
- Disseminating these learnings across the company

Tools

In the spirit of frugality, the CoE wanted to achieve more with minimal resources. Due to the complexity of the project management process, the CoE decided to invest in tools that would automate the project management and tracking. Tooling was built to

- Track active Oracle instances hosted in data centers
- Track the activity of these instances and understand data flow using SQL activity
- Tag databases to teams and individuals that own them; and synchronize this information with the HR database
- Track and manage database migration milestones using the tool in a single portal
- Prepare project status reports by aggregating the status of every service team

To meet these requirements, the CoE developed a web application tool that connects to each active Oracle instance, gathers additional information about it including objects and operations performed, and then displays this information to users through a web browser. The tool also allowed users to communicate project status, prepare status reports and manage exception approvals. It enhanced transparency, improved accountability, and automated the tedious process of tracking databases and their status, marking a huge leap in productivity for the CoE.

Common migration patterns and strategies

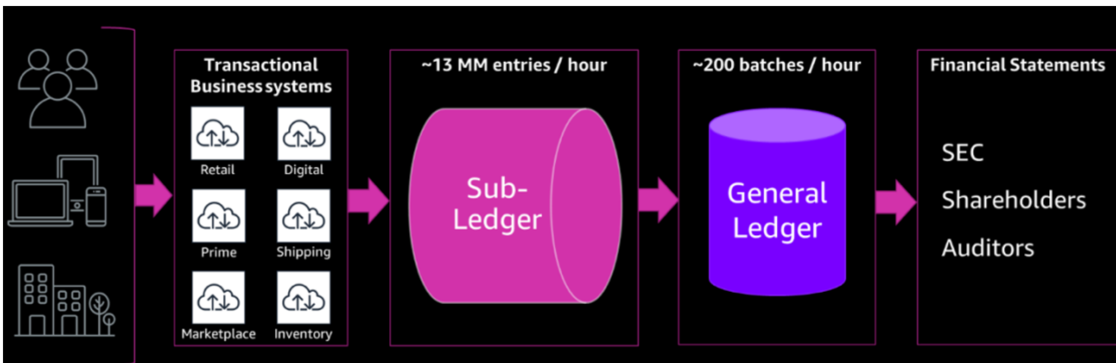
The following section describes the migration journey of four systems used in Amazon from Oracle to AWS. This section also provides insight on design challenges and migration strategies to enable readers to perform a similar migration.

Migrating to Amazon DynamoDB – FLASH

Overview of FLASH

Amazon operates a set of critical services called the Financial Ledger and Accounting Systems Hub (FLASH). FLASH services enable various business entities to post financial transactions to Amazon's sub-ledger. It supports four categories of transactions compliant with Generally Accepted Accounting Principles (GAAP)—account receivables, account payables, remittances, and payments. FLASH aggregates

these sub-ledger transactions and populates them to Amazon's general ledger for financial reporting, auditing, and analytics. Until 2018, FLASH used 90 Oracle databases, 183 instances, and stored over 120 terabytes of data. FLASH used the largest available Oracle-certified single instance hardware.



Data flow diagram of FLASH

Challenges with operating FLASH services on Oracle

As evident, FLASH is a high-throughput, complex, and critical system at Amazon. It experienced many challenges while operating on Oracle databases.

Poor latency

The first challenge was poor service latency despite having performed extensive database optimization. The service latency was degrading every year due to the rapid growth in service throughputs.

Escalating database costs

The second challenge related to yearly escalating database hosting costs. Each year, the database hosting costs were growing by at least 10%, and the FLASH team was unable to circumvent the excessive database administration overhead associated with this growth.

Difficult to achieve scale

The third challenge was negotiating the complex interdependencies between FLASH services when attempting to scale the system. As FLASH used a monolithic Oracle database service, the interdependencies between the various components of the FLASH system were preventing efficient scaling of the system.

These challenges encouraged the FLASH team to migrate the persistence layer of its services to AWS and rearchitect the APIs to use more efficient patterns.

Reasons to choose Amazon DynamoDB as the persistence layer

Among the range of database services offered by AWS, the FLASH engineers picked Amazon DynamoDB. The key reasons that the FLASH team picked DynamoDB follow.

Easier to scale

As DynamoDB can scale to handle trillions of requests per day and can sustain millions of requests per second, it was the ideal choice to handle the high throughput of FLASH. DynamoDB was also ideal due to its near infinite scaling capability.

Easier change management

Relational databases also make it complicated to change tables and schema definitions whereas NoSQL databases, such as DynamoDB, allow for increased flexibility. One item has some shared values, but otherwise every item can have different attributes. In addition, you can add attributes to items any time with no downtime like altering a table in a relational database.

Speed of transactions

Lastly, single key value pair lookups are faster and more efficient on Amazon DynamoDB when compared to a relational database for a variety of reasons such as lower memory usage and automatic partition management.

Easier database management

With DynamoDB, there are no servers to provision, patch, or manage, and no software to install, maintain, or operate. The FLASH team could create full backups of hundreds of terabytes of data instantly with no performance impact to their tables, and recover to any point in time in the preceding 35 days with no downtime.

Challenges and design considerations during refactoring

FLASH engineers realized that designing a robust architecture for FLASH on DynamoDB was essential to achieve scalable performance. The FLASH team faced the following challenges during the re-design of its services on DynamoDB: providing an authoritative booking time for transactions, indexing transactions on a time ordered queue, ensuring accessibility of data to downstream services, and migrating historical data with no loss.

Time stamping transactions and indexed ordering

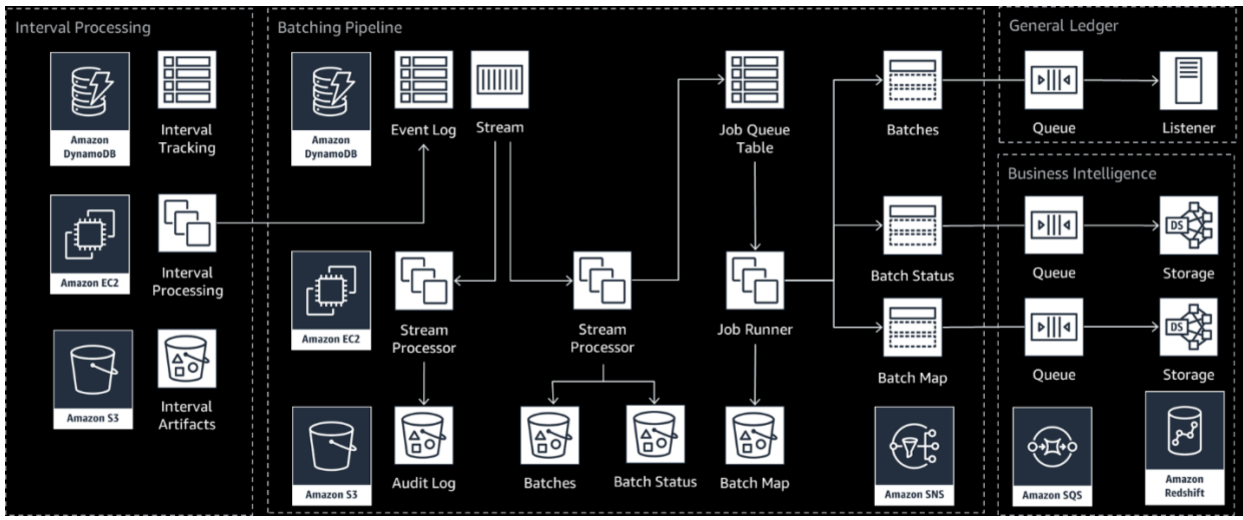
A key requirement for all upstream services requesting transactions from FLASH is a time stamp. These timestamps known as booking dates, help them keep a record of the day and time of transaction. In the previous setup, a single Oracle server committed the transaction and assigned a timestamp. In the distributed environment, the time-stamping and the transaction commit were separated into two different systems. A set of EC2 instances were used to time-stamp incoming transactions. The clock time across these instances were synchronized using NTP consensus algorithms without the use of expensive hardware. After a timestamp was assigned, these transactions were logged in a S3 bucket for durable backup. DynamoDB Streams along with Amazon Kinesis Client Libraries were used to ensure exactly-once, ordered indexing of records. DynamoDB Streams is a powerful service that can combine with other AWS services to solve the problem of ordered delivery and serialization. When enabled, DynamoDB Streams captures a time-ordered sequence of item-level modifications in a DynamoDB table and durably stores the information for up to 24 hours. Applications can access a series of stream *records*, which contain an item change, from a DynamoDB stream in near real time. DynamoDB Streams writes a stream record whenever one of the following events occurs:

- A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
- An item is updated: The stream captures the before and after image of any attributes that were modified in the item.
- An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

After a transaction appears on the DynamoDB stream, it is routed to a Kinesis stream and indexed. These indexes are written back to the records on DynamoDB. The FLASH team used the fact that DynamoDB allows the creation of one or more secondary indexes on a table. A secondary index lets applications query the data in the table using an alternate key, in addition to queries against the primary key. DynamoDB does not require that applications use indexes, but it provides them the flexibility when querying data, especially when the data has many to many relationships. After creating a secondary index on a table, FLASH can read data from the index in much the same way as it does from the table. At the time of the implementation, each table in DynamoDB had a default limit of five global secondary indexes and five local secondary indexes per table.

Providing data to downstream services

A critical requirement of all accounting systems in general, and FLASH in particular, is to enable financial analytics. Previously on Oracle, the same Oracle instances served as compute clusters for analytics thus increasing the workloads on these nodes. FLASH switched the model to an event-sourcing model where an S3 backup of commit logs was created continuously. The team also eliminated the use of unstructured and disparate tables for analytics and data processing as they increased the requirement for processor capacity. The previous system exhibited non-determinism. The team created a single source of truth and converged all the data models to the core event log/journal, to ensure deterministic data processing. Amazon S3 was used as an audit trail of all changes to the DynamoDB journal table. Amazon Simple Notification Service (Amazon SNS) was used to publish these commit logs in batches for downstream consumption. The artifact creation was coordinated using Amazon Simple Queue Service (SQS). The entire system is SOX (Sarbanes-Oxley Act) compliant (SOX is also known as the Corporate and Auditing Accountability, Responsibility, and Transparency Act). These data batches were delivered to the general ledger for financial reporting and analysis.

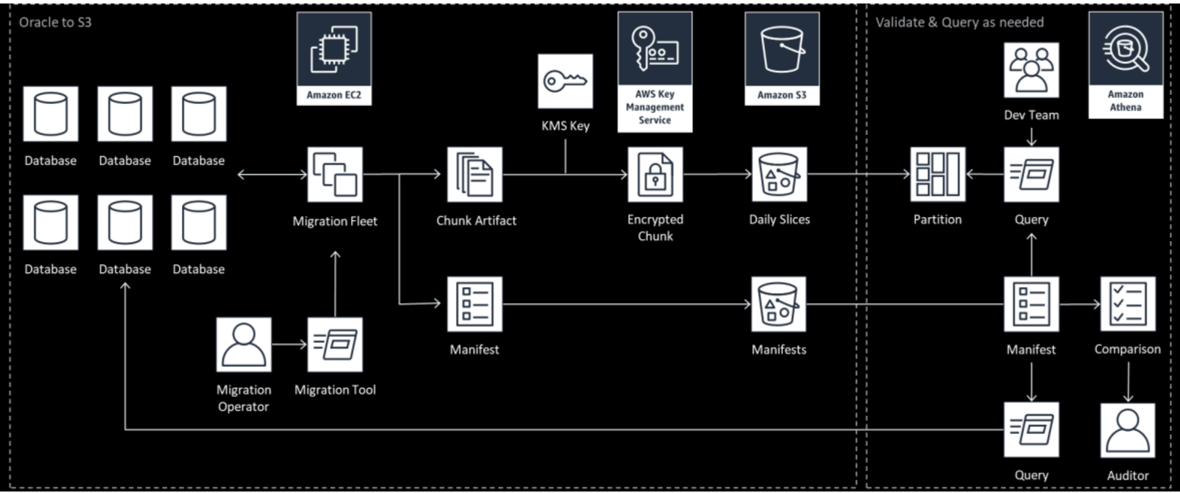


Streaming aggregation pipeline

Archiving historical data

Archiving historical data across multiple Oracle databases was an important activity to perform before decommissioning them. FLASH implemented a pay-as-you go system to query historical data and maintaining a 'hot' database, that is queried rarely, was determined to be too expensive. As a result, FLASH used a common data model and columnar format for ease of access and migrated historical data to Amazon S3 buckets that are accessible by Amazon Athena. Amazon Athena was ideal as it allows for a

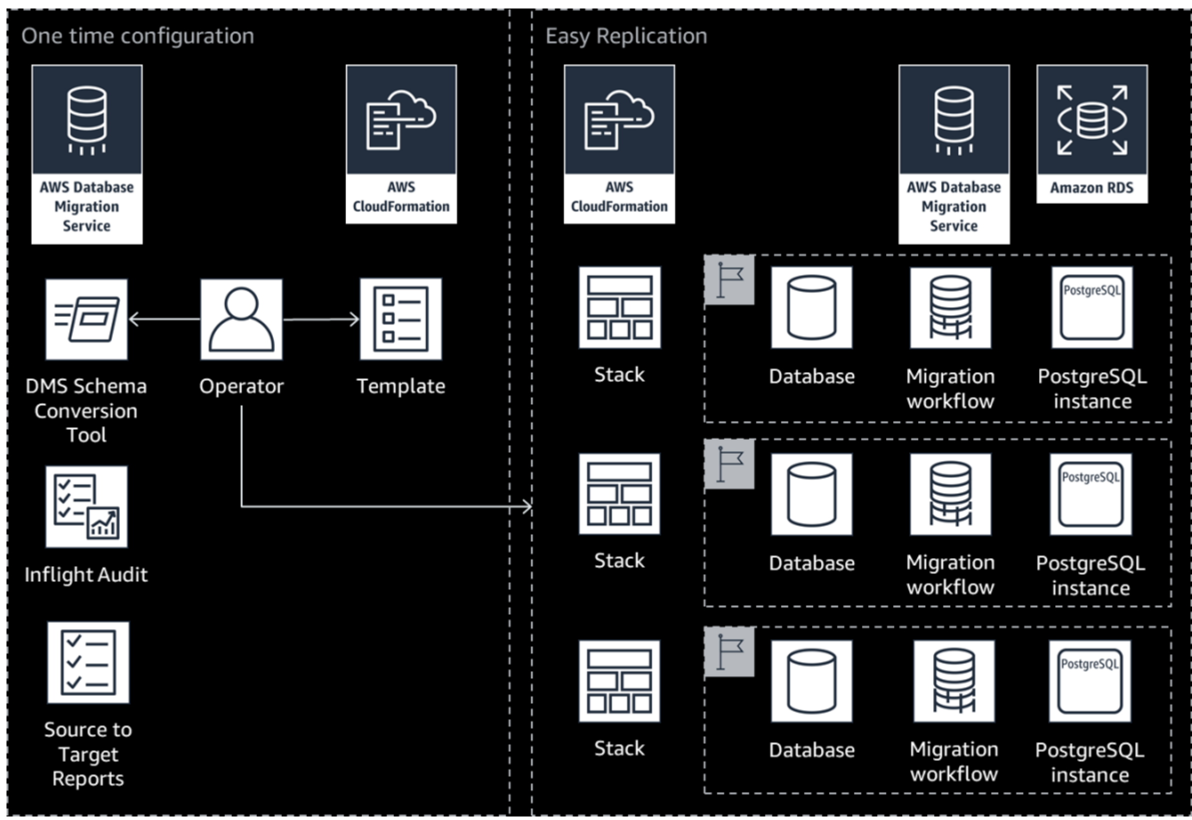
query-as-you-go model which works well as this data is queried on average once every two years. Also, because Amazon Athena is serverless, there is no requirement to manage infrastructure.



Subledger convergence and archival

Performing data backfill

In one particular instance of service migration, the FLASH team needed to migrate legacy data to the new persistence layer and had to pick the most effective strategy to achieve this. The team used AWS Database Migration Service (AWS DMS) to ensure reliable and secure data transfer. AWS DMS was simple to use as there was no need to install any drivers or applications, and it did not require changes to the source database in most cases. AWS DMS also supported an automated migration, easily converted data from Oracle to non-Oracle database engines, and offered a one-click setup for partition comparison and audits. It is also SOX compliant from source to target, provided the team granular insights during the process, and cost a few hundred dollars for the entire migration. To verify the accuracy of historical data transfer, AWS DMS was used to perform a row-by-row validation.



Lift and shift using AWS DMS and RDS

To ensure compliance with SOX, FLASH published recommendations to ensure its engineers selected the right access control mechanisms and encryption parameters. The team also evaluated the compliance and reliability of each AWS service and prepared pre-configured AWS CloudFormation templates to each service team. The team also established strict performance criteria to ensure high performance. Performance metrics that are monitored include average latency, P99 latency, read/write failures, free memory and CPU utilization. Before putting each service into production, each service was tested for peak throughput expectancies.

Benefits

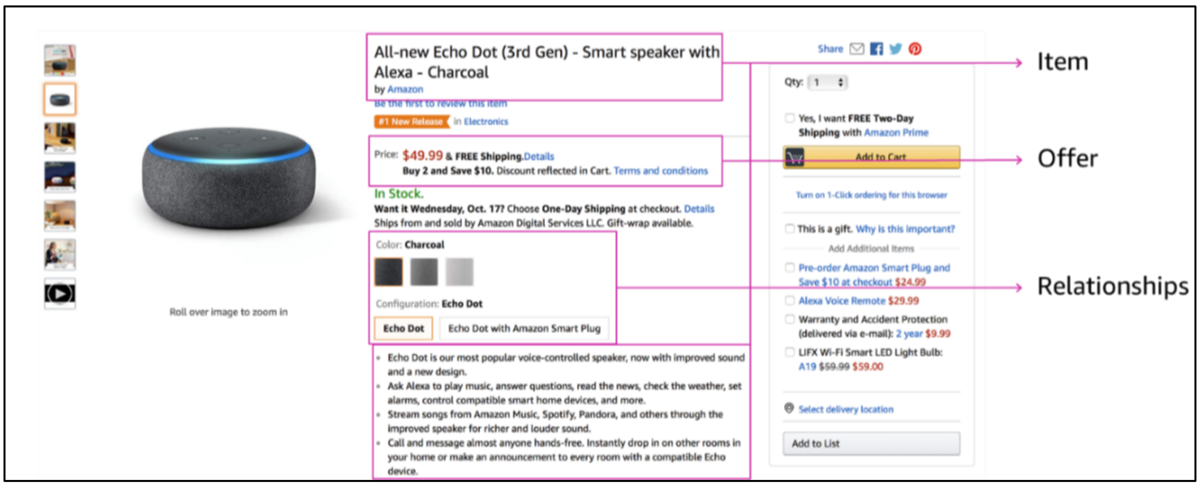
Rearchitecting the FLASH system to work on AWS database services significantly improved its performance. Critical services that moved to DynamoDB saw a 40% reduction in average latency despite handling twice the traffic. Although FLASH provisioned more compute and larger storage, the database operating costs have remained flat or reduced despite processing higher throughputs. This is possible due to the automatic scaling capabilities of AWS services. Additionally, the migration has reduced administrative overhead by over 70% enabling engineers to focus on

optimizing the application layer and worry less about the persistence layers. Automatic scaling has also allowed the FLASH team to reduce costs by dynamically responding to traffic spikes. Overall, the shift to AWS has liberated engineers to work more efficiently and concentrate on innovating.

Migration to Amazon DynamoDB – Items and Offers

Overview of Items and Offers

Amazon offers hundreds of millions of unique products for sale to its customers. To manage the lifecycle of these items and their associated offers, Amazon operates a set of services collectively called Items and Offers. The Items and Offers system manages three components associated with an item – item information, offer information, and relationship information. Item information constitutes product title, product description and product details; offer information constitutes item price and seller information; and the relationships are the different variants of an item such as color, size, and quantity.



Overview of the Items and Offers service

A key service within the Items and Offers system is the Item Service which updates the item information by ingesting updates from millions of sellers and uses multiple workflows to process the three components – items, offers, and relationships. Historically, Item Service used Oracle databases exclusively for its persistence layer.

Challenges faced when operating Item Service on Oracle databases

The Item Service team was facing many challenges when operating on Oracle databases.

Challenging to administer partitions

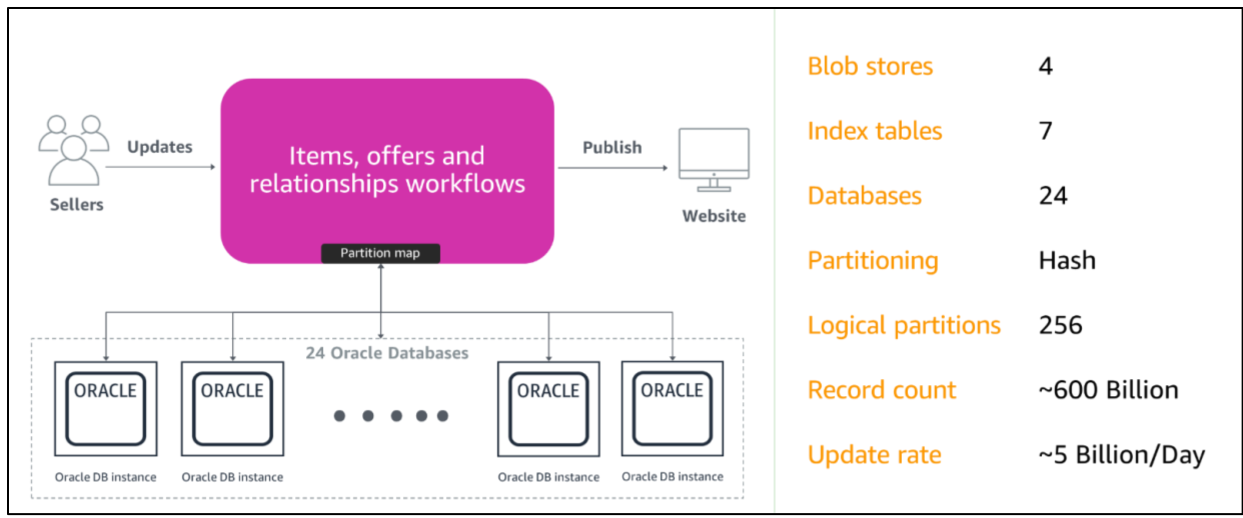
For efficiency, the Item data was partitioned at the service level using hashing, and partition maps were used to route requests to the correct partition. These partitioned databases were becoming difficult to scale and manage. To keep pace with the rapidly growing service throughputs, the team had to spend over twenty weeks each year creating new partitions, updating the hash maps, and testing for scale.

Difficult to achieve high availability

To optimize space utilization by the databases, all tables were partitioned and stored across 24 databases. This configuration exacerbated availability issues as failure of a single database had the potential to create service failure leading to product information becoming inaccessible or outdated for customers and sellers. The fear of suboptimal customer experience failure forced teams to perform frequent maintenance activities which was becoming an unmanageable overhead for the database administrators.

Reaching scaling limits

Due to the preceding challenges of operating the Items and Offers system on Oracle databases, the team was not able to support the growing service throughputs.



Scale of the Item Service

Reasons for choosing Amazon DynamoDB

In 2018, the Item Service unit decided to redesign the Item Service and migrate its persistence layer to AWS. Among the range of databases offered by AWS, Amazon DynamoDB was the best suited persistence layer for IMS. It offered an ideal

combination of features suited for easily operating a highly available and large-scale distributed system like IMS.

Automated database management

One of the biggest benefits of DynamoDB was the fact that the complexity of running a massively scalable, distributed database was managed by the service itself, allowing software developers to focus on building and innovating the service rather than managing infrastructure. DynamoDB also manages the sophisticated distributed computing concepts allowing the developer to focus solely on how to best use the APIs in the AWS SDK.

Automatic scaling

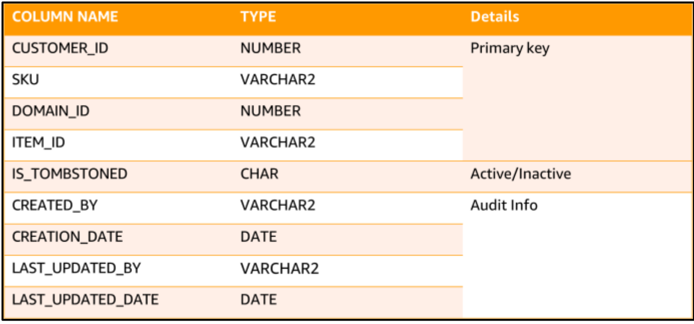
DynamoDB delivers high throughput through horizontal scaling where data and workloads are automatically partitioned over a number of shards by the service. The service grows the partitions for you as the data volume and throughput increase. If the automatic scaling parameters are set correctly and the schema is designed well, the performance of DynamoDB remains consistent despite traffic growth.

Cost effective and secure

Other than being easy to scale, DynamoDB is cost effective and offers a choice of reserved and on-demand capacity to manage throughput. DynamoDB also offers support for end-to-end encryption and fine-grained access controls via AWS Identity and Access Management.

The most distinctive features of DynamoDB for the Item Service team were the low administrative overhead, high performance, and enterprise grade security and availability.

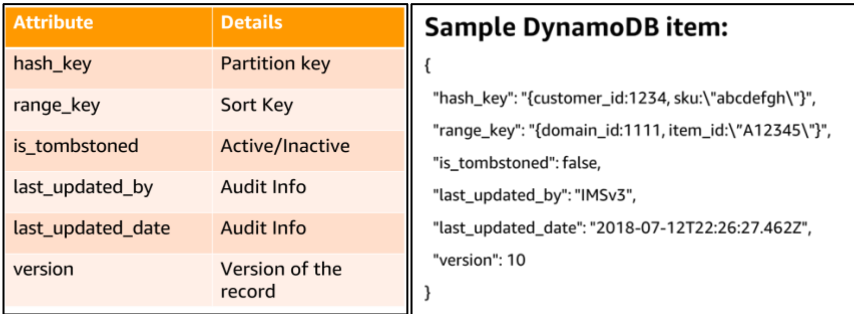
After selecting DynamoDB as the new persistence layer, the Item Service team started redesigning the data model to work on DynamoDB. To illustrate the redesign, The following figure displays one of the index tables on Oracle that stored SKU to ASIN mappings. As shown, it was a simple table containing two key columns along with state and audit information. When modeling the table on DynamoDB, composite keys were used. A composite primary key comprises two attributes—partition key and sort key. DynamoDB uses the partition key value as input to an internal hash function; the output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item is stored. All items with the same partition key values are stored together and sorted by sort key value.



COLUMN NAME	TYPE	Details
CUSTOMER_ID	NUMBER	Primary key
SKU	VARCHAR2	
DOMAIN_ID	NUMBER	
ITEM_ID	VARCHAR2	
IS_TOMBSTONED	CHAR	Active/Inactive
CREATED_BY	VARCHAR2	Audit Info
CREATION_DATE	DATE	
LAST_UPDATED_BY	VARCHAR2	
LAST_UPDATED_DATE	DATE	

Table structure of Item Service on Oracle

The following figure shows the equivalent table represented in DynamoDB. All other Item Service schemas were redesigned using similar principles.



Attribute	Details
hash_key	Partition key
range_key	Sort Key
is_tombstoned	Active/Inactive
last_updated_by	Audit Info
last_updated_date	Audit Info
version	Version of the record

Sample DynamoDB item:


```
{
  "hash_key": "{customer_id:1234, sku:\"abcdefgh\"}",
  "range_key": "{domain_id:1111, item_id:\"A12345\"}",
  "is_tombstoned": false,
  "last_updated_by": "IMSv3",
  "last_updated_date": "2018-07-12T22:26:27.462Z",
  "version": 10
}
```


Table structure of Item Service on DynamoDB

Execution

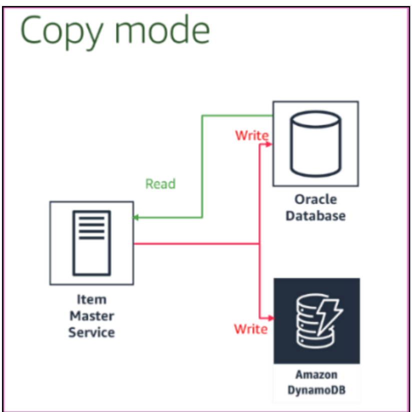
After building the new data model, the next challenge was performing the migration. Item Service is a critical service that processes seller updates and therefore must be fully available and operate at scale throughout the migration. The team had two months to backfill historical data of approximately 100 TB before the next scaling cycle. The Item Service team devised a two-phased approach to achieve the migration—live migration and backfill.

Live migration

The goal of the live migration phase was to transition the main store from Oracle to DynamoDB without any failures and actively migrate all the data being processed by the application. As is illustrated in the following figure, the item Service team used three stages to achieve the goal—copy mode, compatibility mode, and move mode.

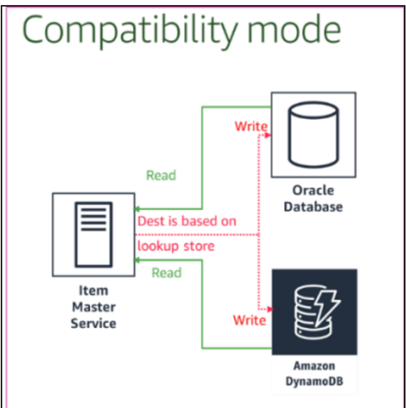
In the copy mode, the application was modified to write simultaneously to both Oracle and DynamoDB stores and perform reads exclusively from Oracle. The purpose of this

mode was to validate the correctness, scale, and performance of DynamoDB. After validating the scale, accuracy, and performance data was purged from the DynamoDB store to start the live migration.



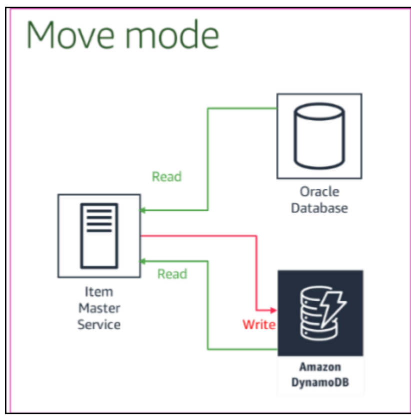
Copy mode

The next mode was compatibility mode which acted as the staging to prepare the application for switching the main store. At this stage, the application was aware of both data stores and could determine the main store. Compatibility mode allowed the Item Service team to pause the migration should issues arise.



Compatibility mode

In the final mode, called the move mode, DynamoDB was designated as the main store and reads were served by reading both the stores and combining the results. After the move mode, the Item Service team began the backfill phase of migration that would make DynamoDB the single main database and deprecate Oracle.

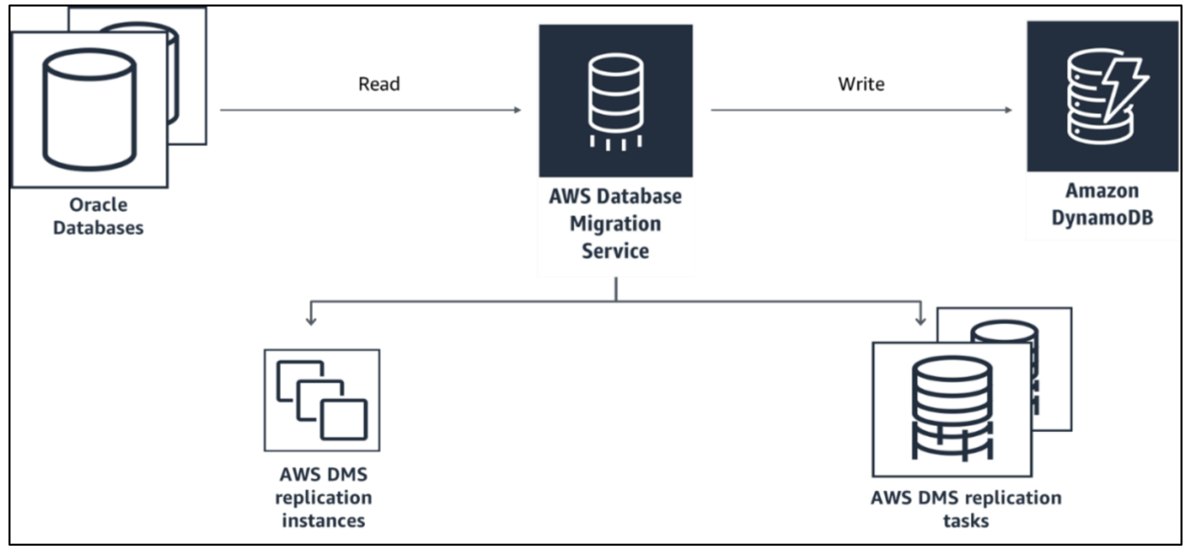


Move mode

Backfill

During the backfill phase, AWS Database Migration Service (AWS DMS) was used to backfill records that were not migrated by the application write logic. The DMS service was easy to set up and its execution involved creating and configuring replication instances, tasks, and endpoints. Once AWS DMS was configured correctly, the next step was to scale it up to achieve the desired throughput for migration. Oracle source tables were partitioned across 24 databases and the destination store on DynamoDB was elastically scalable. The Item Service business unit scaled the migration by running multiple AWS DMS replication instances per table and each instance had parallel loads configured. The business unit achieved a throughput of 100,000 to 150,00 TPS and migrated 600 billion records in about two months by running 70 DMS instances in parallel. To handle AWS DMS replication errors, the Item Service business unit automated the process by creating a library using the AWS DMS SDK. As with any large-scale migration or change, there is a potential for issues. Therefore, the team designed and tested fallback mechanisms for rollback and recovery.

The final step was to fine tune configurations on AWS DMS and Amazon DynamoDB to maximize the throughput and minimize cost.



Backfill process of IMS

Benefits

The overall migration to DynamoDB delivered all the benefits the business unit hoped for and more. Previously, in preparation for peak-load events such as Prime Day, the engineering team had to fine-tune and test database configurations which took on average two weeks. DynamoDB has reduced this time to just a few hours. DynamoDB also supports a simplified architecture using global secondary indexes to improve reliability and availability for faster query processing and record retrieval. Additionally, Amazon simplified the persistence layer for the application by eliminating the need to maintain partition information and related logic to route data. After the migration, the availability of Item Service has improved, ensuring consistent performance and significantly reduced the operational workload for the team. After the migration, the team used the point-in-time recovery feature to simplify backup and restore operations. The team received these benefits at a lower overall cost than previously, due to dynamic automatic scaling capacity feature. The team also had the flexibility to adjust the provisioned read/write capacity based on actual usage instead of having a fixed capacity based on peak loads. This allows the service to grow with little effort and without extensive engineering. The reliable, consistent performance of DynamoDB and virtually no limits on scalability means Amazon customers can enjoy the experience they have come to expect as Amazon grows.

Migrating to Aurora for PostgreSQL – Amazon Fulfillment Technologies (AFT)

Overview of AFT

Amazon and its partners deliver hundreds of millions of orders each year to customers and fulfillment centers (FC) are the backbone of this delivery network. Amazon manages more than 150 FCs in North America and hundreds more around the world. The largest fulfillment centers occupy over a million square feet, employ thousands of associates, and process over a million orders each day. The Amazon Fulfillment Technologies (AFT) business unit builds and maintains the dozens of services that facilitate all fulfillment activities. A set of services called the Inventory Management Services facilitate inventory movement and are used by all other major services to perform critical functions within the FC. These functions include receiving inbound shipments, dispatching outbound shipments, picking items, sorting and packaging items and managing inventory state throughout. All of these functions are critical to customer fulfillment and are expected to operate at near perfect availability. Since the inception of Amazon.com, all of these services used Oracle databases as their persistence layer. Prior to 2018, over 300 Oracle databases were used to support these operations with the Oracle databases hosting terabytes of data on each database and supporting dozens of critical services.

Challenges faced operating AFT on Oracle databases

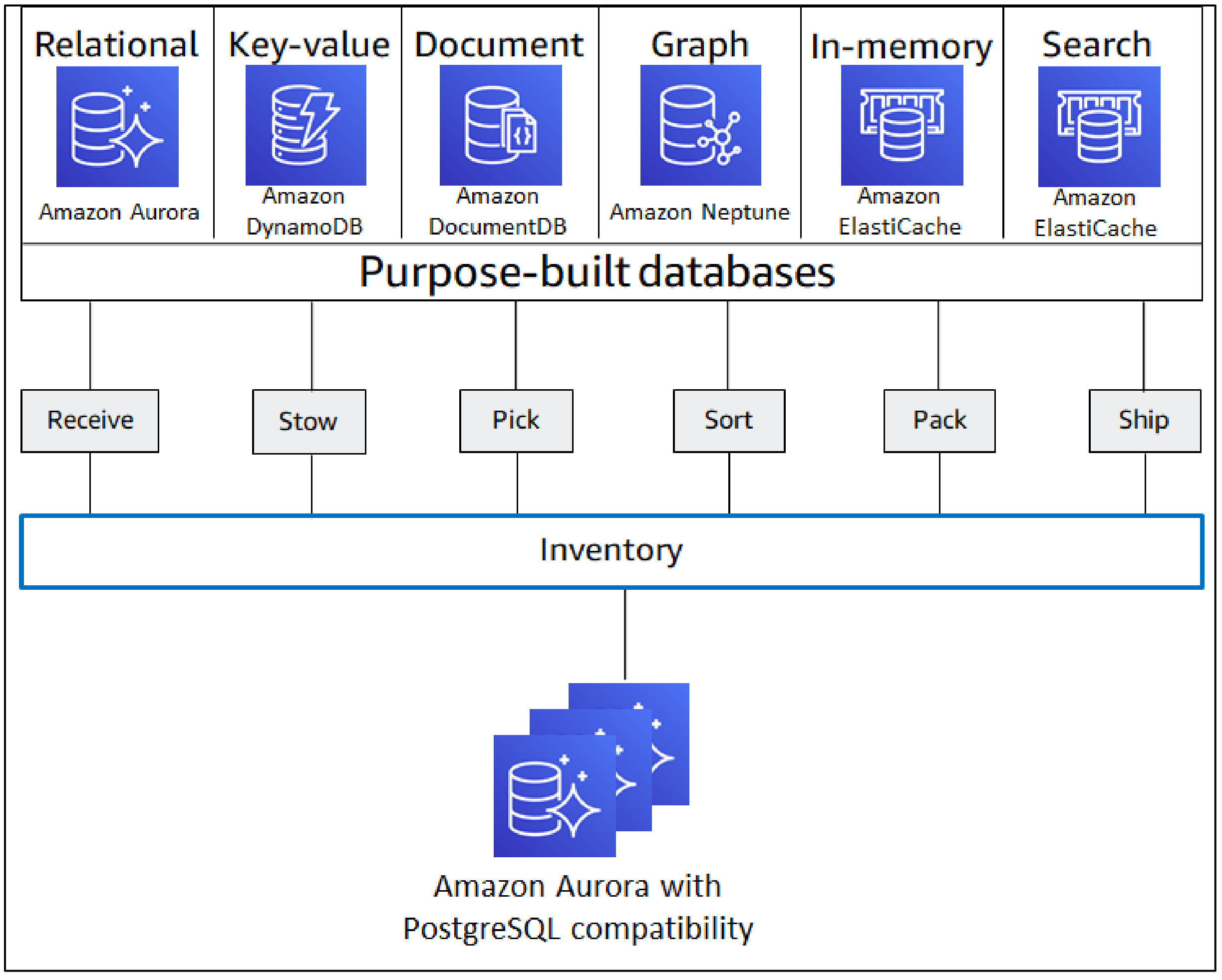
The AFT team faced many challenges operating its services on Oracle databases in the past.

Difficult to scale

All the services were becoming difficult to scale and were facing availability issues during peak throughputs due to both hardware and software limitations.

Complex hardware management

Hardware management was also becoming a growing concern due to the custom hardware requirements required from these Oracle clusters. As part of the migration effort, each AFT service team had the freedom to pick the most appropriate database service from the range of database services that AWS offers. The Inventory Management Services team decided to migrate to Amazon Aurora for PostgreSQL.



Databases services used by AFT

Reasons for choosing Amazon Aurora for PostgreSQL

The Inventory Management Services supports six key tasks at Amazon's fulfillment centers—receive, stow, pick, sort, pack, and ship. The team picked Amazon Aurora for three primary reasons.

Static schemas and relational lookups

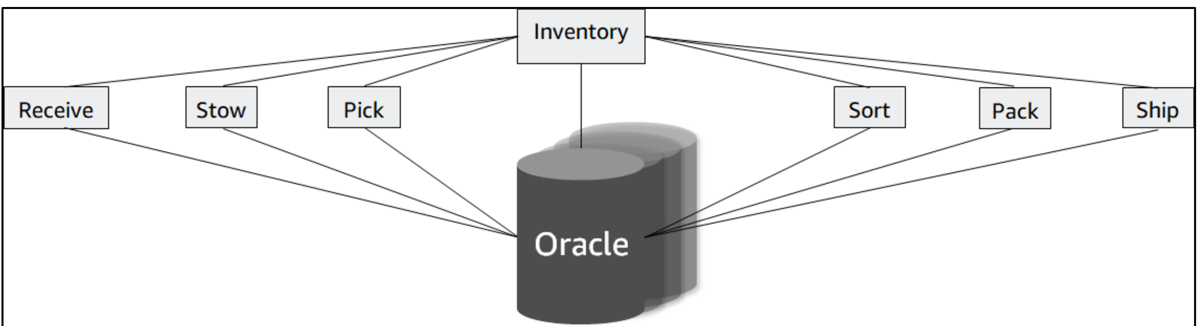
Each of these tasks is supported by services that use static schemas and lookups that use multiple keys across tables containing information such as customer addresses, product codes, product descriptions, item location and seller information. Since inserts and updates to these databases have to adhere to referential integrity constraints, the team decided to migrate the persistence layer to relational database services on AWS.

Ease of scaling and feature parity

As Inventory Management Services deal with very high throughputs, Amazon Aurora for PostgreSQL was an optimal choice. It delivers high performance and availability with up to fifteen low-latency read replicas. PostgreSQL also has close feature parity with Oracle.

Automated administration

Amazon Aurora is managed by the Amazon RDS service that automates administrative tasks including OS patching and software upgrades. This frees up time for the engineers and allows them to focus on schema optimization and service performance improvement.



Fulfillment center activities supported by the Inventory Management Services

Before starting the migration from Oracle to Amazon Aurora, the team decided to re-platform the services rather than rearchitect them. Re-platforming accelerated the migration by preserving the existing architecture while minimizing service disruptions. It also allowed for over 200 external services their dependencies easily.

Migration strategy and challenges

The migration to Aurora was performed in three phases—preparation phase, migration phase, and post-migration phase.

Preparation phase

In the preparation phase, the goal was to arrive at a robust, scalable architecture for the services. This involved making several operational and design decisions. Following recommended best practices, separate production and non-production accounts on AWS were defined to ensure secure and reliable deployment. One early architectural decision made was to extensively use read replicas for horizontal scaling of read workloads. Aurora offers fifteen near real-time read replicas while a central node manages all writes. The ability to scale read workloads provides a substantial tool to

horizontally scale. As security is of upmost priority, the decision was to encrypt all data at rest and in-transit which Aurora provides. Aurora uses SSL (AES-256) to secure connections between the database instances and the application layer. Additionally, all data is encrypted on disk using keys and key management through the [AWS Key Management Service] (AWS KMS). Encryption at rest via KMS also ensures that all data on Aurora instances are encrypted, as are its automated backups, snapshots, and replicas in the same cluster. The team also researched differences between Oracle and PostgreSQL in the preparation phase.

During preparation, an important difference to note is how Oracle and PostgreSQL treat time zones differently. Oracle uses the server’s time zone when logging transactions, whereas PostgreSQL depends on the client’s time zone. Proper preparation to analyze and configure table column types was performed prior to data migration. Another difference between Oracle and PostgreSQL 9.6 is different partitioning strategies and their implementations. In versions of PostgreSQL earlier than 10.x, table partitioning was only possible via inheritance. To partition tables using table inheritance, users have to create a parent table and then create child tables for each partition by explicitly defining the partition rules and constraints.

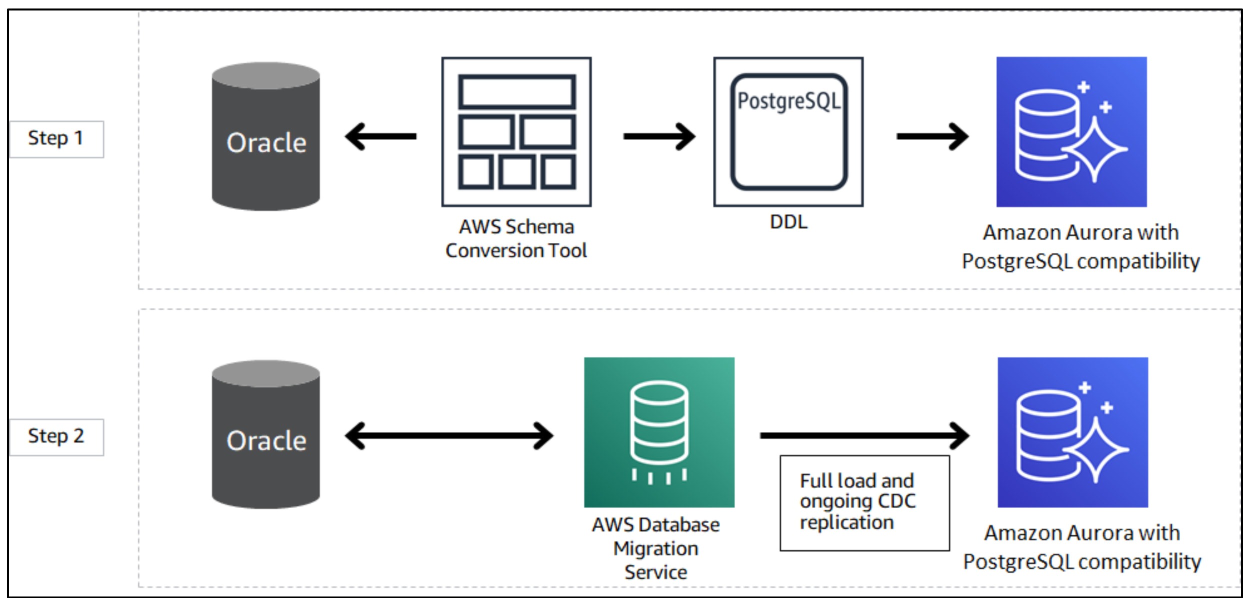
In PostgreSQL 10.x, declarative partitioning was launched which is easier to implement and supports RANGE and LIST partition types. Another difference to consider is the default collation methods used by Oracle and PostgreSQL. The collation specifies the sort order and character classification behavior of data per-column, or even per-operation. This difference between the collation methods could change the sort ordering of character columns. The team planned for additional code testing to ensure that the different collation methods yielded the same results. After identifying a few important differences, the team started performing dependency analysis to inventory the artefacts such as tables, views and sequences, cross-database materialized views and ETL feeds by querying historical view, periodically sampling active views and monitoring logon information.

Migration phase

After the preparation phase, the team established three criteria for a successful migration –automation to the maximum extent possible, no operational impact on the fulfillment centers, and superior key performance metrics of all services same or better than the baseline on Oracle. The first goal of the migration phase was achieved by automating a range of activities including the input of database information of the fleet-wide metadata, creating Reserved Instances, creating databases, onboarding ETL feeds, creating scheduled jobs, integrating Amazon CloudWatch metrics, enabling

database monitoring, creating application schemas, and initiating AWS DMS tasks for migration. The automation of the data migration was achieved through full load and ongoing replication by leveraging available continuous data validation and replication monitoring with alarms. AWS Schema Conversion Tool (AWS SCT) was used to convert the schemas from Oracle to PostgreSQL. Subsequently DMS performed a full load and ongoing Change Data Capture (CDC) replication to move real-time transactional data. During the data transfer phase, the team-initiated Amazon Aurora instances based on current throughput and compute requirements. During the migration of each fulfillment center, services were moved over to the new Amazon Aurora instances during a short downtime where the read and write services were pointed to these new instances.

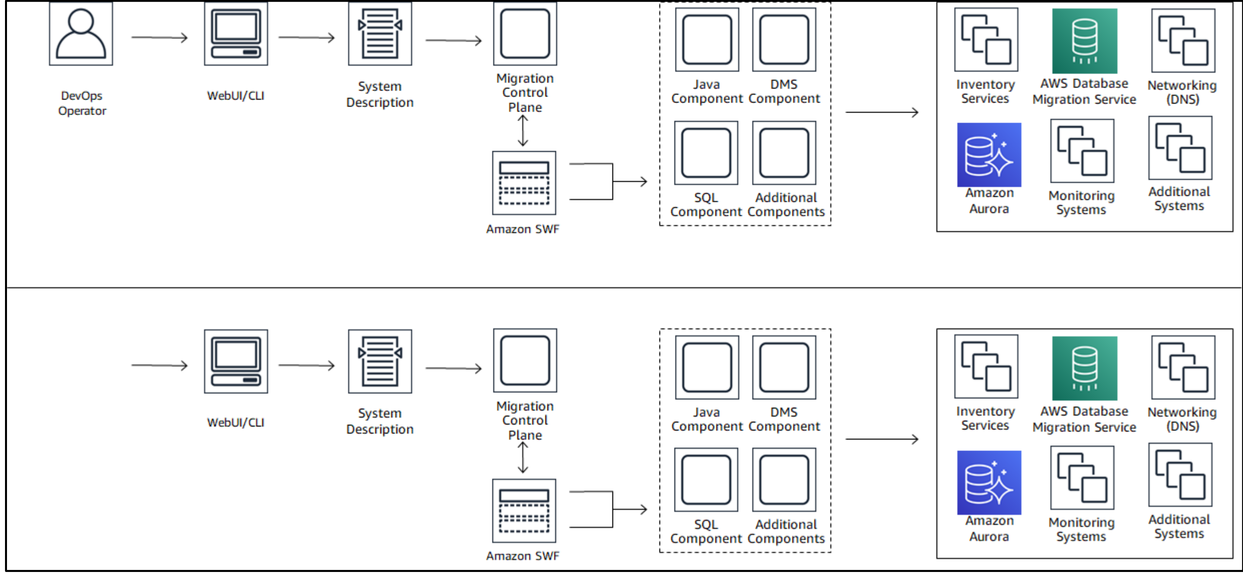
There were many learnings during the migration phases. First, AWS DMS expects all source tables to have a primary or a unique key for data validation. Additionally, a primary key is mandatory while migrating Large Objects (LOBs). LOBs are a set of datatypes designed to hold large amounts of data. DMS uses two methods to balance performance and convenience when migrating LOBs—Limited LOB mode and Full LOB mode. Limited LOB mode migrates all LOB values up to a user-specified size limit. In the Full LOB mode, DMS migrates all LOB data in your tables regardless of size. Full LOB mode provides the convenience of moving all LOB data in your tables but the process can have a significant impact on performance of the migration. The business unit chose the appropriate LOB migration mode depending the LOB size in the source tables with a strong preference for limited LOB mode.



Steps in the migration of schemas using AWS SCT and AWS DMS

```
select
 'select (max(length(' || COLUMN_NAME || '))/(1024)) as "Size in KB"
from
 ' || owner || '.' || TABLE_NAME ||';' "maxlobsizeqry"
 from dba_tab_cols
 where owner='<schema_name>'
 and data_type in ('CLOB','BLOB','LOB');
```

Code snippet to identify max lob size in a table



Workflow of the migration of Inventory Management Services

Ongoing migrations or replications of high-throughput transaction systems using AWS DMS can, at times, consume large amounts of CPU and memory. The R4 instance types can be a good choice for these situations. Also, when performing migration of this type, a large instance size and increasing the `maxFileSize` value can significantly increase throughput. The `maxFileSize` parameter specifies the maximum size (in KB) of any CSV file used to transfer data to PostgreSQL. The team observed that setting `maxFileSize` to 1,048,576KB (1.1 GB) significantly improved migration speed. Since version 2.x, AWS DMS has been able to increase this parameter to 30 GB. The R4 instance comes with four times the memory of the comparable C4 instance (c4.4xlarge). Thus, it completes some of the smaller tables faster and lets the bigger table use the extra memory available to complete the migration.

Post-migration phase

After the move to the Aurora databases was complete, the business unit began the next phase of the project—post migration. Monitoring the health of the database becomes

paramount in this phase and AWS offers many tools to help. One important activity that must occur in PostgreSQL is vacuuming. In PostgreSQL, whenever a row is updated, a new version of the row, known as a tuple, is created and inserted into the table. The old version of the row, referred to as a dead tuple, is not physically removed but is marked as invisible for future transactions. Because every row can have multiple different versions, PostgreSQL stores visibility information inside tuples to help determine whether it is visible to a transaction or query based on its isolation level. If a dead tuple is not visible to any transaction, the vacuum process can remove it by marking its space as available for future reuse. Dead tuples not only decrease space utilization, but they can also lead to database performance issues. When a table has a large number of dead tuples, its size grows much more than it actually needs. A maximum of two billion available Transaction IDs are available and if not cleaned up before this limit, a lengthy single-threaded auto-vacuum process must run which impacts database availability. Aurora PostgreSQL sets auto-vacuum settings according to instance size by default, but one size does not always fit all different workloads, so it is important to ensure auto-vacuum is working properly as expected. AWS provides the CloudWatch metric, `MaximumUsedTransactionIDs`, to monitor and alarm if the number of used Transaction IDs exceeds set thresholds.

Another parameter to consider which is highly dependent on workload is `fillfactor`. The `fillfactor` for a table is set as a percentage between 10 and 100 with 100 (complete packing) being the default value. When a smaller `fillfactor` is specified, INSERT operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables, a smaller `fillfactor` is appropriate.

Monitoring other aspects of the database health is very important. The AFT business unit relied on monitoring and alerting via the Amazon Aurora Amazon CloudWatch metrics, events, and alarms provided to ensure the health of the DB cluster. Amazon CloudWatch provides metrics for the database instance as well as the database itself, such as CPU utilization, database connections, disk queue depth, replica lag, failover, and dozens of others to provide a complete monitoring and alerting solution which can be accessed via Amazon RDS in the AWS Management Console, API, or AWS Command Line Interface (AWS CLI).

One additional tool to note is Amazon RDS Performance Insights which is a database performance tuning and monitoring feature that helps you quickly assess the load on

your database, and determine when and where to act. It is a tool available in the AWS Management Console that allows experts and non-experts to detect performance problems with an easy-to-understand dashboard that visualizes database load.

Benefits

The elastic capacity of preconfigured database hosts on AWS eliminated much of the administrative overhead required to scale the system. The transformation has been significant. On Oracle, a simple change such as scaling from a medium to a large database instance required planning for provisioning hardware, building and configuring primary and standby databases, and managing failover during transitions, which could take a full day for each instance. The business unit also used specialized hardware that had to be ordered months in advance. After migrating to Amazon Aurora, provisioning additional capacity is achieved through a few simple mouse clicks or API calls reducing the scaling effort by as much as 95%.

High availability is another key benefit of Amazon Aurora as reprovisioning happens automatically in just minutes, ensuring data is always fully protected. With the performance of Amazon Aurora, the business unit is no longer limited by the input/output operations the database instance can handle. Migrating Inventory Management Services to Amazon Aurora resulted in a range of benefits—the most important of which is dependable fulfillment for customers.

Migrating to Amazon Aurora – buyer fraud detection

Overview

Amazon retail websites operate a set of services called Transaction Risk Management Services (TRMS) to protect brands, sellers, and consumers from transaction fraud by actively detecting and preventing it. One set of services called Buyer Fraud Services protect Amazon from customers who engage in fraudulent activities including claiming false refunds for damaged goods, purchasing goods via unauthorized payment mechanisms such as stolen credit cards, or falsely claiming theft of received orders. These buyers damage the reputation of brands, impact the ability of sellers to be profitable and raise the transaction costs for Amazon retail consumers. The Buyer Fraud Service applies machine learning algorithms over real-time and historical data to detect and prevent fraudulent activity. It also relies on information provided by adjacent services like the payments and orders to gather real-time information and block potentially fraudulent buyers. The service operates in four regions globally. The two largest regions—US and the EU—run three copies of the service to distribute load.

Each copy of the service relied on an Oracle instance that was six terabytes in size. The busiest service instances handle up to 225 transactions per second (TPS) peak traffic and are expected to handle a regional peak throughput that are significantly higher for events like Prime Day.

Challenges of operating on oracle

The Buyer Fraud Service team faced three challenges operating its services using on-premises Oracle databases.

Complex error-prone database administration

The Buyer Fraud Service business unit shared an Oracle cluster of more than one hundred databases with other fraud detection services at Amazon. This shared pool of databases required three database engineers to perform database administration activities including OS patching, maintenance, and upgrades. Each of these activities had the potential to cause database downtime. Moving to Amazon Aurora abstracted the management of the physical hardware away from engineers and minimized the responsibility for OS patching, maintenance, and upgrades with minimal intervention and impact.

Poor latency

The next challenge was that the Buyer Fraud Service frequently experienced latency issues at peak loads on the provisioned hardware. To maintain performance at scale, Oracle databases were horizontally partitioned. As application code required new database shards to handle the additional throughput, each shard added incremental workload on the infrastructure business unit in terms of backups, patching, and database performance monitoring. The design and implementation of shards was a complex, multi-year engineering exercise. The elastic capacity of preconfigured database hosts on Amazon Aurora eliminated some of the administrative overhead to scale. Amazon Aurora has abstracted the hardware away from the engineers, allowing them to focus on optimizing configurations.

Complication hardware provisioning

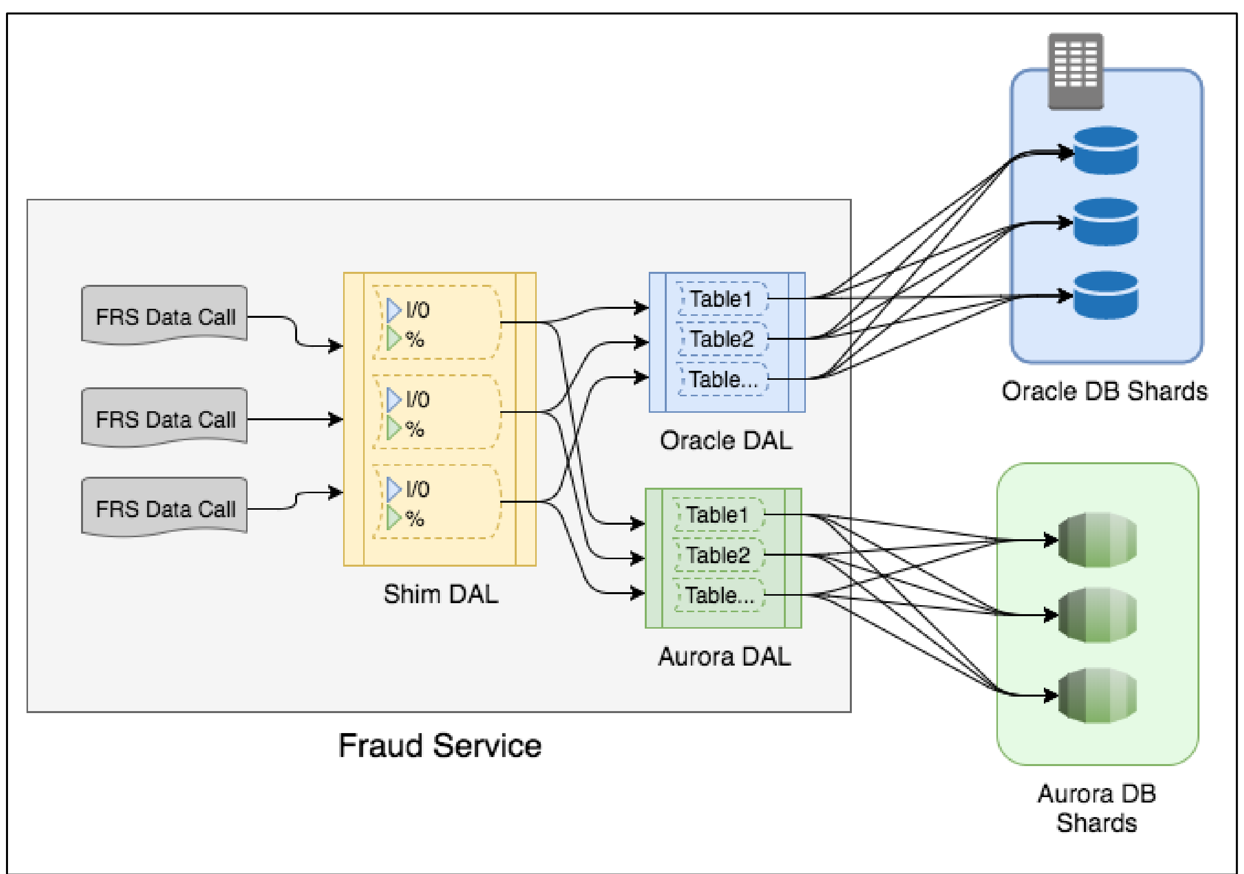
In 2017 the TRMS business unit estimated that the database engineers on each service business unit spent hundreds of hours forecasting and planning database capacity. After capacity planning, the hardware business unit coordinated suppliers, vendors, and Amazon finance business units to purchase the hardware and prepare for installation and testing. After migrating to AWS, provisioning of additional capacity is achieved through a few simple mouse clicks.

Application design and migration strategy

In January 2018, the Buyer Fraud Service business unit decided to migrate its databases from Oracle to Amazon Aurora. The migration exercise took six months and was completed in July 2018. Since Buyer Fraud Service is a critical service, required relational functionality, and is expected to operate at a high availability, Amazon Aurora was the logical choice for a database. The team chose to re-factor the service to accelerate the migration and minimize service disruption. The migration was accomplished in two phases—preparation and execution.

Preparation phase

In the first phase, Amazon Aurora clusters were launched to replicate the existing Oracle databases. Subsequently, the service business unit built a shim layer to perform simultaneous read/write operations to both database engines. The shim layer interpreted service calls and performed simultaneous read/write operations on both Oracle and Amazon Aurora. Once the shim layer was successfully tested to correctly read and write to both databases, the business unit migrated the initial data, and used AWS Database Migration Service (AWS DMS) to establish active replication from Oracle to Aurora. The business unit then activated the shim layer to take over reading and writing to both databases. Once the migration was complete, AWS DMS was used to perform a row-by-row validation and a sum count to ensure that the replication was accurate.



Dual write mode of the Buyer Fraud Service using SHIM layer

Execution phase

In the second phase, Buyer Fraud Service began load testing the Amazon Aurora databases to evaluate read/write latencies and simulate peak throughput events such as Prime Day. Results from these load tests indicated that Amazon Aurora could handle twice the throughput of the legacy infrastructure. Multiple other criteria including average latency, P99 latency, read/write success and failure rates, change data comparison, and count of operations were also evaluated to ensure accurate replication. Standard database utilization metrics such as CPU usage and free memory were also closely monitored and minor tuning was performed to optimize performance. Finally, the Oracle databases were decommissioned after testing was complete.

Benefits

The benefits of migrating Buyer Fraud Service from Oracle to Aurora include performance, scalability, availability, hardware management, cloud-based automation, and cost. AWS manages patching, maintenance, backups, and upgrades allowing

engineers to improve application performance. When operating on Oracle, the team needed three database engineers to keep Oracle updated, and perform activities like database repartitioning and index tuning. After migrating to Aurora, the database administration overhead had dropped by more than half.

The migration has also lowered the cost of delivering the same performance as before. Load tests of the new service at 900 transactions per second (TPS) has shown that Amazon Aurora can scale with minimal CPU usage. In similar loads, the historical Oracle database faced performance issues and led to the deterioration of service throughput. The improved performance of the service on Amazon Aurora has allowed the business unit to handle high throughput events like Prime Day with ease.

In terms of scaling, Buyer Fraud service was able to scale its largest workloads, support strict latency requirements with no impact to snapshot backups. Scaling up or down only takes a few minutes and services experience seamless horizontal scaling.

In terms of availability, the business units saw faster rollovers and most hardware failures were mitigated in minutes with the service going up and running. Hardware management has gotten exponentially easier with new hardware being commissioned in minutes instead of months. This also makes it better for DBA resources to manage. Lastly, no more licenses and open-source support translates to lower costs.

Organization-wide benefits

Amazon saw multiple benefits from this migration including improved service performance, lower operating costs, reduced database administration overhead and ease of scaling. Services that migrated to Amazon DynamoDB, saw significant performance improvements such as a 40% drop in 99th percentile latency. The shutdown of data centers eliminated hardware, freed up real estate, eliminated power and cooling costs and reduced physical security threats. Amazon RDS has allowed engineers to spend time on improving service performance by abstracting away the management of physical hardware and performing maintenance activities including OS patching, database maintenance and software upgrades. Additionally, the elastic capacity of preconfigured database hosts on AWS has eliminated administrative overhead to scale by allowing for capacity provisioning through simple mouse clicks thereby eliminating arduous capacity planning, hardware procurement and installation activities. Eliminating these undifferentiated activities has allowed Amazon to focus on improving customer experience and service quality.

Post-migration operating model

After migrating to AWS, the Amazon consumer facing business segments instituted new mechanisms, processes, and tools to ensure ongoing success with the new model. The section that follows discusses key changes in the operating model for service teams and its benefits.

Distributed ownership of databases

Moving to AWS has decentralized the concept of database ownership. In the past years, most teams in Amazon relied on shared database infrastructure that was purchased, installed, operated and maintained by a shared team of database engineers. This team of engineers estimated hardware capacity by inquiring about requirements from each service team and aggregating the demand. In an ideal world, they should also have allocated the cost of hardware, software, and database administration amongst all teams but this exercise was so cumbersome and complicated that it was hard to execute. The migration to AWS transformed this operating model completely to one focused on distributed ownership. Individual teams now control every aspect of their infrastructure including capacity provisioning, forecasting and cost allocation. This eliminated the need to pool capacity requests and perform complex cost allocations. Each team also had the option to launch Reserved or On-Demand Instances to optimize costs based on the nature of demand. For DynamoDB, it was even easier as the capacity management can be automatic within a window you set. Typically, business segments experiencing unpredictable or highly cyclical loads picked a higher proportion of On-Demand instances to scale with peak capacity. The CoE developed heuristics to identify the optimal ratio of On-Demand to Reserved Instances based on service growth, cyclicality, and price discounts. The CoE also built tools to monitor the usage of hundreds of AWS fleets centrally.

The teams were able to focus on innovation on behalf of customers instead of the burden of specialized database management.

Career growth

The migration presented an excellent opportunity to advance the career paths of scores of database engineers. These engineers who exclusively managed Oracle databases in data centers were offered new avenues of growth and development in the rapidly growing field of cloud services, NoSQL databases, and open-source databases. As basic database administrative tasks such as patching, upgrades, and backups were

automated, these engineers could now leverage their skills and expertise in improving the service tier by optimizing query plans, monitoring performance, and testing new services. These learning and growth opportunities allowed them to advance their careers, refresh their skills, and directly improve customer experience.

Contributors

Contributors to this document include:

- Venkata Akella, Sr. Product Manager, Database Freedom
- John Winford, Principal Product Manager, Database Freedom
- Chris Brack, Software Development Manager, Financial Ledger and Accounting Services
- Phani Bhadimpati, Sr. Manager, Financial Ledger and Accounting Services
- Brent Bigonger, Sr. Database Engineer, Operations – Tech Services
- Gowri Balasubramaniam, Principal Database Solution Architect, AWS Solution Architecture
- Josh Gage, Sr. Software Development Engineer, Transaction Risk Management
- Kirk Kirkconnell, Sr. Technologist, Amazon DynamoDB
- Michael Stearns, Product Marketing Manager, AWS Databases Services

Document revisions

Date	Description
March 8, 2021	Reviewed for technical accuracy
November 2019	First publication