

Introduction

Astra DB for Cloud Native Applications

The Astra DB logo is located on a solid orange rectangular background. The word "Astra" is written in a bold, white, sans-serif font, with the letter "A" having a small triangle pointing upwards. To its right, the word "DB" is written in a smaller, white, sans-serif font.

Astra DB

**Astra DB is managed Cassandra
for cloud native applications**



Outline

- Use Cases
- Features
- Fundamentals
- Getting Started



Astra DB Use Cases

Astra DM in the real world

AstraDB Real World Successes

Profile

Pain Points

AstraDB Outcomes



Industry leading esports org
Services ~11 million users
Event registration/authentication
On-premises C* deployment

Cloud Migration
Performance
Availability

Managed Service
High Performance and scalability
Pay as You Go

Lead classifieds seller in Norway
50 millions visits per month
Personalization for ads
On-premises C* deployment

Cloud Migration
Self managed operations
Availability

Managed Service
Performance and Availability
Cloud Agnostic (GCP)

FinTech leader based in Singapore
> S\$1.5 billion in assets by 2021
Event sourcing microservices arch
On-premises C* deployment

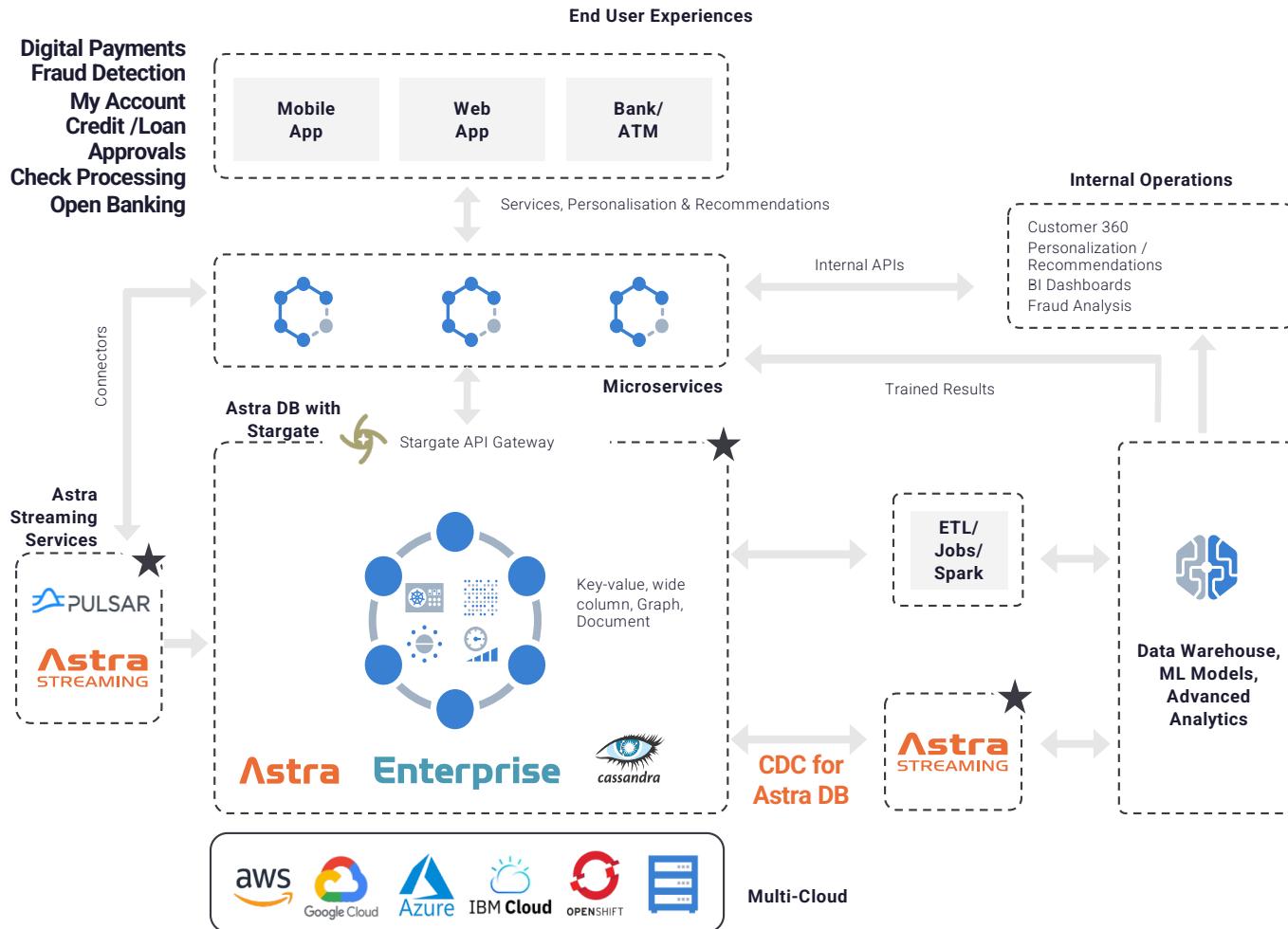
Cloud Migration
Self managed operations
Enterprise grade features not
found in OSS C*

Cloud Agnostic
Reduced Operations
Developer Velocity

Astra DB Use Cases



Astra DB In Financial Services



Astra DB for Financial Services

Data Sources include payment, investment and credit/loan processing experiences from Mobile, Web

DataStax provides fully managed
Astra DB: Multi Cloud, Globally Scalable, with Zero Downtime and Built-in Security

Financial applications are implemented as microservices for rapid, frequent and reliable delivery

Microservices can use **REST API, GraphQL or Document API available in Stargate.**

Astra Streaming service ensures real time ingestion of event data and real-time data pipelines with Data Warehouses or Machine Learning to provide Personalized Services and real-time trading and investment analysis

Endowus Invests for High Growth with DataStax Astra DB

Challenge:

- Singapore-based financial technology leader Endowus sought to put customers first and understand them more holistically through data. Endowus built a cloud-native platform based on an event sourcing pattern with a microservices architecture (MSA) in which all apps run on containers.
- Initially the company hosted its own Cassandra environment, but as Endowus' business grew, the IT team looked for additional support.

Solution:

- **Astra DB** to reduce operational overhead.
- Provides Endowus enterprise-grade features for its Cassandra environment, in addition to the efficiency and flexibility of a cloud-agnostic managed database service
- Solution leverages DataStax contributions to the Stargate Data API Gateway, which eliminates drivers and the need for Endowus' developers to learn Cassandra Query Language (CQL).

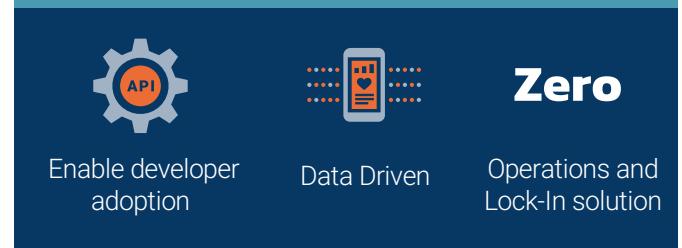
Results:

- Endowus is realizing the benefits of faster data to enable its developers to provide better investment customer experiences.
- With Astra DB and the synergies of the APIs with Endowus' microservices architecture, they can analyze results and iterate quickly to adjust areas that need improvement and more effectively capitalize on success.

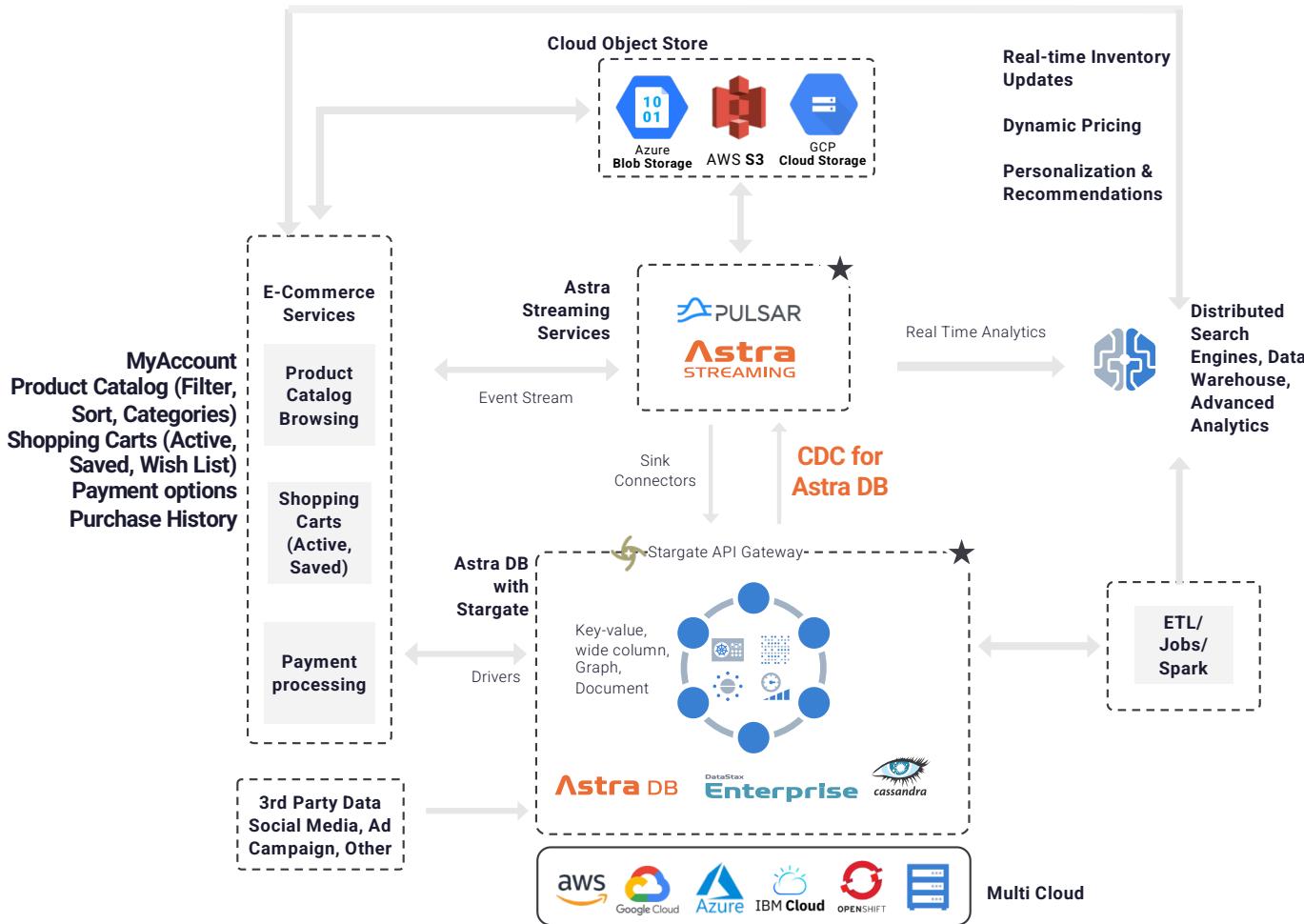
Industry
Finance
EndowUS
Customer Since
2021

"Since we've expanded quickly, we appreciate being able to rely on DataStax and Astra DB to alleviate the administrative burden of our database. Also, we value the resiliency of DataStax and Cassandra. We have to ensure very high availability for our customers, as mandated by our regulator."

— Joo Lee, CTO, Endowus



Astra DB for E-commerce & Customer 360



Astra DB for E-Commerce & Customer 360

Data sources generate data via MyAccount, Product Catalog, Shopping Cart, Personalized Offers.

DataStax **Astra DB** supports **Multi Cloud, Multi Model** workloads with **Global Scale & Zero Downtime..**

Microservices use historical and real-time data to accomplish “360-degree view” to increase customer experience, retention and Loyalty.

API Centric Approach to push data to DataStax with **Stargate** with **REST API, Document API, GraphQL or gRPC**.

Astra Streaming service ensures real time ingestion of event data such as clickstreams and data-pipelines with BI and Data warehouse integrations.

“Craigslist of Norway” Manages Services of All Kinds, Except Their C* Clusters, Thanks to Astra DB

Challenge:

- Finn.no wanted to move its IT infrastructure to the cloud in order to focus on strategic priorities instead of managing infrastructure
- Needed to migrate more than 800 applications and 145 database instances to Google Cloud Platform, including mission-critical personalization engine running on Cassandra

Solution:

- DataStax Astra DB

Results:

- Quickly delivering personalized advertisements to users, based on machine learning models, **in real-time**
- More teams within the parent company, Schibsted Group, can now use personalization and data science in their day-to-day activities
- Finn.no has the **support and expertise** needed to manage its Cassandra clusters with **speed and resiliency** to provide fast, valuable recommendations to users



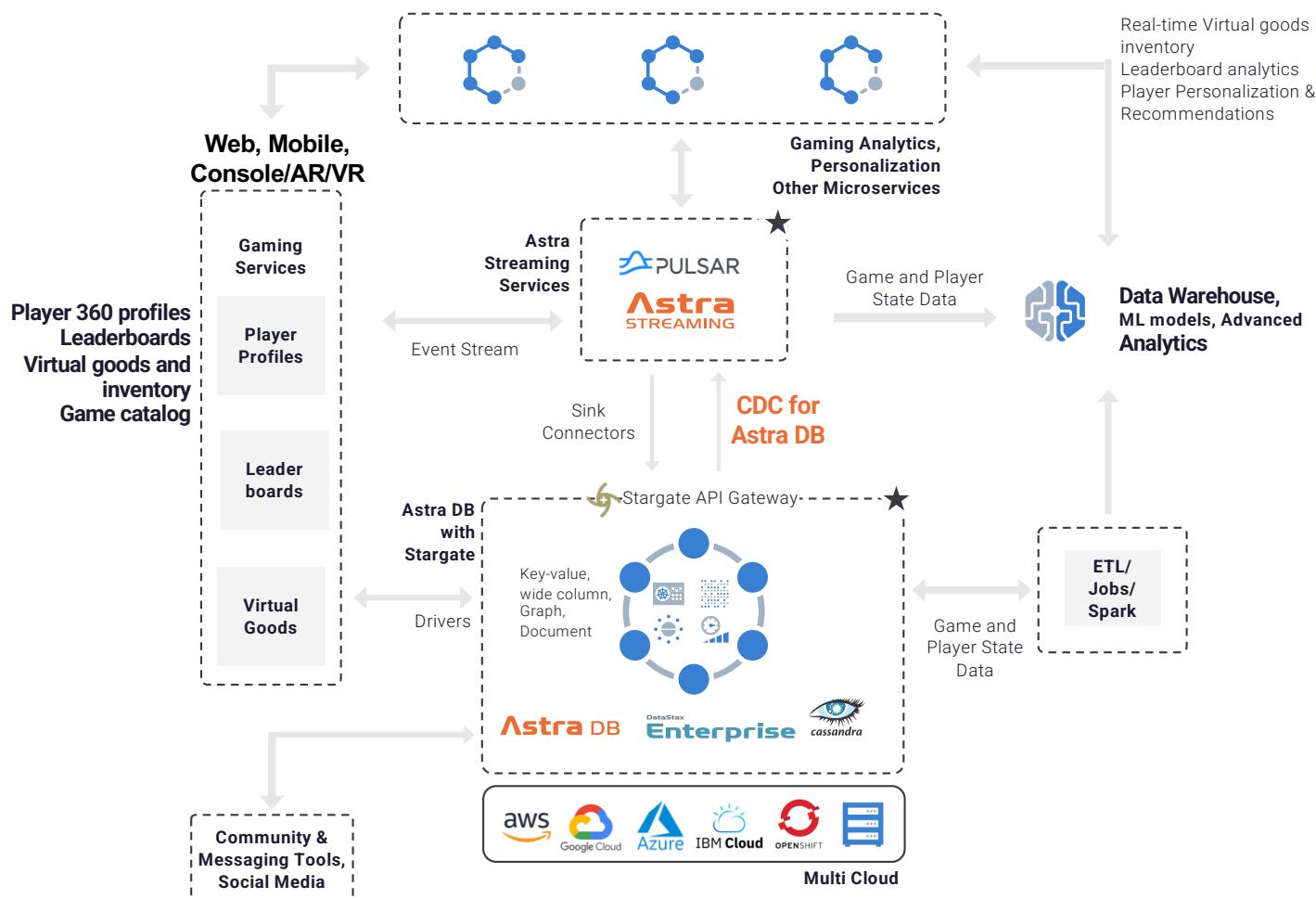
Industry
E-commerce
Customer Since
2020

“We ran on the open source version of Cassandra in the past, but we wanted to get support for running our instances in the future. DataStax Astra represented a way for us to move to a managed Cassandra service that would be part of our company’s overall cloud migration, but it also provided us with more support and expertise over time. For us, Astra was a perfect fit during our move to the cloud.”

— Benjamin Weina Lager, Technical Domain Expert- Data Intelligence, Finn.no

50M Monthly website visits	 Millions of personalized recommendations	Zero Operational overhead with Astra DB
--------------------------------------	---	---

Astra DB for Gaming



Astra DB for Gaming

Data Sources include Player Profiles, virtual goods & inventory catalogs from single/multi-player mobile/web/console games.

DataStax provides fully managed **Astra DB: Multi Cloud, Globally Scalable, with Zero Downtime and Built-in Security**

Gaming services require Astra DB's low-latency high-volume access to the data to avoid any lag/delays or jitter.

Microservices managing player profiles and catalogs can use **REST API, GraphQL or Document API available in Stargate**.

Astra Streaming service ensures real time ingestion of event data.

ETL with Spark for Data Warehouse for BI and leverage Machine Learning to provide Personalized Game offers, leaderboard analytics, real-time inventory and availability

ESL Gaming Levels Up the Fan Experience with Real-Time Data

Challenge:

- As the world's largest esports and gaming lifestyle company, ESL Gaming, based in Germany, needed high performance for gaming events that attract 46 million unique viewers online
- Event registration and authentication required high throughput
- Company wanted to migrate business-critical applications to the cloud, including its Cassandra database, without sacrificing features such as Time To Live (TTL), which governs how long user data exists before it is deleted, and Materialized Views, which speed up reading data at scale

Solution:

- **DataStax Astra DB**

Impact:

- **Smooth cloud migration** with support from DataStax
- **Managed Cassandra environment** and access to expert consulting help ESL's Site Reliability Engineering (SRE) team focus on other business priorities instead
- **High performance** for fast processing of event registrations and registrant authentications
- Leveraging Storage Attached Indexing (SAI), which comes with Astra DB, for use with ESL's materialized views, providing **better scalability and reliability**, especially for event traffic peaks
- **Cost savings** with the Astra DB 'pay as you grow' model where ESL buys only what they need, rather than paying based on estimates of data growth



Industry
Entertainment
Customer Since
2021

"The sun never sets when it comes to gaming. Our services have to be available and they have to perform, so our approach to data has to be fast and reliable at a massive scale...Astra DB means that we have the best possible managed service for our use cases, so that we are able to maintain availability, and focus on providing the best possible experience for players."

— Ben Burns, Vice President, Technology, ESL Gaming



Astra DB cost savings and flexibility

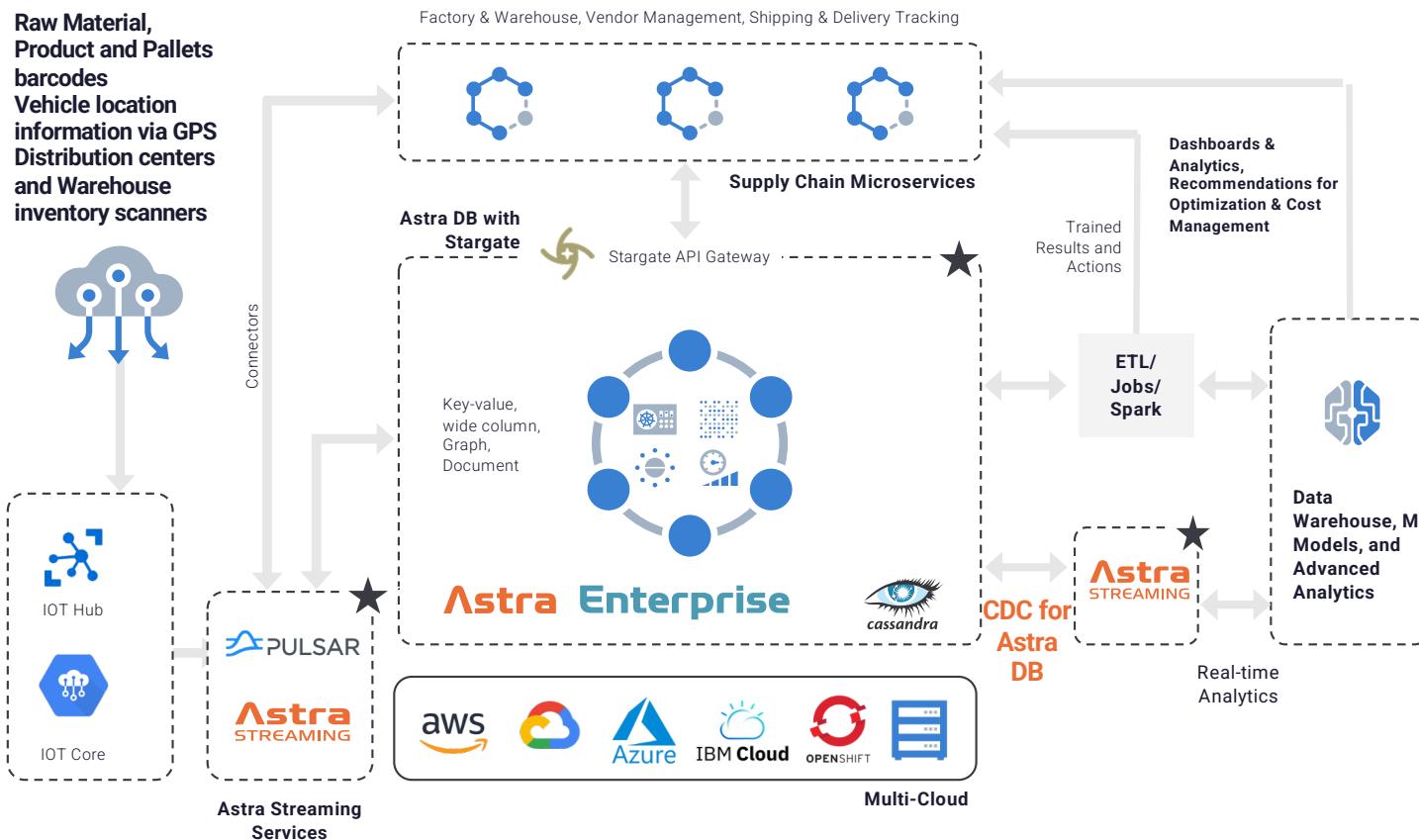


High reliability with scalability



Fast throughput

Astra DB for IoT, Supply Chain, Logistics



Astra DB for IoT, Supply Chain, Logistics

Data sources include IoT devices such as Alarms, Sensors, GPS trackers etc. IoT Message Broker gets data via MQTT a lightweight Messaging protocol with low latency.

DataStax provides fully managed **Astra DB: Multi Cloud, Globally Scalable, with Zero Downtime and Built-in Security**

Microservices handling Factory, Warehouse, Inventory and Shipping data can use **Multi-model Astra DB APIs** including **REST API, GraphQL or Document API available via Stargate**.

Astra Streaming service ensures real time ingestion of event data and data analytics for cost and process optimizations and event dashboards

Ankeri Moves to Astra DB for Global Supply Chain Ship Data

Challenge:

- Ankeri's leadership team saw that there was a gap in the market around real-time data for the maritime shipping sector
- Ability to bring together data from multiple different ship operating applications in real time
- Scale up to ingest, normalize, and manage data from thousands of commercial vessels
- Provide a trusted source of data to multiple ship owners and operators

Solution:

- Full-stack serverless implementation
- DataStax Astra DB
- Storage Attached Indexing (SAI) - Relational Scale Indexing

Results:

- Ankeri chose DataStax as service provider in part due to high level of attentiveness
- Read-optimized tables enable simple app development, easy on-ramp for relational developers
- Guardrails prevented implementation of anti-patterns and problems at scale
- Storage-Attached Indexing improves performance for large clusters as they grow
- Serverless Apache Cassandra-as-a-Service enables developer teams to scale clusters easily and efficiently
- Astra provides a pay-as-you-go model allowing Ankeri to grow with cost flexibility

What's Next:

- Now piloting with customers with plans for scaling up within months

Industry
Supply Chain
Customer Since
2021



"At Ankeri, we are in a unique position to deliver information on ship performance based on gathering data from multiple vessels and their applications in real time. Astra serverless provides that simple, scalable platform for us to build on, supporting real-time data from thousands of ships, and providing trusted data to both ship owners and operators for them to understand their performance."

—Kristinn Aspelund, CEO at Ankeri



Easy on-ramp for relational devs



Real-time data gathering at scale



Pay-as-you-go for faster innovation with less risk



Features

Core features of Astra DB

AstraDB

- Serverless
- Multi-region
- Massively scalable
- High performance
- Zero downtime
- Zero-ops
- Compatible with Apache Cassandra™
- Any data model
- Any cloud
- Any API

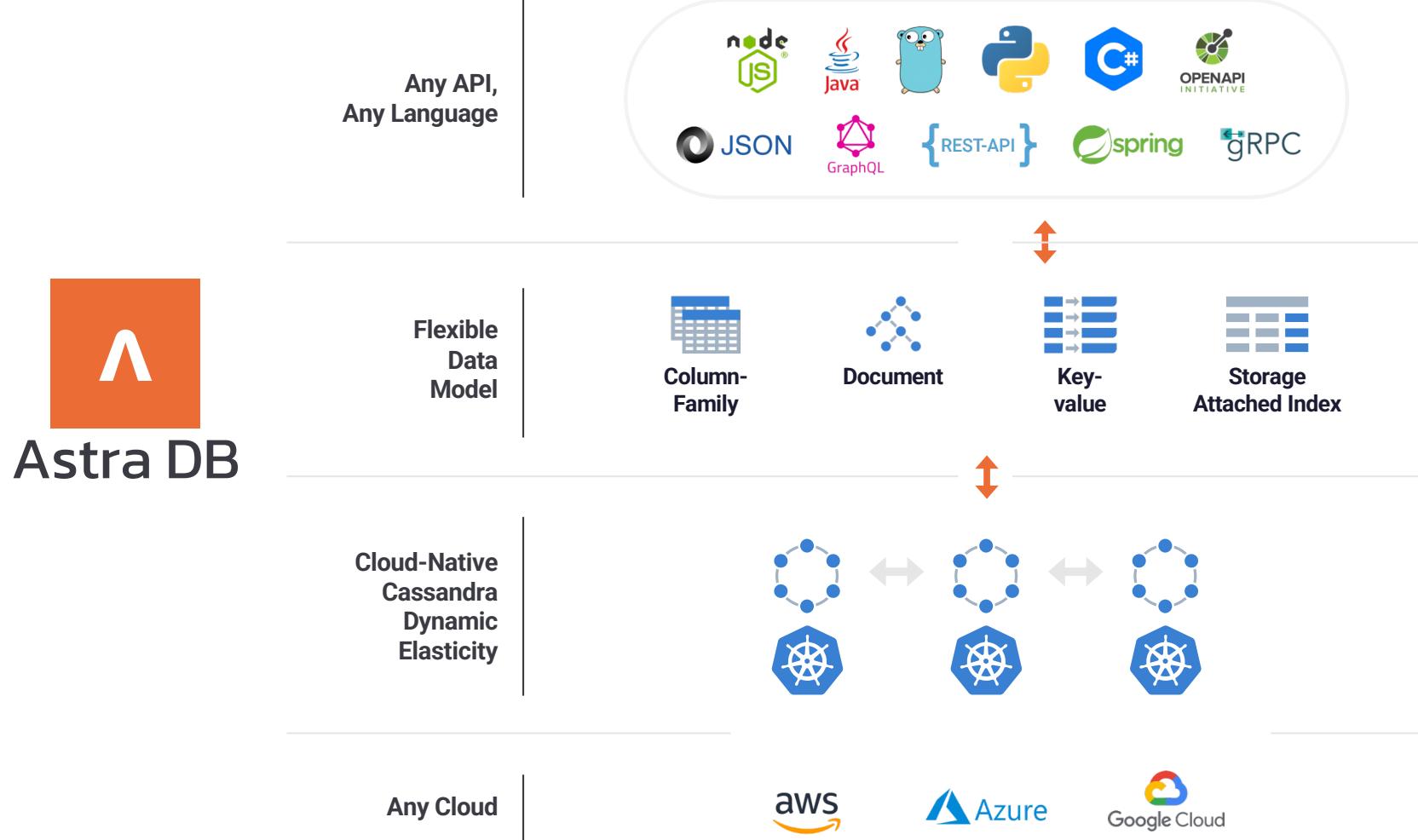


Built on Apache Cassandra™

Apache Cassandra™ was originally released in 2008

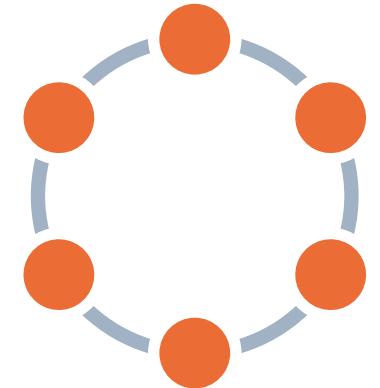
- Distributed
- High performance
- Highly available
- Scalable
- NoSQL

Astra DB is what Cassandra would be if it had been created today



Serverless

- Astra DB is serverless!
- No more
 - Capacity planning
 - Server commissioning/decommissioning
 - Data center configuration



Multi-region

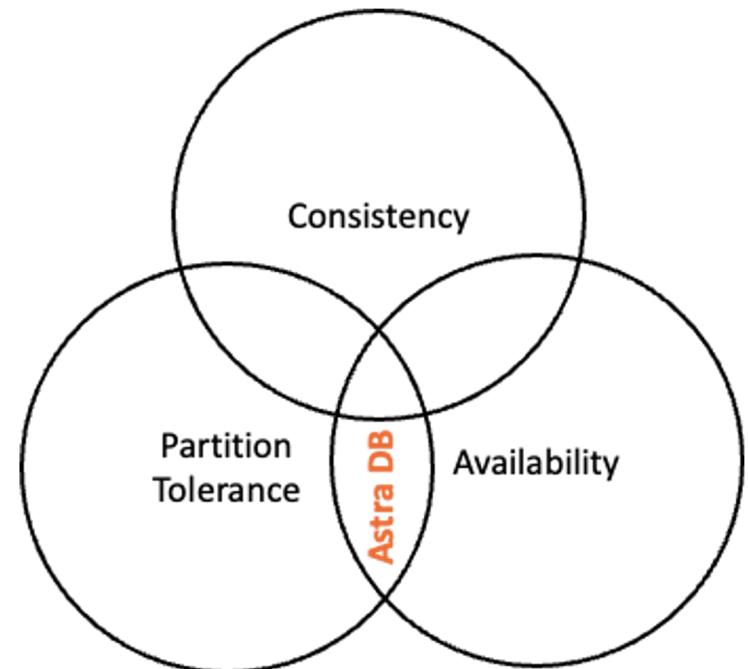
- Deploy across multiple AWS, GCP or Azure regions
- Simple configuration
- Benefits
 - High availability
 - Low latency (clients access data in region close to them)
 - Data sovereignty

High Performance

- Astra DB builds on the performance of Apache Cassandra™
- Optimized for the cloud
- Data modeling best practices for performance
 - Data partitioning
 - De-normalization
 - No ACID transactions
 - Table-per-query pattern

Zero Downtime

- Astra DB is optimized for Availability and Partition Tolerance
 - *Eventual Consistency*
- Default data replication across 3 availability zones



Zero-ops

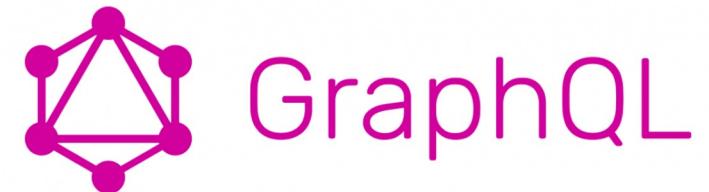
- Astra DB is a *managed service*
- DataStax *auto-magically*
 - Scales elastically
 - Bootstraps new nodes
 - Performs repairs
 - Creates backups

Compatible with Apache Cassandra™

- Astra DB is built on top of Cassandra
- Shares significant parts of the codebase
 - Re-factored for a cloud-native environment
- Core concepts are the same
 - Data modeling
 - CQL
 - Partitioning and indexing

Any Data Model

- Cassandra
- Document (JSON)
- Graph



{JSON}

Any Cloud

- AWS
- GCP
- Azure



Google Cloud



Any API

- Access Astra CB with the API of your choice
 - Driver (Java, C#, Python, Go, etc) and CQL
 - REST
 - gRPC
 - GraphQL
 - Document API (JSON)





Fundamentals

The fundamental concepts behind Cassandra and Astra DB

Distributed

- Cassandra (and Astra DB) are distributed databases
 - Astra DB is *serverless* – there are still nodes (servers) but they are managed for you automatically
- Data distribution
 - All data is stored on multiple nodes
 - No node contains *all* data

NoSQL

- Still think in terms of
 - Tables
 - Rows
 - Columns
- Completely different way of looking at data
 - Non-relational
 - No joins
 - Not de-normalized
 - No ACID transactions
- Even the data modeling methodology is different

Benefits

- Some design decisions may seem *strange* in Astra DB because of three design goals.
 - High performance
 - High availability
 - Global scale

Data Modeling: Astra DB vs Relational Databases

Astra DB	Relational Databases
Model tuned for specific queries	Model tuned for general data access
Queries access single table	Queries access multiple tables
New queries require model change	New queries do not require model change
No ACID Transactions	ACID Transactions
Duplicate data	De-normalized data (no duplication)

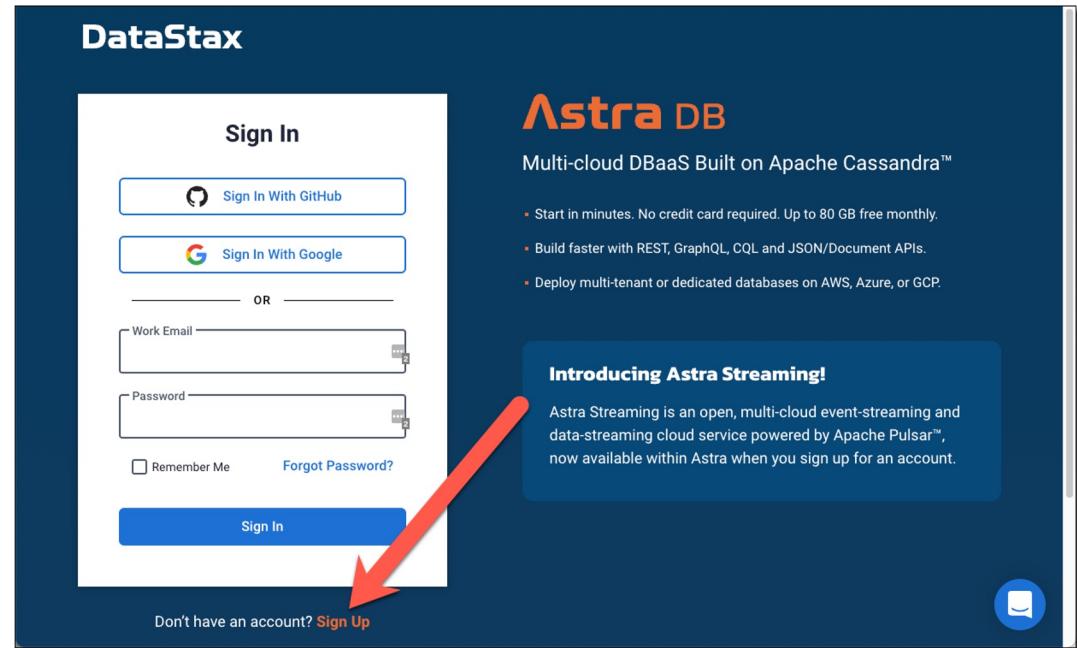


Getting Started

Create an Astra DB account

Connect to Astra

- <https://astra.datastax.com>
- Sign up for a free account



Your Astra account really is free! You will not even have to give DataStax a credit card!

Dashboard

The screenshot shows the Astra DataStax dashboard. On the left, there's a sidebar with navigation links for Dashboard, Databases, Create Database, Streaming, Create Stream, Sample App Gallery, Documentation, Help Center, and Other Resources. The main area features a "Create your First Database" section with a "Create Database" button and a cartoon astronaut illustration. Below this is a "Dashboard" section with a "Free" plan label. Red arrows point from the text "Looking to use dsbulk for bulk uploads or perform load testing? Request a rate limit increase." to the "Free" label and the "Usage" section. The "Usage" section displays current billing period statistics: Read Requests (0), Write Requests (0), Storage Consumed (0.00), and Data Transfer (0.00). A message at the bottom states, "You haven't created a database yet. Once you create a database, it will appear here."

Sample App Gallery

The screenshot shows the DataStax Astra dashboard with the 'Sample App Gallery' section highlighted. The gallery displays several sample applications:

- Netflix Clone using GraphQL and Astra DB**: Last Updated: Feb 11, 2022, 2 hours, Beginner. Companion code for the Java Brains 'code with me' series that reads data from DataStax Astra DB.
- New Code With Me Series!**: Spring Boot Big Data Application Development Building a Goodreads Clone
- BetterReads Spring App**: Last Updated: Oct 5, 2021, 180 minutes, Advanced. Companion code for the Java Brains 'code with me' series that reads data from DataStax Astra DB.
- React Native Todo List, an Astra DB + Netlify Workshop**: Last Updated: Feb 3, 2022, 40 minutes, Intermediate. A workshop where you build a React Native Todo application using DataStax Astra DB.
- Including Search**: Can you build a FULLSTACK APP in 30mins? React
- TikTok**: A screenshot of a TikTok interface showing a user's profile and feed.
- Astra todos**: A screenshot of a mobile application interface titled "Astra todos" showing a list of tasks: Walk my dog, Clean the dishes, Make dinner.

Create a Database

- Specify:
 - Database name
 - Keyspace name
- Select cloud provider (AWS, GCP, Azure)
 - Select region

Not all regions are available for *free* plan use.

Create a Database

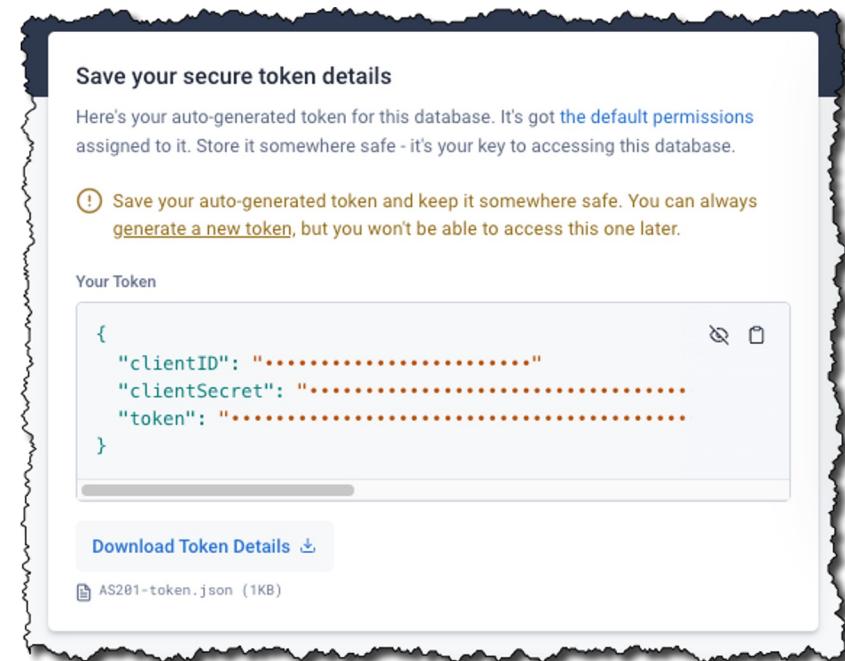
The screenshot shows the Astra interface for creating a database. It has two main sections:

- 1 Enter the Basic Details:** This section contains fields for "Database Name" (AS201) and "Keyspace Name" (class). A note says "Give it a memorable name - this can't be changed later." and "Want to know more? [Read our Docs](#) to learn more about keyspaces."
- 2 Select a Provider and Region:** This section allows selecting a provider (Google Cloud, AWS, Microsoft Azure) and a region. It shows "North America 1 of 6 regions selected" with "Moncks Corner, South Carolina" (us-east1) selected. Other options include "Ashburn, Virginia" (us-east2), "The Dalles, Oregon" (us-west2), "Council Bluffs, Iowa" (northamerica-east1), "Montreal, Quebec" (northamerica-east1), and "US West (Las Vegas)" (us-west2). A red arrow points to the "Select a Region" dropdown.

To the right, there's a sidebar for the "Current Plan" which is "Free". It states: "You're currently on our free plan, which gives you free credits monthly. That recurring credit should be more than sufficient for your development needs, running sample code or apps, building proof-of-concepts, hackathon participation – even running small production workloads." A link "Learn more" is provided. At the bottom right of the sidebar is a "Create Database" button.

Secure Token

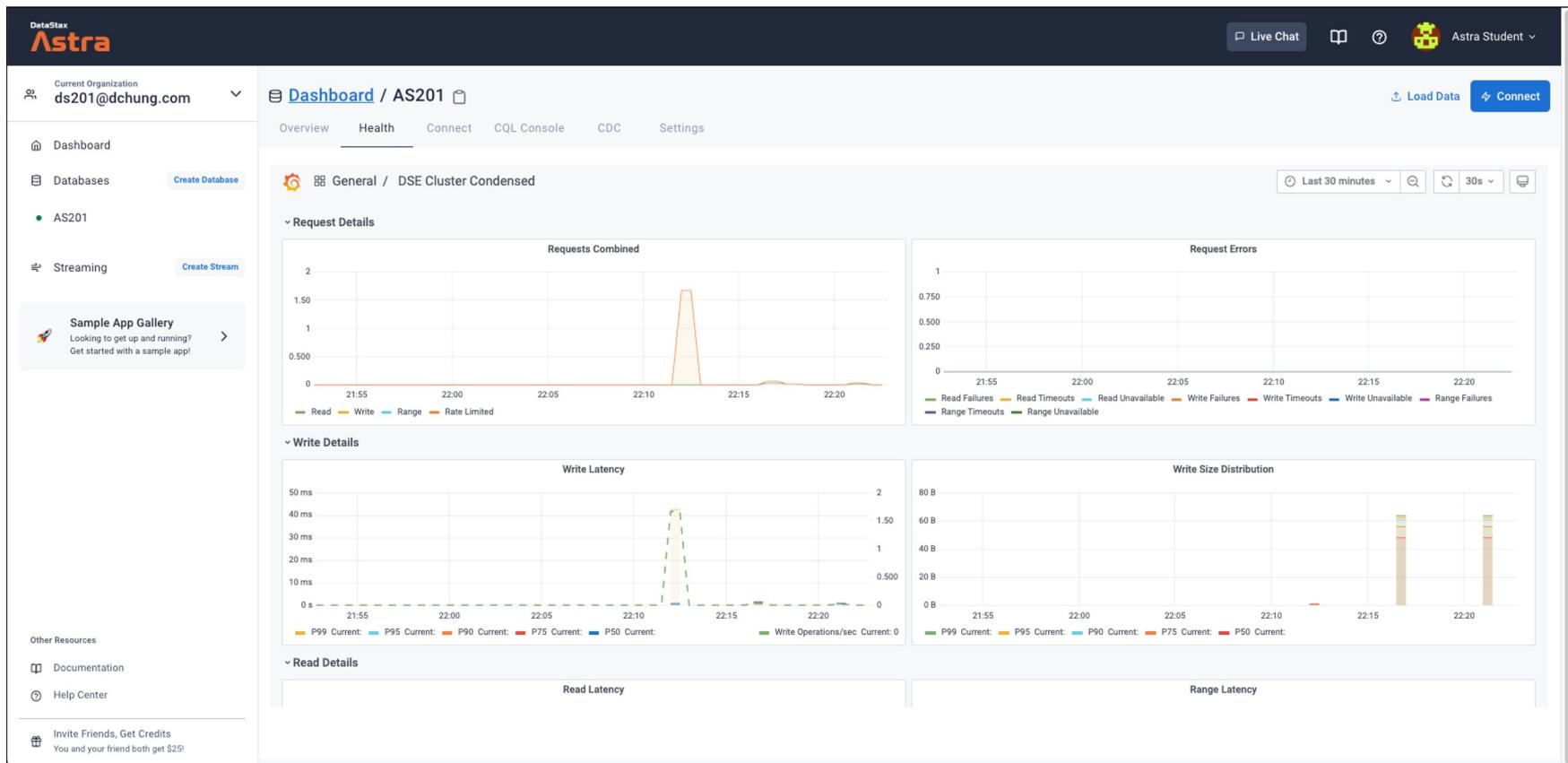
- Remote db access requires a secure token
 - Download and store in a safe place
 - New tokens can be generated later
 - Existing tokens cannot be recovered



Database View

The screenshot shows the Astra Database View dashboard for the organization `ds201@dchung.com`. The main navigation bar includes links for Overview, Health, Connect, CQL Console, CDC, and Settings. The dashboard displays usage metrics for the current billing period for AS201, including 0 Read Requests, 0 Write Requests, 0.00 Storage Consumed, and 3.57 KB Data Transfer. It also lists a single region entry: Google Cloud, North America, us-east1, Moncks Corner, South Carolina, Datacenter ID 6130b58b-56a6-441f-8b45-64f7461f8393-1, and Region Availability Online. The Keyspaces section shows a single keyspace named 'class'. The left sidebar provides links to Dashboard, Databases (with AS201 selected), Streaming, Sample App Gallery (with a note about Pay as you go), Documentation, Help Center, and an Invite Friends, Get Credits link.

Health Tab



Connect

The screenshot shows the Astra Database dashboard for the organization `ds201@dchung.com`. The left sidebar includes sections for Dashboard, Databases (with `AS201` selected), Streaming, and Sample App Gallery. The main content area is titled `Dashboard / AS201` and has tabs for Overview, Health, Connect (which is active), CQL Console, CDC, and Settings. The `Connect` tab displays instructions for connecting to the database using various methods:

- Connect using an API**: Includes links for Document API, GraphQL API, and REST API.
- Using the Document API to connect to your database**: Describes how the Document API allows storing JSON documents in Astra DB without a schema. It includes prerequisites and a code snippet for setting environment variables:

```
export ASTRA_DB_ID=6130b58b-56a6-441f-8b45-64f7461f8393
export ASTRA_DB_REGION=us-east1
export ASTRA_DB_KEYSPACE=class
export ASTRA_DB_APPLICATION_TOKEN=<app_token>
```
- Prerequisites**:
 - An Application Token (create a new one [here](#)) with the appropriate role set (API Admin User is needed for example below).
 - In the command-line interface associated with your development environment, paste the following and replace `<app_token>` with your Application Token:
- Launching Swagger UI**: Visualize and interact with your database's REST API directly from your web browser with Swagger UI: <https://6130b58b-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com/api/rest/swagger-ui/>
- Writing a document**: You can create a document with the HTTP POST verb and it will be automatically assigned a unique id:
- Connect using a gRPC client**: Includes links for JavaScript SDK, Python SDK, Java SDK, and Rust.

CQL Console

The screenshot shows the DataStax Astra web interface. The top navigation bar includes 'Live Chat', a search icon, a help icon, and 'Astra Student' with a dropdown arrow. The main header 'Dashboard / AS201' has a back arrow icon. Below the header are tabs: Overview, Health, Connect, **CQL Console**, CDC, and Settings. A 'Load Data' button and a 'Connect' button are on the right.

The left sidebar shows 'Current Organization: ds201@dchung.com'. It has sections for Dashboard, Databases (with 'AS201' selected), Streaming (with 'Create Stream'), and Sample App Gallery (with a 'Sample App' button). Other Resources include Documentation and Help Center. A 'Invite Friends, Get Credits' section offers \$25 for both users.

The central content area is titled 'Connect to your CQL Console'. It says 'Interact with your database through Cassandra Query Language (CQL). Need help? Check out our [quick reference guide on CQL](#)'. Below this, it shows a CQL prompt: 'Connected as ds201@dchung.com. Connected to cndb at cassandra.ingress:9042. (cqlsh 6.0.0 | Cassandra 4.0.0.6816 | CQL spec 3.4.5 | Native protocol v4) Use HELP for help. token@cqlsh> []'



Hands-on Lab

Lab 01: Create an Astra DB account

- Create an Astra DB account
- Create a database
- Save credentials
- Explore the Astra UI

Tables and Keys

Astra DB for Cloud Native Applications

The Astra DB logo is located on a solid orange rectangular background. The word "Astra" is written in a bold, white, sans-serif font, with a stylized 'A' that has a vertical line extending upwards from the top of the 'a'. To the right of "Astra", the word "DB" is written in a smaller, white, sans-serif font.

Astra DB



Outline

- Simple Table Definition
- Inserts, Updates and Upserts
- Partitions
- Clustering Columns
- Multiple Clustering Columns



Simple Table Definition

Create and query simple tables

Tables

- Astra stores data in tables
- Tables consists of rows
- Rows consist of columns

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Cassandra Query Language – CQL

- Native language of Astra (and Cassandra)
- Syntax similar to SQL (Structured Query Language)

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    make text,
    model text,
    year int
);
```

INSERT

```
INSERT INTO cars(id, make, model, year)
    values(1001, 'Dodge', 'Challenger', 1971);
INSERT INTO cars(id, make, model, year)
    values(1002, 'Ford', 'Mustang', 1968);
INSERT INTO cars(id, make, model, year)
    values(1003, 'Chevy', 'Camaro', 1969);
INSERT INTO cars(id, make, model, year)
    values(1004, 'Dodge', 'Daytona', 1969);
INSERT INTO cars(id, make, model, year)
    values(1005, 'Dodge', 'Challenger', 1972);
INSERT INTO cars(id, make, model, year)
    values(1006, 'Ford', 'Mustang', 1971);
INSERT INTO cars(id, make, model, year)
    values(1007, 'Dodge', 'Charger', 1969);
```

SELECT

```
SELECT * FROM cars;
```

```
token@cqlsh:sandbox> select * from cars;
```

id	make	model	year
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1972
1001	Dodge	Challenger	1971
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1968

```
(7 rows)
```

```
token@cqlsh:sandbox> []
```



Hands-on Lab



LAB 02: Create a Table

- Create a table
- Insert data
- Retrieve data



Inserts, Updates and Upserts

Insert data and experiment with
duplicate or unknown primary keys

INSERTs with Duplicate Primary Keys

- What happens if you attempt to INSERT a row with a primary key that is already in the table?
 - In a relational database, the result would be an error because duplicate primary keys are not allowed
 - In Astra, the insert would be allowed and it would *update* the columns of the selected row

UPSETTs

- An UPSERT occurs if you INSERT a row with a primary key that is already in the table
 - UPSERTS act like UPDATES
- Astra does not check to see if the primary key is already in the database
 - Checking would require a *read-before-write* and add overhead to the write process

```
token@cqlsh:sandbox> INSERT INTO cars(id, make, model, year) values(1010, 'Chevy', 'Corvette', 1963);
token@cqlsh:sandbox> INSERT INTO cars(id, make, model, year) values(1010, 'Ford', 'Thunderbird', 1963);
token@cqlsh:sandbox> select * from cars where id=1010;
```

id	make	model	year
1010	Ford	Thunderbird	1963

```
(1 rows)
token@cqlsh:sandbox> █
```

UPDATE

- Similar to SQL UPDATE

```
UPDATE cars SET year = 1970 WHERE id = 1002;
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id=1002;
token@cqlsh:sandbox> select * from cars;
```

id	make	model	year
1006	Dodge	Challenger	1972
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1971
1001	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1970

UPDATE with Unknown Primary Key

- Try an update with a primary key that is not in the database

```
UPDATE cars SET year = 1970 WHERE id = 1012;
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1012;
token@cqlsh:sandbox> SELECT * FROM cars;
```

id	make	model	year
1006	Dodge	Challenger	1972
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1971
1001	Ford	Mustang	1969
1012	null	null	1970
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1970

IF EXISTS

- Use with UPDATE to force primary key check (*read-before-write*)
 - Returns *True* and update succeeds if key exists
 - Returns *False* and update fails if key does not exist

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1012 IF EXISTS;
```

```
[applied]
```

```
-----  
True
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1022 IF EXISTS;
```

```
[applied]
```

```
-----  
False
```



Hands-on Lab

LAB 03: INSERTs, UPSERTs and UPDATEs

- Use INSERT to add new rows and perform UPSERTs
- Use UPDATE
- Experiment with IF EXISTS



Partitions

Create and use partitions

Partitions

- Astra is a distributed database
- No single server holds all the data
 - Data is partitioned among Astra servers
- Understanding partitions is crucial to effectively using Astra

Partitions

This is the original table from this module

- Partitioning in Astra groups rows together
- All rows in the same partition are *guaranteed* to stored be on the same server
- Partitions affect how data is retrieved
- Defining partitions is a key part of data modeling

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Partitions

- Part of the data modeling process is deciding how to partition data
- In this table the data is partitioned by *make (Dodge, Chevy, Ford)*

1001	Dodge	Challenger	1971
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1007	Dodge	Charger	1969

1003	Chevy	Camaro	1969
------	-------	--------	------

1002	Ford	Mustang	1968
1006	Ford	Mustang	1971

Create the Tables with Partitions

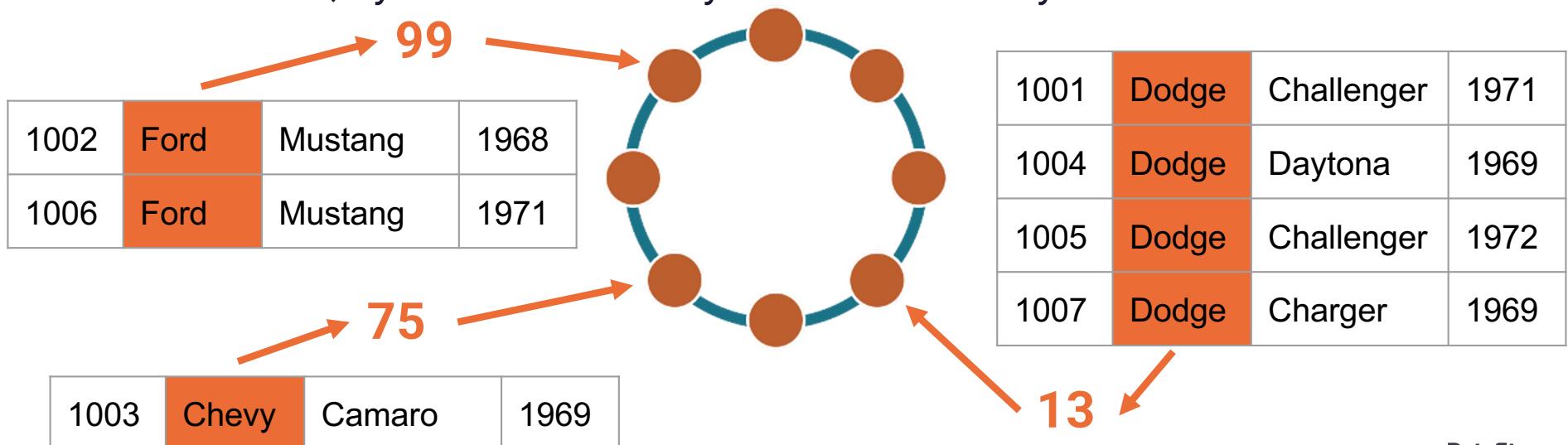
- In Astra (like an RDBMS) the primary key uniquely identifies a row
- Primary keys are made up of two parts:
 - Partition key (the column(s) in the inner parenthesis or first column if no inner parenthesis)
 - Clustering column(s) (the other columns in the primary key)

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make), id)
);
```

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY (make, id)
);
```

Partitions

- Astra hashes partition keys to create a token
- Each node *owns* a range of tokens
- Tokens are evenly distributed across the cluster
- Astra knows, by the token exactly where to find any row



Queries

- Queries with a *where clause* must include the partition key

```
token@cqlsh:class> SELECT * from cars WHERE make='Dodge';

  make | id   | model      | year
-----+-----+-----+-----+
  Dodge | 1001 | Challenger | 1971
  Dodge | 1004 | Daytona    | 1969
  Dodge | 1005 | Challenger | 1972
  Dodge | 1007 | Charger    | 1969

(4 rows)

token@cqlsh:class> SELECT * from cars WHERE year=1969;
InvalidRequest: Error from server: code=2200 [Invalid query
table performance. If you want to execute this query desp
token@cqlsh:class> ■
```

1001	Dodge	Challenger	1971
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1007	Dodge	Charger	1969
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971

🚫 ALLOW FILTERING

- ALLOW FILTERING enables queries without specifying the partition key
- This can be a very expensive operation
- This error message indicates a problem with the data model

ALLOW FILTERING is an anti-pattern,
DO NOT USE IT!!!

```
token@cqlsh:class> SELECT * from cars WHERE year=1969
;
InvalidRequest: Error from server: code=2200 [Invalid
query] message="Cannot execute this query as it migh
t involve data filtering and thus may have unpredicta
ble performance. If you want to execute this query de
spite the performance unpredictability, use ALLOW FIL
TERING"
token@cqlsh:class> SELECT * from cars WHERE year=1969
ALLOW FILTERING;
+-----+
| make | id   | model | year |
+-----+
| Dodge| 1004 | Daytona| 1969|
| Dodge| 1007 | Charger| 1969|
| Chevy| 1003 | Camaro | 1969|
+-----+
(3 rows)
token@cqlsh:class>
```

SELECT * FROM *table-name*;

- This query works in Astra even though the partition key is not included in the *WHERE* clause
- Astra will have to go to every node to get all the data anyway so there is no way to optimize this query

```
token@cqlsh:class> SELECT * FROM cars;  
  
make | id | model | year  
---+---+---+---  
Dodge | 1001 | Challenger | 1971  
Dodge | 1004 | Daytona | 1969  
Dodge | 1005 | Challenger | 1972  
Dodge | 1007 | Charger | 1969  
Chevy | 1003 | Camaro | 1969  
Ford | 1002 | Mustang | 1968  
Ford | 1006 | Mustang | 1971  
  
(7 rows)  
token@cqlsh:class> █
```

Wait a Minute ...

- What about the first table in this module?
 - The primary key has only one column
 - The id column *is* the partition key
 - Each partition only contains one row
- This is an *anti-pattern*
 - Queries for multiple rows require data from multiple partitions (possibly nodes)

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    make text,
    model text,
    year int
);
```

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Composite Partition Keys

- Composite partition keys are partition keys made up of multiple columns
 - All the columns of the partition key are hashed to create the token

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make, model), id)
);
```

1001	Dodge	Challenger	1971
1005	Dodge	Challenger	1972
1003	Chevy	Camaro	1969
1007	Dodge	Charger	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969

Queries with Composite Partition Keys

Valid queries

```
SELECT * FROM cars;  
SELECT * FROM cars WHERE make='Dodge'  
    AND model='Challenger';  
SELECT * FROM cars WHERE  
    make IN ('Dodge', 'Ford')  
    AND  
    model IN ('Challenger', 'Mustang');
```

Invalid queries

```
SELECT * FROM cars WHERE make='Dodge';  
SELECT * FROM cars WHERE model='Camaro';
```

1001	Dodge	Challenger	1971
1005	Dodge	Challenger	1972
1003	Chevy	Camaro	1969
1007	Dodge	Charger	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969



Hands-on Lab

LAB 04: Partition Keys

- Create a table and define partition keys
- Use partition keys to query results
- Learn query patterns and anti-patterns



Clustering Columns

Ordering data within partitions

Revisiting Partition Keys

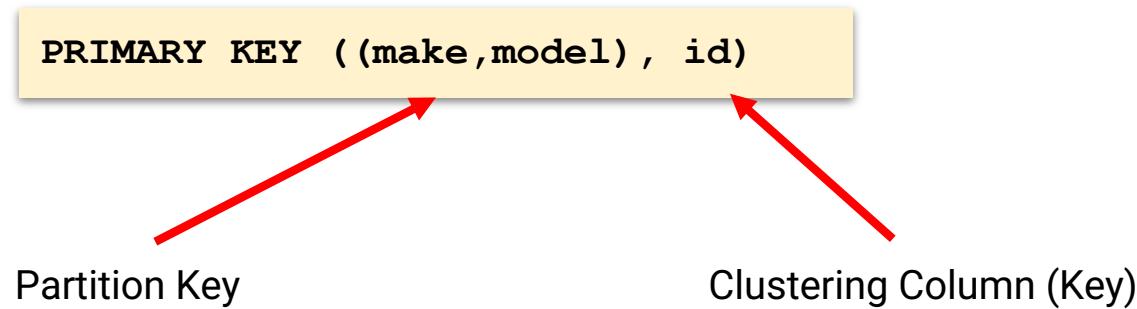
- In the previous section all the primary keys had multiple fields
 - Even if the partition key only had one field

```
PRIMARY KEY ((make), id)
PRIMARY KEY ((make, model), id)
```

- Primary keys have to be unique
 - Partition keys are not unique
 - All of the primary keys had *id* in them for uniqueness
- Without the *id* column multiple *Dodges* or multiple *Ford Mustangs* would have resulted in UPSERTs not INSERTs

Primary Keys and Clustering Columns

- Primary Keys in Astra uniquely identify rows
- Two parts
 - Partition key: one or more column(s) used to generate a token and group/distribute rows around the cluster
 - Clustering columns (keys): zero or more columns that define order of rows within a partition



Clustering Columns

- In this table the cars are grouped by the partition key (make, model)
- Within a partition (Dodge, Challenger) or (Ford, Mustang) the rows are ordered by id
- Clustering columns define both the logical and physical storage for data

make	model	id	year
Dodge	Challenger	1001	1971
Dodge	Challenger	1005	1972
Ford	Mustang	1002	1968
Ford	Mustang	1006	1971
Dodge	Daytona	1004	1969
Dodge	Charger	1007	1969
Chevy	Camaro	1003	1969

PRIMARY KEY ((make, model), id)

Queries with Clustering Columns

- All queries must use the partition key
 - This rule still applies
- Queries may use the clustering column(s)

```
SELECT * FROM cars;
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger';
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger'
    AND id = 1005;
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger'
    AND id > 1000 AND id < 1005;
```

Clustering Columns and Order

- Clustering columns define the *physical* ordering of data in memory or on disk
 - Default order is ascending
- Override default

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make, model), id)
) WITH CLUSTERING ORDER BY (id DESC);
```

Order in Tables

id: ascending (default) order

```
token@cqlsh:class> select * from cars ;  
  
make | model | id | year  
---+---+---+---  
Dodge | Challenger | 1001 | 1971  
Dodge | Challenger | 1005 | 1972  
Ford | Mustang | 1002 | 1968  
Ford | Mustang | 1006 | 1971  
Dodge | Daytona | 1004 | 1969  
Dodge | Charger | 1007 | 1969  
Chevy | Camaro | 1003 | 1969  
  
(7 rows)  
token@cqlsh:class>
```

id: descending order

```
token@cqlsh:class> select * from cars ;  
  
make | model | id | year  
---+---+---+---  
Dodge | Challenger | 1005 | 1972  
Dodge | Challenger | 1001 | 1971  
Ford | Mustang | 1006 | 1971  
Ford | Mustang | 1002 | 1968  
Dodge | Daytona | 1004 | 1969  
Dodge | Charger | 1007 | 1969  
Chevy | Camaro | 1003 | 1969  
  
(7 rows)  
token@cqlsh:class>
```

Ordering Data on Retrieval

```
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger';
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
(2 rows)
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger' ORDER BY id DESC;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
(2 rows)
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger' ORDER BY id ASC;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1001 | 1971
Dodge | Challenger | 1005 | 1972
```

```
token@cqlsh:class> select * from cars ;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
Ford | Mustang | 1006 | 1971
Ford | Mustang | 1002 | 1968
Dodge | Daytona | 1004 | 1969
Dodge | Charger | 1007 | 1969
Chevy | Camaro | 1003 | 1969
(7 rows)
token@cqlsh:class>
```

```
...
PRIMARY KEY ((make, model), id)
) WITH CLUSTERING ORDER BY (id DESC);
```



Hands-on Lab

LAB 05: Clustering Columns

- Create a table
- Insert data
- Retrieve the data



Multiple Clustering Columns

Hierarchical data ordering within a partition

Multiple Clustering Columns

- Two clustering columns
 - mileage
 - year
- One *non-key* column: color

```
CREATE TABLE cars (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((make, model), miles, year)
);
```

Make	Model	Miles	Year	Color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Ford	Mustang	45000	1968	green
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow
Chevy	Camaro	3100	1971	red
Chevy	Camaro	60000	1970	blue

Grouping and Ordering

	Make	Model	Miles	Year	Color
Ford, Mustang partition	Ford	Mustang	34000	1969	red
	Ford	Mustang	40000	1969	blue
	Ford	Mustang	45000	1968	green
Chevy, Camaro partition	Chevy	Camaro	13000	1969	red
	Chevy	Camaro	31000	1969	yellow
	Chevy	Camaro	31000	1971	red
	Chevy	Camaro	60000	1970	blue

Valid Queries with Multiple Clustering Columns

All Chevy Camaros	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro';</pre>
All Chevy Camaros highest miles first	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' ORDER BY miles DESC;</pre>
All Chevy Camaros with 31000 miles	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND miles=31000;</pre>
All Chevy Camaros with 31000 miles newest first	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND miles=31000 ORDER BY year DESC;</pre>

Query Rules - Constrain Clustering Columns L-R

- Order matters, if a *where* clause constrains a clustering column it must:
 - Constrain the first (L-R) before the second then, the second before the third ...
- In the primary key shown below a valid query would include one of:
 - *make* and *model*
 - *make*, *model* and *miles*
 - *make*, *model*, *miles* and *year*
 - *make*, *model*, *miles*, *year* and *price*

```
PRIMARY KEY (make, model), miles, year, price)
```

Invalid Queries with Multiple Clustering Columns

All Chevy Camaros from 1969	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND year=1969;</pre>	miles must be constrained before year
All Chevy Camaros that are red	<pre>SELECT * FROM cars WHERE make='Chevy' AND color='red';</pre>	where clause can only contain key columns
All cars 31000 miles	<pre>SELECT * FROM cars WHERE miles=31000;</pre>	where clause must include partition key
All Camaros	<pre>SELECT * FROM cars WHERE model='Camaro';</pre>	where clause must include all columns of partition key

Query Rules - Constrain Clustering Columns Ordering

- Order matters, if a *where* clause constrains a clustering column it must:
 - A table's *WITH CLUSTERING ORDER BY* clause determines the physical storage order
 - A query's *ORDER BY* clause determines the order in which results are retrieved
- The query *ORDER BY* must either agree with the table's *CLUSTERING ORDER BY* or reverse it entirely

Valid and Invalid Queries with ORDER BY

```
PRIMARY KEY (make, model), miles, year, price) WITH CLUSTERING ORDER BY  
(miles DESC, year ASC, price DESC);
```

Valid ORDER BY clause	Invalid ORDER BY clause
... ORDER BY miles ASC, year DESC;	... ORDER BY miles ASC, year ASC;
... ORDER BY miles DESC, year ASC, price DESC;	... ORDER BY miles ASC, year ASC, price ASC;
... ORDER BY miles ASC, year DESC, price ASC;	... ORDER BY miles ASC, year DESC, price DESC;
... ORDER BY miles DESC;	... ORDER BY price DESC;



Hands-on Lab

LAB 06: Multiple Clustering Columns

- Create a table with multiple clustering columns
- Use CLUSTERING ORDER BY in table definitions
- Use ORDER BY in queries

Replication and Consistency

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, white, sans-serif font with a stylized 'A', followed by "DB" in a smaller, white, sans-serif font.

Λ

Outline

- Replication
- Consistency



Replication

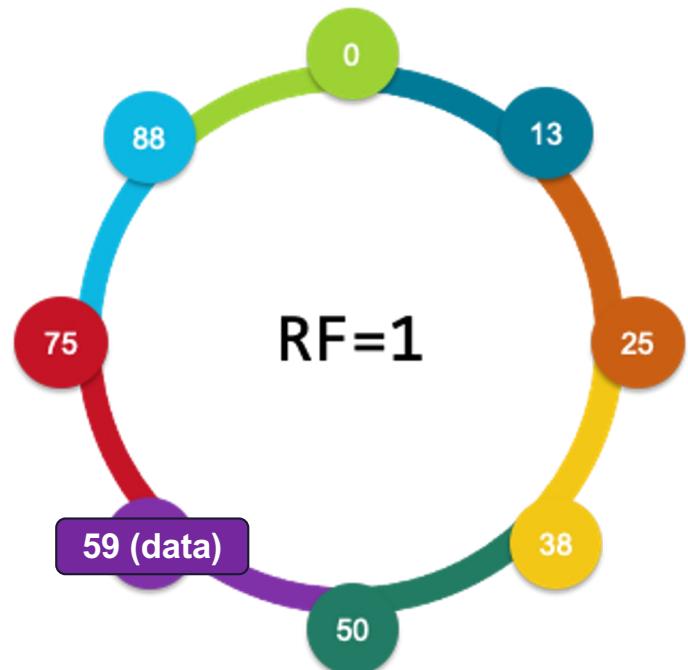
Data replication in Astra DB

Replication

- What happens if you attempt to INSERT a row with a primary key that is already in the table?
 - In a relational database, the result would be an error because duplicate primary keys are not allowed
 - In Astra, the insert would be allowed and it would *update* the columns of the selected row

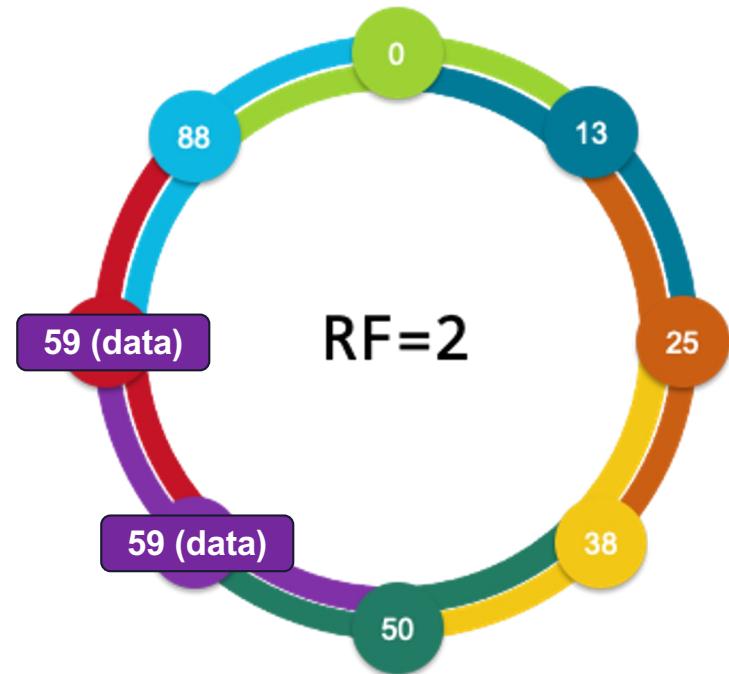
Replication Factor 1

- No replication
- Each node responsible for a subset of the token range (0-99)
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63



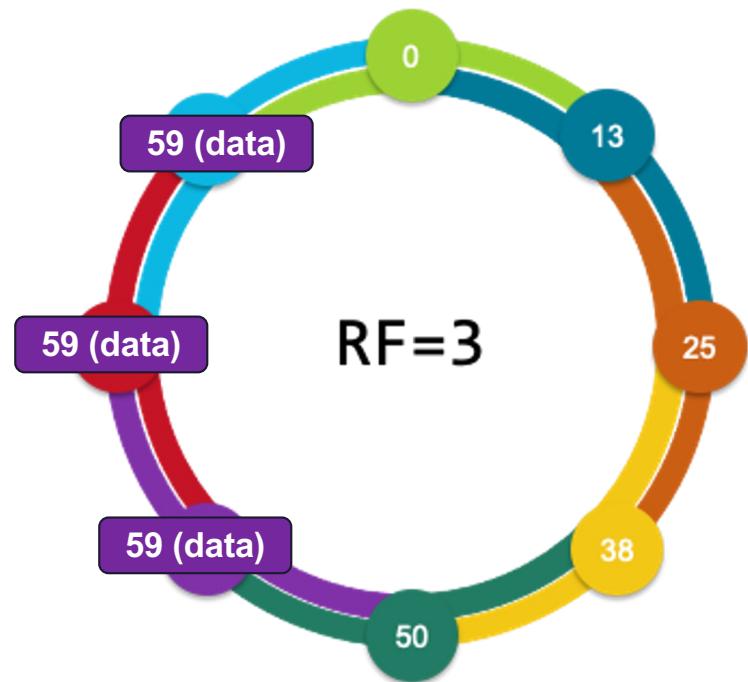
Replication Factor 2

- Each node responsible for the same subset of ranges
 - Plus a subset from their neighbor
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63
 - Also replicated to neighboring node



Replication Factor 3

- Each node responsible for the same subset of ranges
 - Plus a subset from two of their neighbors
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63
 - Also replicated to neighboring node(s)



Replication in Astra DB

- Replication Factor = 3
 - Cannot be changed
- Data is targeted to three nodes
 - Replica nodes are in different *availability zones (AZs)*
 - AZ failure means data is still available in two AZs
- Multiple data centers
 - Replication factor is 3 *per data center*

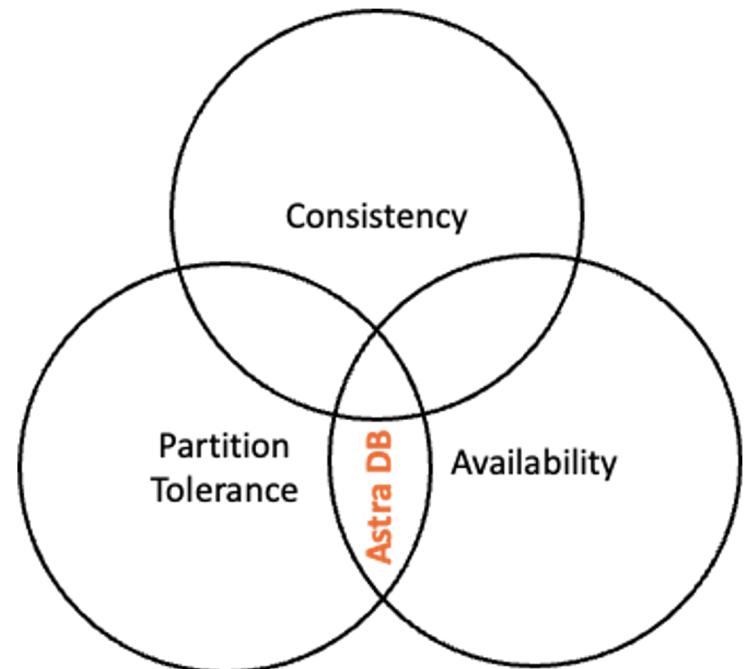
Λ

Consistency

Eventual consistency

Consistency

- Definition (simplified): the likelihood that client read the most current data
- In this CAP theorem diagram Astra DB is at the intersection of *partition tolerance* and *availability*
 - Astra does not emphasize consistency
- Consistency in Astra is *tunable*
 - Astra is more *opinionated* about consistency than OSS Cassandra



Consistency Level

- The number of replica nodes that must respond to an operation in order to declare it successful
- If there are not enough replicas to meet the consistency level the operation fails
- Consistency level is related to *replication factor*
 - Astra uses a replication factor of 3
 - Astra uses a default consistency level or LOCAL_QUORUM for reads and writes

Commonly Used Consistency Levels

Consistency Level	READS	WRITES	How many replicas must ACK
LOCAL_ONE	X	N/A	One replica in the local datacenter
LOCAL QUORUM	X	X	More than half of the replicas in the <i>local</i> datacenter
EACH_QUORUM	X	X	More than half of the replicas in <i>each</i> datacenter
ALL	X	X	All replicas

Set Consistency Level

- Use the `CONSISTENCY` command to set or view consistency level
- Consistency level can be set for each operation

```
token@cqlsh> CONSISTENCY;
Current consistency level is LOCAL_QUORUM.
token@cqlsh>
token@cqlsh> CONSISTENCY LOCAL_ONE;
Consistency level set to LOCAL_ONE.
token@cqlsh>
token@cqlsh> CONSISTENCY;
Current consistency level is LOCAL_ONE.
token@cqlsh>
```

Writing and Reading

- Astra will always try to write to the number of replicas (in each data center) required to satisfy the replication factor
 - The write will succeed if enough replicas ack to satisfy the consistency level
- Astra will only attempt to read from the number of replicas required to satisfy the consistency level
 - If those replicas do not ack then the read fails

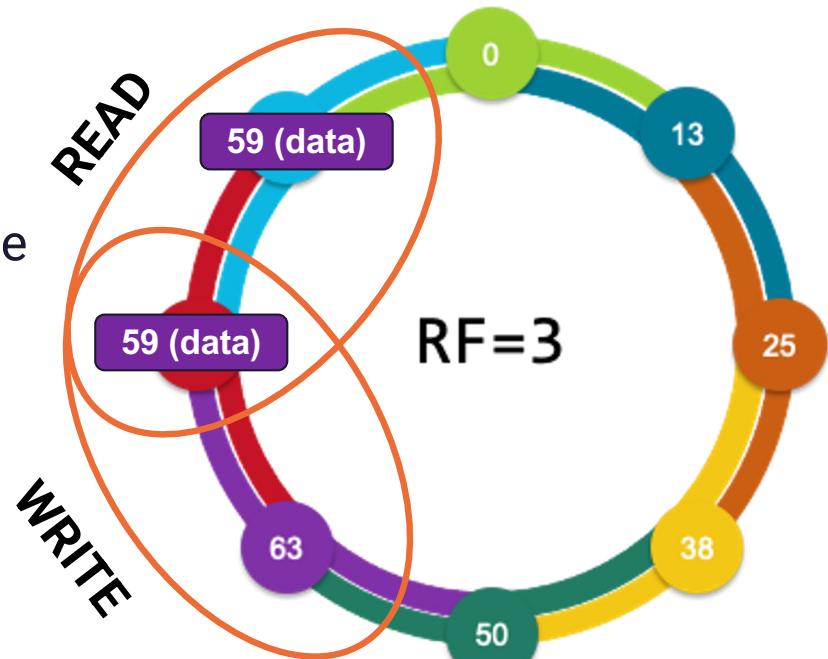
Consistency in Astra DB vs OSS Cassandra

- In OSS Cassandra you need to:
 - Set consistency levels
 - Configure commit log timeouts
 - Define tombstone expiry
 - Perform periodic repairs
 - Do compaction
- Astra DB allows you to set consistency levels
 - No other configuration is required to manage consistency

Immediate Consistency

$$CL_{\text{Write}} + CL_{\text{Read}} > RF \Rightarrow \text{Immediate Consistency}$$

- Write at LOCAL_QUORUM
 - Guarantees that more than half of the replicas have the data
- Read at LOCAL_QUORUM
 - Guarantees that at least one of the responding replicas has the data
- Read at LOCAL_ONE
 - No guarantees about the responding replica having the data





Hands-on Lab

LAB 07: Consistency Levels

- Read and write data with different consistency levels

Tables and Keys

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, sans-serif font with a stylized 'A', followed by "DB" in a smaller, regular sans-serif font.



Outline

- Simple Table Definition
- Inserts, Updates and Upserts
- Partitions
- Clustering Columns
- Multiple Clustering Columns



Simple Table Definition

Create and query simple tables

Tables

- Astra stores data in tables
- Tables consists of rows
- Rows consist of columns

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Cassandra Query Language – CQL

- Native language of Astra (and Cassandra)
- Syntax similar to SQL (Structured Query Language)

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    make text,
    model text,
    year int
);
```

INSERT

```
INSERT INTO cars(id, make, model, year)
    values(1001, 'Dodge', 'Challenger', 1971);
INSERT INTO cars(id, make, model, year)
    values(1002, 'Ford', 'Mustang', 1968);
INSERT INTO cars(id, make, model, year)
    values(1003, 'Chevy', 'Camaro', 1969);
INSERT INTO cars(id, make, model, year)
    values(1004, 'Dodge', 'Daytona', 1969);
INSERT INTO cars(id, make, model, year)
    values(1005, 'Dodge', 'Challenger', 1972);
INSERT INTO cars(id, make, model, year)
    values(1006, 'Ford', 'Mustang', 1971);
INSERT INTO cars(id, make, model, year)
    values(1007, 'Dodge', 'Charger', 1969);
```

SELECT

```
SELECT * FROM cars;
```

```
token@cqlsh:sandbox> select * from cars;
```

id	make	model	year
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1972
1001	Dodge	Challenger	1971
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1968

```
(7 rows)
```

```
token@cqlsh:sandbox> []
```



Hands-on Lab



LAB 02: Create a Table

- Create a table
- Insert data
- Retrieve data



Inserts, Updates and Upserts

Insert data and experiment with
duplicate or unknown primary keys

INSERTs with Duplicate Primary Keys

- What happens if you attempt to INSERT a row with a primary key that is already in the table?
 - In a relational database, the result would be an error because duplicate primary keys are not allowed
 - In Astra, the insert would be allowed and it would *update* the columns of the selected row

UPSETTs

- An UPSERT occurs if you INSERT a row with a primary key that is already in the table
 - UPSERTS act like UPDATES
- Astra does not check to see if the primary key is already in the database
 - Checking would require a *read-before-write* and add overhead to the write process

```
token@cqlsh:sandbox> INSERT INTO cars(id, make, model, year) values(1010, 'Chevy', 'Corvette', 1963);
token@cqlsh:sandbox> INSERT INTO cars(id, make, model, year) values(1010, 'Ford', 'Thunderbird', 1963);
token@cqlsh:sandbox> select * from cars where id=1010;
```

id	make	model	year
1010	Ford	Thunderbird	1963

```
(1 rows)
token@cqlsh:sandbox> █
```

UPDATE

- Similar to SQL UPDATE

```
UPDATE cars SET year = 1970 WHERE id = 1002;
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id=1002;
token@cqlsh:sandbox> select * from cars;
```

id	make	model	year
1006	Dodge	Challenger	1972
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1971
1001	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1970

UPDATE with Unknown Primary Key

- Try an update with a primary key that is not in the database

```
UPDATE cars SET year = 1970 WHERE id = 1012;
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1012;
token@cqlsh:sandbox> SELECT * FROM cars;
```

id	make	model	year
1006	Dodge	Challenger	1972
1004	Dodge	Daytona	1969
1007	Dodge	Charger	1969
1005	Dodge	Challenger	1971
1001	Ford	Mustang	1969
1012	null	null	1970
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1970

IF EXISTS

- Use with UPDATE to force primary key check (*read-before-write*)
 - Returns *True* and update succeeds if key exists
 - Returns *False* and update fails if key does not exist

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1012 IF EXISTS;
```

```
[applied]
```

```
-----  
True
```

```
token@cqlsh:sandbox> UPDATE cars SET year = 1970 WHERE id = 1022 IF EXISTS;
```

```
[applied]
```

```
-----  
False
```



Hands-on Lab

LAB 03: INSERTs, UPSERTs and UPDATEs

- Use INSERT to add new rows and perform UPSERTs
- Use UPDATE
- Experiment with IF EXISTS



Partitions

Create and use partitions

Partitions

- Astra is a distributed database
- No single server holds all the data
 - Data is partitioned among Astra servers
- Understanding partitions is crucial to effectively using Astra

Partitions

This is the original table from this module

- Partitioning in Astra groups rows together
- All rows in the same partition are *guaranteed* to stored be on the same server
- Partitions affect how data is retrieved
- Defining partitions is a key part of data modeling

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Partitions

- Part of the data modeling process is deciding how to partition data
- In this table the data is partitioned by *make (Dodge, Chevy, Ford)*

1001	Dodge	Challenger	1971
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1007	Dodge	Charger	1969

1003	Chevy	Camaro	1969
------	-------	--------	------

1002	Ford	Mustang	1968
1006	Ford	Mustang	1971

Create the Tables with Partitions

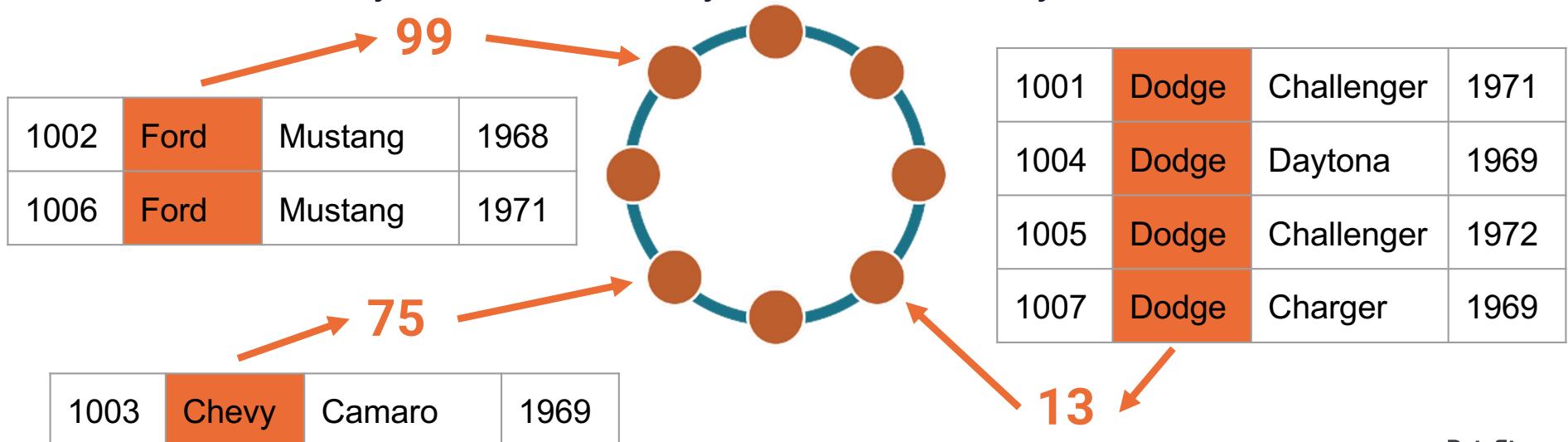
- In Astra (like an RDBMS) the primary key uniquely identifies a row
- Primary keys are made up of two parts:
 - Partition key (the column(s) in the inner parenthesis or first column if no inner parenthesis)
 - Clustering column(s) (the other columns in the primary key)

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make), id)
);
```

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY (make, id)
);
```

Partitions

- Astra hashes partition keys to create a token
- Each node *owns* a range of tokens
- Tokens are evenly distributed across the cluster
- Astra knows, by the token exactly where to find any row



Queries

- Queries with a *where clause* must include the partition key

```
token@cqlsh:class> SELECT * from cars WHERE make='Dodge';

  make | id   | model      | year
-----+-----+-----+-----+
  Dodge | 1001 | Challenger | 1971
  Dodge | 1004 | Daytona    | 1969
  Dodge | 1005 | Challenger | 1972
  Dodge | 1007 | Charger    | 1969

(4 rows)

token@cqlsh:class> SELECT * from cars WHERE year=1969;
InvalidRequest: Error from server: code=2200 [Invalid query
table performance. If you want to execute this query desp
token@cqlsh:class>
```

1001	Dodge	Challenger	1971
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1007	Dodge	Charger	1969
1003	Chevy	Camaro	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971

🚫 ALLOW FILTERING

- ALLOW FILTERING enables queries without specifying the partition key
- This can be a very expensive operation
- This error message indicates a problem with the data model

ALLOW FILTERING is an anti-pattern,
DO NOT USE IT!!!

```
token@cqlsh:class> SELECT * from cars WHERE year=1969
;
InvalidRequest: Error from server: code=2200 [Invalid
query] message="Cannot execute this query as it migh
t involve data filtering and thus may have unpredicta
ble performance. If you want to execute this query de
spite the performance unpredictability, use ALLOW FIL
TERING"
token@cqlsh:class> SELECT * from cars WHERE year=1969
ALLOW FILTERING;
+-----+
| make | id   | model | year |
+-----+
| Dodge| 1004 | Daytona| 1969|
| Dodge| 1007 | Charger| 1969|
| Chevy| 1003 | Camaro | 1969|
+-----+
(3 rows)
token@cqlsh:class>
```

SELECT * FROM *table-name*;

- This query works in Astra even though the partition key is not included in the *WHERE* clause
- Astra will have to go to every node to get all the data anyway so there is no way to optimize this query

```
token@cqlsh:class> SELECT * FROM cars;  
  
make | id | model | year  
---+---+---+---  
Dodge | 1001 | Challenger | 1971  
Dodge | 1004 | Daytona | 1969  
Dodge | 1005 | Challenger | 1972  
Dodge | 1007 | Charger | 1969  
Chevy | 1003 | Camaro | 1969  
Ford | 1002 | Mustang | 1968  
Ford | 1006 | Mustang | 1971  
  
(7 rows)  
token@cqlsh:class> █
```

Wait a Minute ...

- What about the first table in this module?
 - The primary key has only one column
 - The id column *is* the partition key
 - Each partition only contains one row
- This is an *anti-pattern*
 - Queries for multiple rows require data from multiple partitions (possibly nodes)

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    make text,
    model text,
    year int
);
```

ID	Make	Model	Year
1001	Dodge	Challenger	1971
1002	Ford	Mustang	1968
1003	Chevy	Camaro	1969
1004	Dodge	Daytona	1969
1005	Dodge	Challenger	1972
1006	Ford	Mustang	1971
1007	Dodge	Charger	1969

Composite Partition Keys

- Composite partition keys are partition keys made up of multiple columns
 - All the columns of the partition key are hashed to create the token

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make, model), id)
);
```

1001	Dodge	Challenger	1971
1005	Dodge	Challenger	1972
1003	Chevy	Camaro	1969
1007	Dodge	Charger	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969

Queries with Composite Partition Keys

Valid queries

```
SELECT * FROM cars;  
SELECT * FROM cars WHERE make='Dodge'  
    AND model='Challenger';  
SELECT * FROM cars WHERE  
    make IN ('Dodge', 'Ford')  
    AND  
    model IN ('Challenger', 'Mustang');
```

Invalid queries

```
SELECT * FROM cars WHERE make='Dodge';  
SELECT * FROM cars WHERE model='Camaro';
```

1001	Dodge	Challenger	1971
1005	Dodge	Challenger	1972
1003	Chevy	Camaro	1969
1007	Dodge	Charger	1969
1002	Ford	Mustang	1968
1006	Ford	Mustang	1971
1004	Dodge	Daytona	1969



Hands-on Lab

LAB 04: Partition Keys

- Create a table and define partition keys
- Use partition keys to query results
- Learn query patterns and anti-patterns



Clustering Columns

Ordering data within partitions

Revisiting Partition Keys

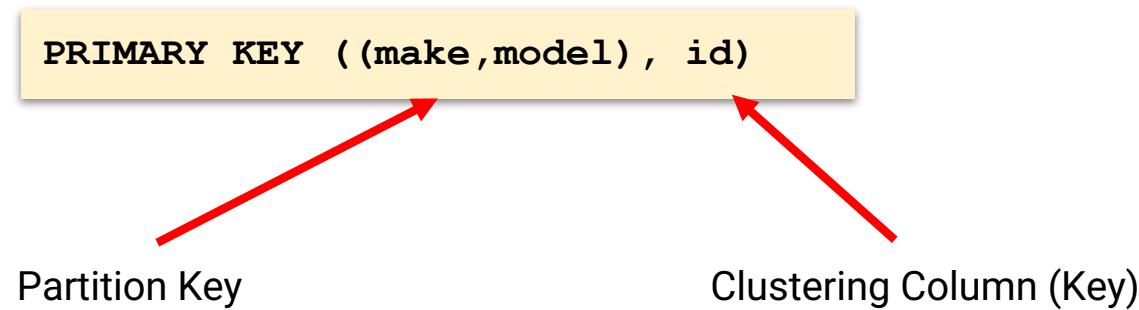
- In the previous section all the primary keys had multiple fields
 - Even if the partition key only had one field

```
PRIMARY KEY ((make), id)
PRIMARY KEY ((make, model), id)
```

- Primary keys have to be unique
 - Partition keys are not unique
 - All of the primary keys had *id* in them for uniqueness
- Without the *id* column multiple *Dodges* or multiple *Ford Mustangs* would have resulted in UPSERTs not INSERTs

Primary Keys and Clustering Columns

- Primary Keys in Astra uniquely identify rows
- Two parts
 - Partition key: one or more column(s) used to generate a token and group/distribute rows around the cluster
 - Clustering columns (keys): zero or more columns that define order of rows within a partition



Clustering Columns

- In this table the cars are grouped by the partition key (make, model)
- Within a partition (Dodge, Challenger) or (Ford, Mustang) the rows are ordered by id
- Clustering columns define both the logical and physical storage for data

make	model	id	year
Dodge	Challenger	1001	1971
Dodge	Challenger	1005	1972
Ford	Mustang	1002	1968
Ford	Mustang	1006	1971
Dodge	Daytona	1004	1969
Dodge	Charger	1007	1969
Chevy	Camaro	1003	1969

PRIMARY KEY ((make, model), id)

Queries with Clustering Columns

- All queries must use the partition key
 - This rule still applies
- Queries may use the clustering column(s)

```
SELECT * FROM cars;
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger';
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger'
    AND id = 1005;
SELECT * FROM cars WHERE make='Dodge'
    AND model='Challenger'
    AND id > 1000 AND id < 1005;
```

Clustering Columns and Order

- Clustering columns define the *physical* ordering of data in memory or on disk
 - Default order is ascending
- Override default

```
CREATE TABLE cars (
    id int,
    make text,
    model text,
    year int,
    PRIMARY KEY ((make, model), id)
) WITH CLUSTERING ORDER BY (id DESC);
```

Order in Tables

id: ascending (default) order

```
token@cqlsh:class> select * from cars ;  
  
make | model | id | year  
---+---+---+---  
Dodge | Challenger | 1001 | 1971  
Dodge | Challenger | 1005 | 1972  
Ford | Mustang | 1002 | 1968  
Ford | Mustang | 1006 | 1971  
Dodge | Daytona | 1004 | 1969  
Dodge | Charger | 1007 | 1969  
Chevy | Camaro | 1003 | 1969  
  
(7 rows)  
token@cqlsh:class>
```

id: descending order

```
token@cqlsh:class> select * from cars ;  
  
make | model | id | year  
---+---+---+---  
Dodge | Challenger | 1005 | 1972  
Dodge | Challenger | 1001 | 1971  
Ford | Mustang | 1006 | 1971  
Ford | Mustang | 1002 | 1968  
Dodge | Daytona | 1004 | 1969  
Dodge | Charger | 1007 | 1969  
Chevy | Camaro | 1003 | 1969  
  
(7 rows)  
token@cqlsh:class>
```

Ordering Data on Retrieval

```
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger';
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
(2 rows)
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger' ORDER BY id DESC;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
(2 rows)
token@cqlsh:class> SELECT * FROM cars WHERE make='Dodge' AND model='Challenger' ORDER BY id ASC;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1001 | 1971
Dodge | Challenger | 1005 | 1972
```

```
token@cqlsh:class> select * from cars ;
make | model | id | year
-----+-----+-----+-----+
Dodge | Challenger | 1005 | 1972
Dodge | Challenger | 1001 | 1971
Ford | Mustang | 1006 | 1971
Ford | Mustang | 1002 | 1968
Dodge | Daytona | 1004 | 1969
Dodge | Charger | 1007 | 1969
Chevy | Camaro | 1003 | 1969
(7 rows)
token@cqlsh:class>
```

```
...
PRIMARY KEY ((make, model), id)
) WITH CLUSTERING ORDER BY (id DESC);
```



Hands-on Lab

LAB 05: Clustering Columns

- Create a table
- Insert data
- Retrieve the data



Multiple Clustering Columns

Hierarchical data ordering within a partition

Multiple Clustering Columns

- Two clustering columns
 - mileage
 - year
- One *non-key* column: color

```
CREATE TABLE cars (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((make, model), miles, year)
);
```

Make	Model	Miles	Year	Color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Ford	Mustang	45000	1968	green
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow
Chevy	Camaro	3100	1971	red
Chevy	Camaro	60000	1970	blue

Grouping and Ordering

	Make	Model	Miles	Year	Color
Ford, Mustang partition	Ford	Mustang	34000	1969	red
	Ford	Mustang	40000	1969	blue
	Ford	Mustang	45000	1968	green
Chevy, Camaro partition	Chevy	Camaro	13000	1969	red
	Chevy	Camaro	31000	1969	yellow
	Chevy	Camaro	31000	1971	red
	Chevy	Camaro	60000	1970	blue

Valid Queries with Multiple Clustering Columns

All Chevy Camaros	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro';</pre>
All Chevy Camaros highest miles first	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' ORDER BY miles DESC;</pre>
All Chevy Camaros with 31000 miles	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND miles=31000;</pre>
All Chevy Camaros with 31000 miles newest first	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND miles=31000 ORDER BY year DESC;</pre>

Query Rules - Constrain Clustering Columns L-R

- Order matters, if a *where* clause constrains a clustering column it must:
 - Constrain the first (L-R) before the second then, the second before the third ...
- In the primary key shown below a valid query would include one of:
 - *make* and *model*
 - *make*, *model* and *miles*
 - *make*, *model*, *miles* and *year*
 - *make*, *model*, *miles*, *year* and *price*

```
PRIMARY KEY (make, model), miles, year, price)
```

Invalid Queries with Multiple Clustering Columns

All Chevy Camaros from 1969	<pre>SELECT * FROM cars WHERE make='Chevy' AND model='Camaro' AND year=1969;</pre>	miles must be constrained before year
All Chevy Camaros that are red	<pre>SELECT * FROM cars WHERE make='Chevy' AND color='red';</pre>	where clause can only contain key columns
All cars 31000 miles	<pre>SELECT * FROM cars WHERE miles=31000;</pre>	where clause must include partition key
All Camaros	<pre>SELECT * FROM cars WHERE model='Camaro';</pre>	where clause must include all columns of partition key

Query Rules - Constrain Clustering Columns Ordering

- Order matters, if a *where* clause constrains a clustering column it must:
 - A table's *WITH CLUSTERING ORDER BY* clause determines the physical storage order
 - A query's *ORDER BY* clause determines the order in which results are retrieved
- The query *ORDER BY* must either agree with the table's *CLUSTERING ORDER BY* or reverse it entirely

Valid and Invalid Queries with ORDER BY

```
PRIMARY KEY (make, model), miles, year, price) WITH CLUSTERING ORDER BY  
(miles DESC, year ASC, price DESC);
```

Valid ORDER BY clause	Invalid ORDER BY clause
... ORDER BY miles ASC, year DESC;	... ORDER BY miles ASC, year ASC;
... ORDER BY miles DESC, year ASC, price DESC;	... ORDER BY miles ASC, year ASC, price ASC;
... ORDER BY miles ASC, year DESC, price ASC;	... ORDER BY miles ASC, year DESC, price DESC;
... ORDER BY miles DESC;	... ORDER BY price DESC;



Hands-on Lab

LAB 06: Multiple Clustering Columns

- Create a table with multiple clustering columns
- Use CLUSTERING ORDER BY in table definitions
- Use ORDER BY in queries

Replication and Consistency

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, white, sans-serif font with a stylized 'A', followed by "DB" in a smaller, white, sans-serif font.

Λ

Outline

- Replication
- Consistency



Replication

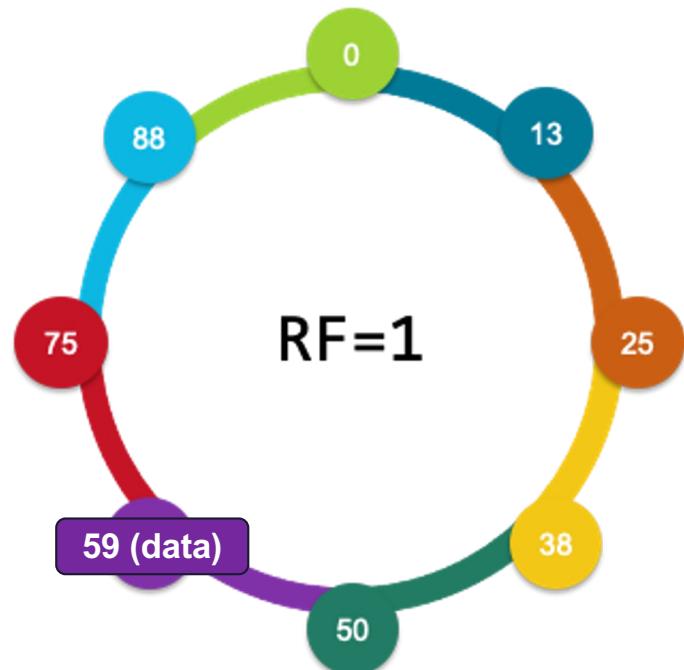
Data replication in Astra DB

Replication

- What happens if you attempt to INSERT a row with a primary key that is already in the table?
 - In a relational database, the result would be an error because duplicate primary keys are not allowed
 - In Astra, the insert would be allowed and it would *update* the columns of the selected row

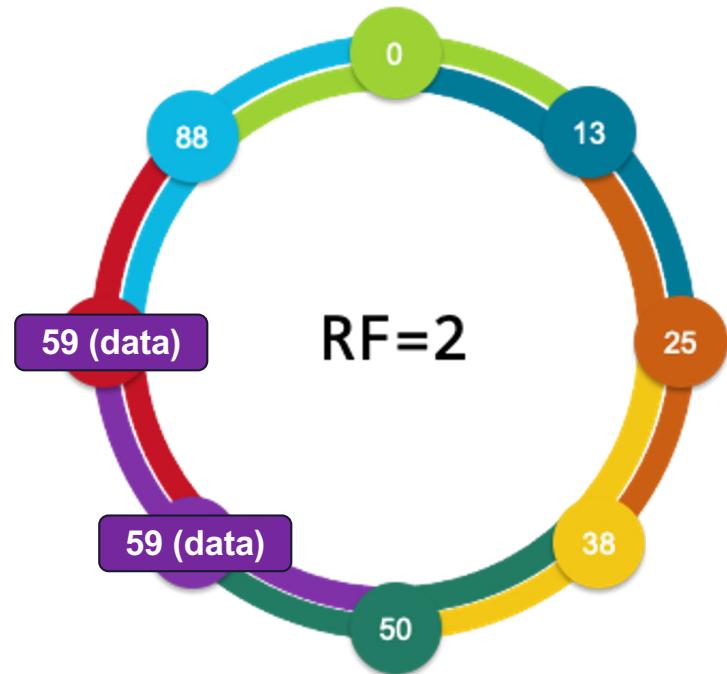
Replication Factor 1

- No replication
- Each node responsible for a subset of the token range (0-99)
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63



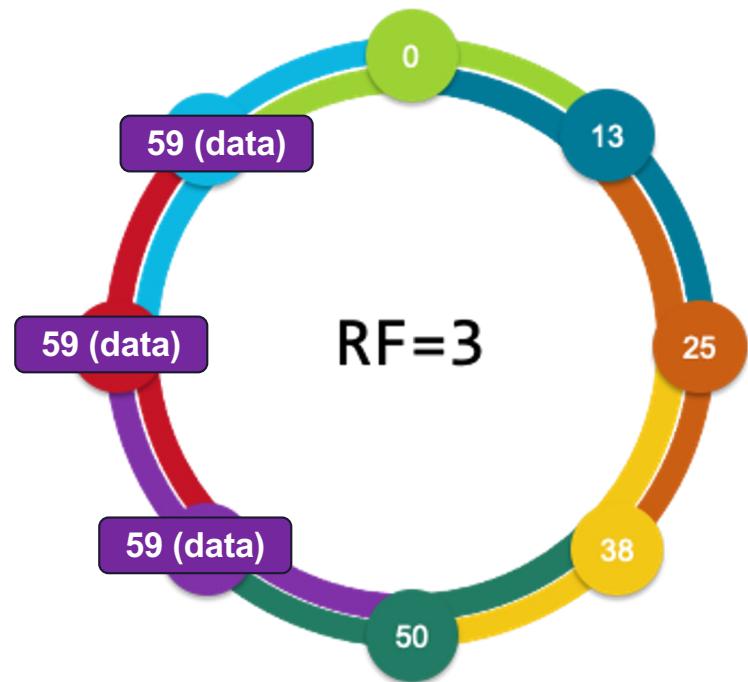
Replication Factor 2

- Each node responsible for the same subset of ranges
 - Plus a subset from their neighbor
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63
 - Also replicated to neighboring node



Replication Factor 3

- Each node responsible for the same subset of ranges
 - Plus a subset from two of their neighbors
- When data is inserted, the partition key is hashed
 - Token = 59
 - Stored on node with range 51-63
 - Also replicated to neighboring node(s)



Replication in Astra DB

- Replication Factor = 3
 - Cannot be changed
- Data is targeted to three nodes
 - Replica nodes are in different *availability zones (AZs)*
 - AZ failure means data is still available in two AZs
- Multiple data centers
 - Replication factor is 3 *per data center*

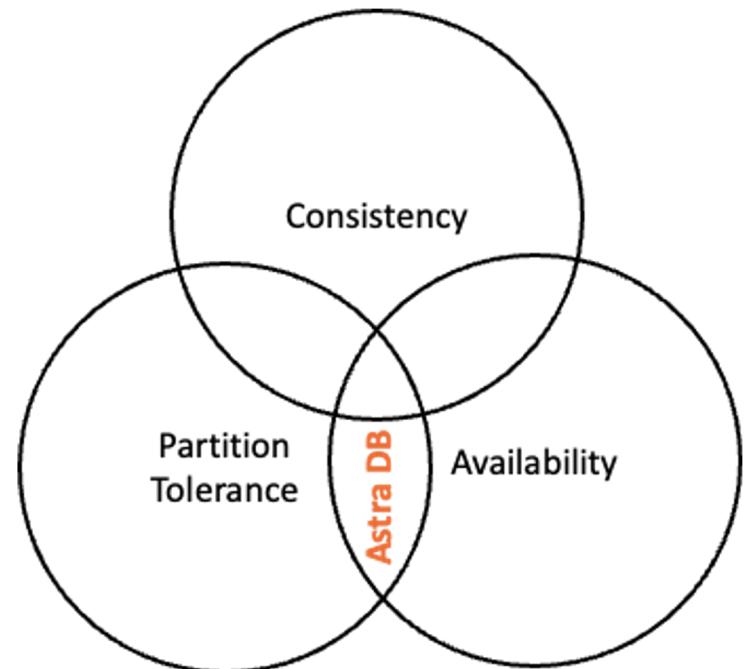


Consistency

Eventual consistency

Consistency

- Definition (simplified): the likelihood that client read the most current data
- In this CAP theorem diagram Astra DB is at the intersection of *partition tolerance* and *availability*
 - Astra does not emphasize consistency
- Consistency in Astra is *tunable*
 - Astra is more *opinionated* about consistency than OSS Cassandra



Consistency Level

- The number of replica nodes that must respond to an operation in order to declare it successful
- If there are not enough replicas to meet the consistency level the operation fails
- Consistency level is related to *replication factor*
 - Astra uses a replication factor of 3
 - Astra uses a default consistency level or LOCAL_QUORUM for reads and writes

Commonly Used Consistency Levels

Consistency Level	READS	WRITES	How many replicas must ACK
LOCAL_ONE	X	N/A	One replica in the local datacenter
LOCAL QUORUM	X	X	More than half of the replicas in the <i>local</i> datacenter
EACH_QUORUM	X	X	More than half of the replicas in <i>each</i> datacenter
ALL	X	X	All replicas

Set Consistency Level

- Use the `CONSISTENCY` command to set or view consistency level
- Consistency level can be set for each operation

```
token@cqlsh> CONSISTENCY;
Current consistency level is LOCAL_QUORUM.
token@cqlsh>
token@cqlsh> CONSISTENCY LOCAL_ONE;
Consistency level set to LOCAL_ONE.
token@cqlsh>
token@cqlsh> CONSISTENCY;
Current consistency level is LOCAL_ONE.
token@cqlsh>
```

Writing and Reading

- Astra will always try to write to the number of replicas (in each data center) required to satisfy the replication factor
 - The write will succeed if enough replicas ack to satisfy the consistency level
- Astra will only attempt to read from the number of replicas required to satisfy the consistency level
 - If those replicas do not ack then the read fails

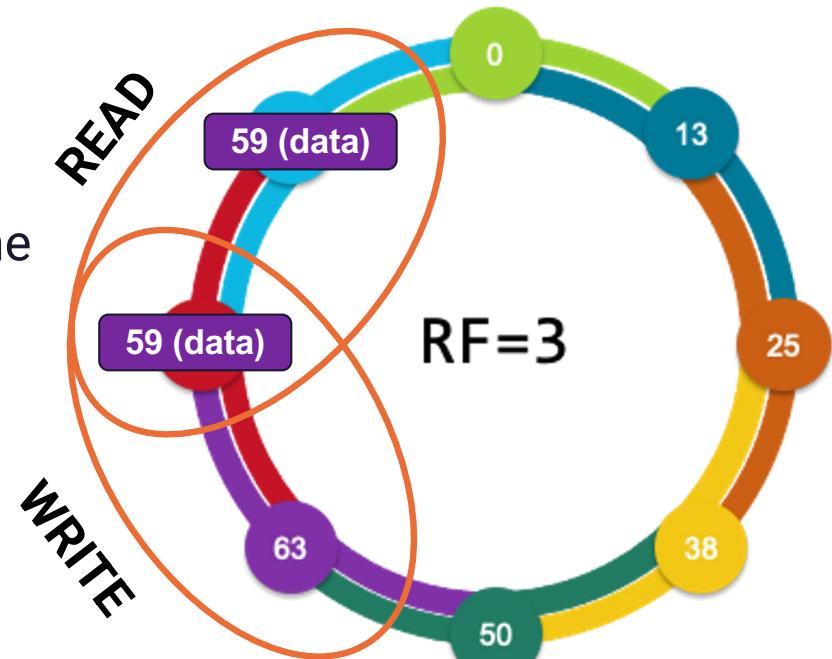
Consistency in Astra DB vs OSS Cassandra

- In OSS Cassandra you need to:
 - Set consistency levels
 - Configure commit log timeouts
 - Define tombstone expiry
 - Perform periodic repairs
 - Do compaction
- Astra DB allows you to set consistency levels
 - No other configuration is required to manage consistency

Immediate Consistency

$$CL_{\text{Write}} + CL_{\text{Read}} > RF \Rightarrow \text{Immediate Consistency}$$

- Write at LOCAL_QUORUM
 - Guarantees that more than half of the replicas have the data
- Read at LOCAL_QUORUM
 - Guarantees that at least one of the responding replicas has the data
- Read at LOCAL_ONE
 - No guarantees about the responding replica having the data





Hands-on Lab

LAB 07: Consistency Levels

- Read and write data with different consistency levels

Advanced Queries

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, white, sans-serif font with a stylized 'A', followed by "DB" in a smaller, white, sans-serif font.



Outline

- Illegal Queries
- Denormalization
- Storage Attached Indexes (SAI)



Illegal Queries

Explore query patterns that Astra does not allow

Relational Databases vs Astra

- If data is in a relational database there is a query that will retrieve it
 - Even if the data has to be combined from multiple tables
 - Even if the query is *expensive (time consuming)*
- Astra was designed to run high performance reads and writes
 - Some queries are simply not allowed
 - New queries may require data model changes
 - Query pattern *query-per-table*

Examples of Illegal Queries in Astra

- WHERE clauses that do not specify all the fields of the partition key
- ORDER BY clauses that violate the table (disk) ordering
- Queries that require multiple tables to complete
- Queries that constrain clustering columns out of order
- Case insensitive queries



Denormalization

It's OK to duplicate data

Denormalized Data is the Norm

- This one is hard – relational database experience is to normalize data
 - First Normal Form, Second Normal Form, Third Normal Form
- In Astra the opposite is true
 - It is perfectly normal to have *denormalized* data
 - Denormalized data allows for high performance queries
 - Table naming convention reflects denormalization
- It is common in Astra to have two tables with *exactly the same data*
 - With different partition or clustering keys to support different queries

Cars Table

- Two clustering columns
 - mileage
 - year
- One *non-key* column: color

```
CREATE TABLE cars (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((make, model), miles, year)
);
```

Make	Model	Miles	Year	Color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Ford	Mustang	45000	1968	green
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow
Chevy	Camaro	3100	1971	red
Chevy	Camaro	60000	1970	blue

Denormalize the Cars Table

- Supports multiple queries but does not support *find all red Chevy Camaros*
 - The data is in there but since color is not a clustering column this query cannot be performed

```
CREATE TABLE
cars_by_make_model_miles_year (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((make, model), miles, year)
);
```

```
CREATE TABLE
cars_by_color_make_model_miles_year (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((color), make, model, miles, year)
);
```

Queries

`cars_by_make_model_miles_year`

make	model	miles	year	color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Ford	Mustang	45000	1968	green
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow
Chevy	Camaro	31000	1971	red
Chevy	Camaro	60000	1970	blue

`cars_by_color_make_model_miles_year`

color	make	model	miles	year
red	Chevy	Camaro	13000	1969
red	Chevy	Camaro	31000	1971
red	Ford	Mustang	34000	1969
green	Ford	Mustang	45000	1968
blue	Chevy	Camaro	60000	1970
blue	Ford	Mustang	40000	1969
yellow	Chevy	Camaro	31000	1969

- Supports queries:

```
SELECT * FROM cars_by_make_model_miles_year  
WHERE make='Chevy' AND model='Camaro';
```

```
SELECT * FROM cars_by_color_make_model_miles_year  
WHERE color='red' AND make='Chevy' AND model='Camaro';
```

Keeping Denormalized Tables In Sync

- Use batches for all INSERT, UPDATE and DELETE operations
- Batches guarantee *atomicity* of operations
 - These are not ACID transactions you do not get *consistency* or *isolation*
 - No roll backs

```
BEGIN BATCH
    INSERT INTO cars_by_make_model_miles_year
        (make, model, year, miles, color)
        values ('Chevy', 'Camaro', 1971, 31000, 'red');
    INSERT INTO cars_by_color_make_model_miles_year
        (color, make, model, miles, year)
        values ('red', 'Chevy', 'Camaro', 31000, 1971);
APPLY BATCH;
```

Pros and Cons of Denormalized Tables

- Pros
 - High performance queries
 - Data is partitioned and clustering columns are tuned for specific queries
- Cons
 - Complex to maintain
 - Batches are not isolated



Hands-on Lab

LAB 08: Denormalization

- Create Denormalized Tables
- Update in Batches
- Run queries



Storage Attached Indexes SAI

A performant index scheme

Storage Attached Indexes

SAI provides traditional, relational database style indexing and querying capabilities for Astra which is easier to use, more efficient and simpler to maintain



Query Data using
non Primary-Key Columns



Eliminates the need to use ALLOW
FILTERING keyword or create custom
tables for each query pattern



The indexes live where the data
lives in Astra



Column-based for flexibility



Minimal user configuration



Not a Full-text Search

SAI

- Allows indexing on multiple fields
- Data is still distributed across the cluster by partition key (token)
 - SAI lets Astra know where to find particular values
 - There is still overhead involved in retrieving data from multiple nodes
- Compared to denormalization
 - Less performant
 - Simpler to implement and maintain

Cars Table

- This table does not support:
 - Finding all *red* cars
 - Finding all cars between 1968 and 1969

```
CREATE TABLE cars (
    make text,
    model text,
    year int,
    miles int,
    color text,
    PRIMARY KEY
        ((make, model), miles, year)
);
```

Make	Model	Miles	Year	Color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Ford	Mustang	45000	1968	green
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow
Chevy	Camaro	3100	1971	red
Chevy	Camaro	60000	1970	blue

Add SAI Index for Year to the Cars Table

- Add an index for year
 - Since the indexed column is an *int* there is no need to define any options for the index
- This command creates and populates the index

```
CREATE CUSTOM INDEX ON cars(year) USING 'StorageAttachedIndex';
```

```
token@cqlsh:class> SELECT * FROM cars WHERE year>1968 AND year <1970;
```

make	model	miles	year	color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow

Add SAI Index for Color to the Cars Table

- Add an index for color
- Options for *text* index
 - 'ascii': 'true' – converts characters to ascii (e.g. à to a)
 - 'case_sensitive': 'false' – allows case insensitive searches
 - 'normalize': 'true' – performs Unicode normalization

```
CREATE CUSTOM INDEX ON cars(color) USING 'StorageAttachedIndex'  
    WITH OPTIONS = {  
        'ascii': 'true',  
        'case_sensitive': 'false',  
        'normalize': 'true'};
```

Using SAI Indexes in Queries

- Normal query syntax
 - No need to specify the index
 - If the index exists, the query works

```
token@cqlsh:class> SELECT * FROM cars WHERE year>1968 AND year <1970;
```

make	model	miles	year	color
Ford	Mustang	34000	1969	red
Ford	Mustang	40000	1969	blue
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1969	yellow

```
token@cqlsh:class> SELECT * FROM cars WHERE color='RED';
```

make	model	miles	year	color
Ford	Mustang	34000	1969	red
Chevy	Camaro	13000	1969	red
Chevy	Camaro	31000	1971	red

SAI Odds and Ends

- The DESCRIBE TABLE command displays the SAI indexes associated with the table
- Default index name is *table-name_column-name_idx*
- SAI index are supported for collection types
- Remove index: DROP INDEX *index-name*
- Indexes are updated automatically with table updates

Pros and Cons of SAI Index

- Pros
 - Better performance than ALLOW FILTERING
 - Automatic updates
- Cons
 - May involve reading from multiple nodes
 - Does not take full advantage of partitioning
 - Less performant than denormalization



Hands-on Lab

LAB 09: Storage Attached Indexes (SAI)

- Add SAI to existing tables
- Update tables with SAI
- Queries with SAI

Complex Types

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, white, sans-serif font with a stylized 'A', followed by "DB" in a smaller, white, sans-serif font.



Outline

- Collections
- Querying Collection Values with SAI
- User Defined Types
- Frozen Types



Collections

Multi-valued fields

Collections

- Group and store data together in a column
- Collection columns are multi-valued columns
- Designed to store a small amount of data
- Retrieved in its entirety
- Cannot nest a collection inside another collection—unless you use FROZEN
 - More on FROZEN to come!

Restrictions

- Max number of items - 64k
 - In practice dozens or hundreds
- Collection columns may not be part of a primary key
 - Not allowed in partition key(s)
 - Not allowed in clustering key(s)
- Cannot use in WHERE clause unless indexed

List

- Ordered
 - Duplicates allowed

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    states_registered list<text>
);
INSERT INTO cars(id, model, year, states_registered)
    values(1001, 'Challenger', 1971, ['CA', 'NV', 'NM']);
```

id	model	states_registered	year
1001	Challenger	['CA', 'NV', 'NM']	1971

Modifying a List

- Amend or prepend using (+)

```
UPDATE cars SET states_registered=states_registered+['AZ'] WHERE id=1001;  
UPDATE cars SET states_registered=['WA']+states_registered WHERE id=1001;
```

id	model	states_registered	year
1001	Challenger	['WA', 'CA', 'NV', 'NM', 'AZ']	1971

- Replace an entire LIST

```
UPDATE cars SET states_registered=['HI','CA'] WHERE id=1001;
```

id	model	states_registered	year
1001	Challenger	['HI', 'CA']	1971

Deleting a List

- Update and set to an empty LIST
 - Individual items may not be deleted

```
UPDATE cars SET states_registered=[] WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>states_registered</code>	<code>year</code>
1001	Challenger	null	1971

SET

- Unique values
 - No Duplicates

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    options set<text>
);
INSERT INTO cars(id, model, year, options)
    values(1001, 'Camaro', 1969, {'FM radio', '427 engine','FM radio'});
```

<code>id</code>	<code>model</code>	<code>options</code>	<code>year</code>
1001	Camaro	{'427 engine', 'FM radio'}	1969

Modifying a SET

- Add to SET using (+)

```
UPDATE cars SET options=options+{'mag wheels'} WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>options</code>	<code>year</code>
1001	Camaro	{'427 engine', 'FM radio', 'mag wheels'}	1969

- Replace an entire SET

```
UPDATE cars SET options={'427 engine','racing stripe'} WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>options</code>	<code>year</code>
1001	Camaro	{'427 engine', 'racing stripe'}	1969

Deleting a SET

- Update and set to an empty SET
 - Individual items may not be deleted

```
UPDATE cars SET options={} WHERE id=1001;
```

id	model	options	year
1001	Camaro	null	1969

MAP

- Key-Value pairs
 - Keys and values may have different types

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    service map<text,text>
);
INSERT INTO cars(id, model, year, service) values(1001, 'Challenger',
    1971, {'JAN':'Oil Change', 'FEB':'Rotate tires'});
```

<code>id</code>	<code>model</code>	<code>service</code>	<code>year</code>
1001	Challenger	{'FEB': 'Rotate tires', 'JAN': 'Oil Change'}	1971

Modifying a MAP

- Add to MAP using (+)

```
UPDATE cars SET service=service+{ 'MAR' : 'Check fluids' } WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>service</code>	<code>year</code>
1001	Challenger	{'FEB': 'Rotate tires', 'JAN': 'Oil Change', 'MAR': 'Check fluids'}	1971

- Replace an entire MAP

```
UPDATE cars SET service={'JAN': 'Replace rear tires'} WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>service</code>	<code>year</code>
1001	Challenger	{'JAN': 'Replace rear tires'}	1971

Deleting from a MAP

- Delete an element from a MAP

```
DELETE service['JAN'] from cars where id =1001;
```

<code>id</code>	<code>model</code>	<code>service</code>	<code>year</code>
1001	Challenger	null	1971

- Update and set to an empty MAP

```
UPDATE cars SET service={} WHERE id=1001;
```

<code>id</code>	<code>model</code>	<code>service</code>	<code>year</code>
1001	Challenger	null	1971



Hands-on Lab

LAB 10: Collections

- Create tables with collections
- Modify the collections



Querying Collection Values with SAI

Use SAI to access collection values

Querying Collection Values

- Collections cannot be part of partition or clustering key
 - Cannot be queried directly
- Require SAI
 - Uses CONTAINS keyword
 - Works with MAP, LIST and SET types

Table Definition with SET/LIST

- Create a table with a SET

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    options set<text>
);
INSERT INTO cars(id, model, year, options)
    values(1001, 'Camaro', 1969, {'FM radio', '427 engine', 'FM radio'});
```

Create Index and Query for SET/LIST

- Query for options cannot be performed
 - Unless ALLOW FILTERING is enabled
 - Never use ALLOW Filtering
- Create an SAI index on the LIST/SET column

```
token@cqlsh:class> SELECT * FROM cars WHERE options CONTAINS '427 engine';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as
it might involve data filtering and thus may have unpredictable performance. If you want to execute
this query despite the performance unpredictability, use ALLOW FILTERING"
token@cqlsh:class> CREATE CUSTOM INDEX ON cars(options) USING 'StorageAttachedIndex';
token@cqlsh:class> SELECT * FROM cars WHERE options CONTAINS '427 engine';

  id   | model | options          | year
-----+-----+-----+-----+
  1001 | Camaro | {'427 engine', 'FM radio'} | 1969
```

Table Definition with MAP

- Create a table with a MAP

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    service map<text,text>
);
INSERT INTO cars(id, model, year, service) values(1001, 'Challenger',
    1971, {'JAN':'Oil Change','FEB':'Rotate tires'});
INSERT INTO cars(id, model, year, service) values(1002, 'Camaro',
    1969, {'FEB':'Check Battery','MAR':'Oil Change'});
INSERT INTO cars(id, model, year, service) values(1003, 'GTO',
    1964, {'MAR':'Oil Change'});
```

Indexes and Queries for MAPs

- MAPs are more complex because they have both keys and values
- By default MAPs are not searchable because they cannot be included in partition or clustering keys
- Indexing (SAI) enables searching on MAPs
- Three types of indexes
 - KEY
 - VALUE
 - ENTRY

KEY Index and Query for MAP

- Query for specific KEYS

```
token@cqlsh:class> CREATE INDEX ON cars( KEYS(service) );
token@cqlsh:class> SELECT * FROM cars WHERE service CONTAINS KEY 'FEB';

  id | model      | service                                         | year
----+-----+-----+-----+
  1001 | Challenger | {'FEB': 'Rotate tires', 'JAN': 'Oil Change'} | 1971
  1002 | Camaro     | {'FEB': 'Check Battery', 'MAR': 'Oil Change'} | 1969

(2 rows)
token@cqlsh:class> 
```

VALUE Index and Query for MAP

- Query for specific VALUES

```
token@cqlsh:class> CREATE INDEX ON cars( VALUES(service) );
token@cqlsh:class> SELECT * FROM cars WHERE service CONTAINS 'Oil Change';
```

id	model	service	year
1001	Challenger	{'FEB': 'Rotate tires', 'JAN': 'Oil Change'}	1971
1003	GTO	{'MAR': 'Oil Change'}	1964
1002	Camaro	{'FEB': 'Check Battery', 'MAR': 'Oil Change'}	1969

(3 rows)

```
token@cqlsh:class> █
```

ENTRY Index and Query for MAP

- Query for specific ENTRIES
 - KEY/VALUE pairs

```
token@cqlsh:class> CREATE INDEX ON cars( ENTRIES(service) );
token@cqlsh:class> SELECT * FROM cars WHERE service['MAR'] = 'Oil Change';

  id  | model | service                                | year
-----+-----+-----+-----+
  1003 |   GTO | {'MAR': 'Oil Change'} | 1964
  1002 | Camaro | {'FEB': 'Check Battery', 'MAR': 'Oil Change'} | 1969

(2 rows)
token@cqlsh:class>
```



Hands-on Lab

LAB 11: Collections and SAI

- Create a table with a MAP
- Create indexes to allow queries
- Query MAP values/keys/entries



User Defined Types (UDT)

Encapsulation of canonical types

User Defined Types

- Encapsulate related data
 - Named
 - Reusable
 - Scoped to a keyspace
- Use in tables like any other data type

Define a UDT

- Create the *owner* type

```
CREATE TYPE owner (
    last text,
    license int
);
```

Create a Table with a UDT

- Create a car table with an *owner* field

```
CREATE TABLE cars (
    id int PRIMARY KEY,
    model text,
    year int,
    customer owner
);
```

INSERTS in a Table with a UDT

- Create a car table with an owner field

```
INSERT INTO cars(id, model, year, customer)
    values(1001, 'BMW', 1969, {last:'Goldfinger', license:123});
INSERT INTO cars(id, model, year, customer)
    values(1002, 'Aston Martin', 1969, {last:'Largo', license:456});
INSERT INTO cars(id, model, year, customer)
    values(1003, 'Thunderbird', 1969, {last:'Blofeld', license:789});
```

id	customer	model	year
1001	{last: 'Goldfinger', license: 123}	BMW	1969
1003	{last: 'Blofeld', license: 789}	Thunderbird	1969
1002	{last: 'Largo', license: 456}	Aston Martin	1969

UPDATES in a Table with a UDT

- Update a field of the UDT

```
UPDATE cars SET customer.license=900 WHERE id=1002;
```

<code>id</code>	<code>customer</code>	<code>model</code>	<code>year</code>
1001	{last: 'Goldfinger', license: 123}	BMW	1969
1003	{last: 'Blofeld', license: 789}	Thunderbird	1969
1002	{last: 'Largo', license: 900}	Aston Martin	1969



Hands-on Lab

LAB 12: User Defined Types

- Create A UDT
- Use a UDT in a table
- Update a UDT



Frozen Types

Adding power to complex types

Frozen Types

- A Collection or UDT whose constituent values cannot be individually modified
 - The entire type may be replaced
- Benefits
 - May be used in partition or clustering keys
 - May be nested in UDTs or collections

Collection and UDTs

- When nesting collections and/or UDTs the *nested* type must be frozen

```
CREATE TYPE phone (
    type text,
    number text,
);

CREATE TABLE customers (
    id int PRIMARY KEY,
    first text,
    last text,
    phone_numbers list<frozen<phone>>
);
```

INSERT with a nested collection/UDT

```
INSERT INTO customers (id, first, last, phone_numbers)
  values(1001,'Glenn','Miller',
  [
    {type:'home',number:'PENNSYLVANIA 6-5000'},
    {type:'cell',number:'515-555-1212'}
  ]);
```

<code>id</code>	<code>first</code>	<code>last</code>	<code>phone_numbers</code>
1001	Glenn	Miller	[{type: 'home', number: 'PENNSYLVANIA 6-5000'}, {type: 'cell', number: '515-555-1212'}]

Update a Nested Collection/UDT

- Individual phone_numbers cannot be updated since the type is frozen
- Glenn Miller didn't have a cell phone so the following replaces the all the phone numbers and removes the cell phone

```
UPDATE customers SET phone_numbers =  
    [{type: 'home', number: 'PENNSYLVANIA 6-5000'}] WHERE id=1001;
```

id	first	last	phone_numbers
1001	Glenn	Miller	[{type: 'home', number: 'PENNSYLVANIA 6-5000'}]

Collection and UDT in Primary Keys

- One of the benefits of FROZEN types is that they can be used in Primary Keys
- In this example we will use a variant of the cars table for car collectors/collections
 - The collector (a UDT) will be the partition key
 - The partitions will be sorted by make and model
 - The VIN will be included in the primary key for uniqueness

Collection and UDTs

```
CREATE TYPE owner (
    first text,
    last text,
);

CREATE TABLE cars (
    collector frozen<owner>,
    make text,
    model text,
    vin text,
    year int,
    PRIMARY KEY((collector), make, model, vin, year)
);
```

INSERTs

```
INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jay',last:'Leno'},'McLaren','P1','MC1024',2015);
INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jay',last:'Leno'},'Porsche','356A Outlaw','P945',1957);
INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jay',last:'Leno'},'Stutz','Bearcat','S101',1918);

INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jerry',last:'Seinfeld'},'Porsche','911S','P9111',1970);
INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jerry',last:'Seinfeld'},'Lambourghini','Miura','LA1',1969);
INSERT INTO cars (collector, make, model, vin, year)
  values({first:'Jerry',last:'Seinfeld'},'Porsche','Carrera GT','PG0',2004);
```

Collection and UDT in Primary Keys

- Cars partitioned by owner (the UDT)
 - Sorted (within each partition) by make/model/vin

collector	make	model	vin	year
{first: 'Jerry', last: 'Seinfeld'}	Lambourghini	Miura	LA1	1969
{first: 'Jerry', last: 'Seinfeld'}	Porsche	911S	P9111	1970
{first: 'Jerry', last: 'Seinfeld'}	Porsche	Carrera GT	PG0	2004
{first: 'Jay', last: 'Leno'}	McLaren	P1	MC1024	2015
{first: 'Jay', last: 'Leno'}	Porsche	356A Outlaw	P945	1957
{first: 'Jay', last: 'Leno'}	Stutz	Bearcat	S101	1918

A Different Query

- Most of our queries have been limited to a single partition
 - Queries may span a *small number* of partitions using the `IN` keyword
- This query looks in two partitions (Jay Leno and Jerry Seinfeld)
 - It then looks for all Porsches (`make` is the first clustering column)

```
SELECT * FROM cars WHERE collector IN
  ({first: 'Jerry', last: 'Seinfeld'}, {first: 'Jay', last: 'Leno'})
  AND make = 'Porsche';
```

collector	make	model	vin	year
{first: 'Jay', last: 'Leno'}	Porsche	356A Outlaw	P945	1957
{first: 'Jerry', last: 'Seinfeld'}	Porsche	911S	P9111	1970
{first: 'Jerry', last: 'Seinfeld'}	Porsche	Carrera GT	PG0	2004



Hands-on Lab

LAB 13: Frozen Types

- Create a table using nested and frozen types
- Populate the table
- Execute queries



Tuples

Fixed size, fixed type and frozen

TUPLES

- User defined complex type
- Unlike Collections
 - Fixed number of elements
 - Predefined, heterogeneous types
- Unlike UDTs
 - Elements are not labeled
- FROZEN by default
 - May be used in partition and clustering keys
 - May be nested

Create a Table with a Tuple

- Most of our queries have been limited to a single partition

```
CREATE TABLE cars (
    owner tuple<text,text>,
    make text,
    model text,
    vin text,
    year int,
    PRIMARY KEY((owner), make, model, vin, year)
);
```

INSERTs

```
INSERT INTO cars (owner, make, model, vin, year)
  values('Jay','Leno'), 'McLaren', 'P1', 'MC1024', 2015);
INSERT INTO cars (owner, make, model, vin, year)
  values('Jay','Leno'), 'Porsche', '356A Outlaw', 'P945', 1957);
INSERT INTO cars (owner, make, model, vin, year)
  values('Jay','Leno'), 'Stutz', 'Bearcat', 'S101', 1918);

INSERT INTO cars (owner, make, model, vin, year)
  values('Jerry','Seinfeld'), 'Porsche', '911S', 'P9111', 1970);
INSERT INTO cars (owner, make, model, vin, year)
  values('Jerry','Seinfeld'), 'Lambourghini', 'Miura', 'LA1', 1969);
INSERT INTO cars (owner, make, model, vin, year)
  values('Jerry','Seinfeld'), 'Porsche', 'Carrera GT', 'PG0', 2004);
```

Tuple as Partition Key

- Cars partitioned by owner (the Tuple)
 - Sorted (within each partition) by make/model/vin

owner	make	model	vin	year
('Jerry', 'Seinfeld')	Lambourghini	Miura	LA1	1969
('Jerry', 'Seinfeld')	Porsche	911S	P9111	1970
('Jerry', 'Seinfeld')	Porsche	Carrera GT	PG0	2004
('Jay', 'Leno')	McLaren	P1	MC1024	2015
('Jay', 'Leno')	Porsche	356A Outlaw	P945	1957
('Jay', 'Leno')	Stutz	Bearcat	S101	1918

A Multi-Partition Query

```
SELECT * FROM cars WHERE owner IN (('Jerry', 'Seinfeld'), ('Jay', 'Leno'))  
AND make = 'Porsche';
```

owner	make	model	vin	year
('Jay', 'Leno')	Porsche	356A Outlaw	P945	1957
('Jerry', 'Seinfeld')	Porsche	911S	P9111	1970
('Jerry', 'Seinfeld')	Porsche	Carrera GT	PG0	2004

Question: How would you search for all cars owned by Jay Leno or Jerry Seinfeld manufactured before 1940?



Hands-on Lab

LAB 14: Tuples

- Create a table with tuples
- Insert data
- Execute queries

Other Types

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, sans-serif font with a stylized 'A', and "DB" in a smaller, regular sans-serif font.



Outline

- Basic Types
- Timestamps and Aggregation
- UUIDs and TimeUUIDs
- Counters
- Tombstones

Λ

Basic Types

Supported Types

Data Types - Text

- ASCII: US-ASCII characters
- TEXT: UTF-8 encoded string
- VARCHAR: UTF-8 encoded string

Data Types - Numeric

- TINYINT: 8-bit signed integer
- SMALLINT: 16-bit signed
- INT: 32-bit signed integer
- BIGINT: 64-bit signed integer
- VARINT: Arbitrary-precision integer--F-8 encoded string
- DECIMAL: Variable-precision decimal, supports integers and floats.
- FLOAT: 32-bit IEEE-754 floating point
- DOUBLE: 64-bit IEEE-754 floating point

Data Types—Date, Time and Unique Identifiers

- DATE: 32-bit unsigned integer, Unix time - days since epoch (Jan 1, 1970)
- DURATION: Signed 64-bit integer—amount of time in nanoseconds
- TIME: Encoded 64-bit signed—nanoseconds since midnight
- TIMESTAMP: 64-bit signed integer—date and time since epoch in milliseconds
- UUID: 128 bit universally unique identifier—generate with the UUID() function
- TIMEUUID: Unique identifier that includes a “conflict-free” timestamp—generate with the NOW() function

Data Types—Specialty Types

- BLOB: Arbitrary bytes (no validation), expressed as hexadecimal
- BOOLEAN: Stored internally as true or false
- COUNTER: 64-bit signed integer--only one counter column is allowed per table
- INET: IP address string in IPv4 or IPv6 format



Timestamps and Aggregation

Grouping by time

Timestamps

- 64 bit signed integers
 - Milliseconds since the epoch
- Literal
 - 1660263836000 - Raw number
 - '2022-08-12 00:24:06' - ISO 8061 (GMT)
 - '2022-08-12 00:20:06 -0500' - ISO 8061 with RFC-222 timezone (EST)

Using Timestamps

- Track visitors per hour at a collection of theme parks
 - Partitioned by park
 - Ordered by time

```
CREATE TABLE visitors_per_hour (
    park text,
    time timestamp,
    count int,
    PRIMARY KEY((park), time)
);
```

Using Timestamps

```
INSERT INTO visitors_per_hour (park,time,count)
    values('Dizzy World','2022-07-04 09:00 -0500',1001);
INSERT INTO visitors_per_hour (park,time,count)
    values('Dizzy World','2022-07-04 10:00 -0500',990);
INSERT INTO visitors_per_hour (park,time,count)
    values('Dizzy World','2022-07-04 11:00 -0500',1200);
INSERT INTO visitors_per_hour (park,time,count)
    values('Dizzy World','2022-07-04 12:00 -0500',878);
INSERT INTO visitors_per_hour (park,time,count)
    values('Epic','2022-07-04 09:00 -0500',755);
INSERT INTO visitors_per_hour (park,time,count)
    values('Epic','2022-07-04 10:00 -0500',600);
INSERT INTO visitors_per_hour (park,time,count)
    values('Epic','2022-07-04 11:00 -0500',590);
```

Using Timestamps

- Grouped by park
- Ordered by time
- Time was entered in EST returned in GMT

park	time	count
Epic	2022-07-04 14:00:00.000000+0000	755
	2022-07-04 15:00:00.000000+0000	600
	2022-07-04 16:00:00.000000+0000	590
Dizzy World	2022-07-04 14:00:00.000000+0000	1001
	2022-07-04 15:00:00.000000+0000	990
	2022-07-04 16:00:00.000000+0000	1200
	2022-07-04 17:00:00.000000+0000	878

Using Timestamps in WHERE Clauses

```
SELECT count FROM visitors_per_hour  
WHERE park='Dizzy World' AND time='2022-07-04 10:00 -0500';
```

```
count  
----  
990
```

```
SELECT count FROM visitors_per_hour WHERE park='Dizzy World' AND  
time > '2022-07-04 09:00 -0500' AND time < '2022-07-04 12:00 -0500';
```

```
count  
----  
990  
1200
```

Aggregate Functions

- Aggregate data and return a single result
- Available functions
 - Count
 - Max
 - Min
 - Sum
 - Avg

Using Timestamps with Aggregation

```
SELECT SUM(count) FROM visitors_per_hour  
WHERE park='Dizzy World'  
AND time > '2022-07-04 09:00 -0500'  
AND time < '2022-07-04 12:00 -0500';
```

```
system.sum(count)
```

```
-----  
2190
```

```
SELECT AVG(count) FROM visitors_per_hour  
WHERE park='Dizzy World';
```

```
system.avg(count)
```

```
-----  
1017
```

Partition Size

- In the theme park example the data sampled visitors hourly
- What happens in an IoT example where the data ingestion is more frequent
 - Example: water temp, air temp every 10 seconds from 1000 buoys in 40 different harbors

```
CREATE TABLE buoys (
    harbor int,
    buoy text,
    air decimal,
    h20 decimal,
    time timestamp,
    PRIMARY KEY((harbor), buoy, time)
);
```

Buckets

- The problem is that there are 86,400 seconds in a day so 8,640 entries per buoy with 250 buoys per harbor
 - The result is over 2 million rows per partition per day!
- To keep the partition sizes manageable, add another field to the partition key to represent the hour of the day
 - This breaks the partition up into 1 hour *buckets*
 - Each bucketed partition would contain a manageable 90,000 rows

```
CREATE TABLE buoys (
    harbor int,
    bucket int,
    ...
    PRIMARY KEY((harbor, bucket), buoy, time)
);
```



Hands-on Lab



LAB 15: Timestamps and Aggregation

- Create a table that uses timestamps
- Retrieve values
- Retrieve Aggregate values



UUIDs and TimeUUIDs

Uniqueness

The need for Uniqueness

- In Astra (like RDBMSs) primary keys must be unique
- Many Astra tables have *compound* primary keys
 - The partition key fields are not unique
 - At least one field (or a combination of fields) must be unique
- Some data already contains a unique value that can be used as a *natural* key
 - Examples: ISBNs, SKUs, SSNs, email addresses, IPs, phone numbers
- Other data has no unique values and requires a *synthetic* key
 - Examples: Student names, Appointments

Uniqueness in Astra DB

- Because Astra is distributed the RDBMS style strategies for creating unique keys will not work
 - E.g retrieving keys from a sequence table would require increased consistency levels and force a *read-before-write* on inserts
- Astra uses *UUIDs* and *TimeUUIDs* for uniqueness
 - RFC 4122
 - 128 bit values

Uniqueness in Astra DB

- In this table *make* is the partition key
- The UUID is necessary for uniqueness

```
CREATE TABLE cars (
    make text,
    model text,
    year int,
    id uuid,
    color text,
    PRIMARY KEY((make), model, year, id)
);
```

Uniqueness in Astra DB

- Notice that this table has two 1969, yellow, Chevy Camaros.
- None of the data *natural* data columns are unique

```
INSERT INTO cars (make, model, year, id, color) values
  ('Ford', 'Mustang', 1968, uuid(), 'red');
INSERT INTO cars (make, model, year, id, color) values
  ('Chevy', 'Camaro', 1969, uuid(), 'yellow');
INSERT INTO cars (make, model, year, id, color) values
  ('Pontiac', 'GTO', 1964, uuid(), 'blue');
INSERT INTO cars (make, model, year, id, color) values
  ('Chevy', 'Camaro', 1969, uuid(), 'yellow');
INSERT INTO cars (make, model, year, id, color) values
  ('Dodge', 'Charger', 1968, uuid(), 'red');
```

Uniqueness in Astra DB

- If the UUID were not part of the primary key there would only be one Camaro in the table
 - The UUID (being part of the primary key) prevented an *upsert*

make	model	year	id	color
Pontiac	GTO	1964	d119e25c-9b64-4022-9601-48bbfa5246ba	blue
Dodge	Charger	1969	14075ea3-b4a3-47d3-ba1e-05e1d47b-e11	red
Chevy	Camaro	1969	38489fa8-b79c-4d0f-8a96-c62302dec0fd	yellow
Chevy	Camaro	1969	3fd519a2-dfc8-48db-b8f4-64e67f498efe	yellow
Ford	Mustang	1968	08c144b1-071d-4726-9090-c0a002667044	red

TimeUUIDs

- Encapsulate timestamp data into a unique identifier
- Useful as clustering keys because they provide uniqueness and order

```
CREATE TABLE truck_weight (
    id int,
    time timeuuid,
    station text,
    weight int,
    PRIMARY KEY((id), time)
);
```

Inserting data

- The `now()` function generates a new TimeUUID each time it is invoked
 - The TimeUUID serves two purposes: *encoding time* and *uniqueness*
- Every time the truck is weighed this INSERT specifies the station code, time and weight

```
INSERT INTO truck_weight (id,time,station,weight)
    values(1001,now(),'B100',2300);
```

<code>id</code>	<code>time</code>	<code>station</code>	<code>weight</code>
1001	d7c77920-19e6-11ed-9fdf-6793e7745935	A123	2300
1001	debdb320-19e6-11ed-97c9-cdba8d30ef73	B432	2300
1001	e4ccc170-19e6-11ed-867c-bb58ca097d5b	B100	2300

TimeUUID Functions

- **dateOf()** - used in a SELECT clause, this function extracts the timestamp of a timeuuid column in a result set.

```
SELECT dateOf(time), weight FROM truck_weight;
```

system.dateof(time)		weight
2022-08-12 02:31:17.170000+0000		2300
2022-08-12 02:31:28.850000+0000		2300
2022-08-12 02:31:39.015000+0000		2300

TimeUUID Functions

- `minTimeuuid()` and `maxTimeuuid()` - returns a UUID-like result given a conditional time component as an argument

```
SELECT * FROM truck_weight WHERE id = 1001
    AND time > maxTimeuuid('2022-08-12 02:31:18')
    AND time < minTimeuuid('2022-08-12 02:31:38');
```

<code>id</code>	<code>time</code>	<code>station</code>	<code>weight</code>
1001	debdb320-19e6-11ed-97c9-cdba8d30ef73	B432	2300

Related Functions

- `toDate(timeuuid)` - converts timeuuid to date in YYYY-MM-DD format
- `toTimestamp(timeuuid)` - converts timeuuid to timestamp format
- `toUnixTimestamp(timeuuid)` - converts timeuuid to UNIX timestamp format
- `toDate(timestamp)` - converts timestamp to date in YYYY-MM-DD format
- `toUnixTimestamp(timestamp)` - converts timestamp to UNIX timestamp format
- `toTimestamp(date)` - converts date to timestamp format
- `toUnixTimestamp(date)` - converts date to UNIX timestamp format



Hands-on Lab

LAB 16: UUIDs and TimeUUIDs

- Use UUIDs and TimeUUIDs for uniqueness
- Query a range of results using a TimeUUID



Counters

Keeping tabs in a distributed CNDB world

Counters

- Column used to store a 64-bit signed integer
- Changed incrementally—incremented or decremented
- Values are changed using UPDATE
 - UPDATE is the only operation permitted
- Need specially dedicated tables—can only have primary key and counter columns
- Can have more than one counter column
- Initial counter value is 0

Table With Counter Columns

- Track security card swipes entering and exiting a facility
- Primary key is the card ID

```
CREATE TABLE card_swipes (
    card text PRIMARY KEY,
    entry_swipes counter,
    exit_swipes counter
);
```

Table With Counter Columns

- Track security card swipes entering and exiting a facility
- Primary key is the card ID

```
CREATE TABLE card_swipes (
    card text PRIMARY KEY,
    entry_swipes counter,
    exit_swipes counter
);
```

Table With Counter Columns

- Card *A123* swipes in and out
- Card *B456* swipes in
- Card *C789* swipes in, out and in again
- No need to initialize the counter, automatically set to 0
 - UPDATE is the only operation that can be performed on a counter table
 - First entry (swipe) with a new card is an *UPDATE* not an *INSERT*

```
UPDATE card_swipes SET entry_swipes = entry_swipes + 1 WHERE card = 'A123';
UPDATE card_swipes SET entry_swipes = entry_swipes + 1 WHERE card = 'B456';
UPDATE card_swipes SET entry_swipes = entry_swipes + 1 WHERE card = 'C789';
UPDATE card_swipes SET exit_swipes = exit_swipes + 1 WHERE card = 'A123';
UPDATE card_swipes SET exit_swipes = exit_swipes + 1 WHERE card = 'C789';
UPDATE card_swipes SET entry_swipes = entry_swipes + 1 WHERE card = 'C789';
```

Table With Counter Columns

- Track security card swipes entering and exiting a facility
- Primary key is the card ID

card	entry_swipes	exit_swipes
B456	1	null
C789	2	1
A123	1	1

Counter Concerns

- Keeping counters *in-sync* in a distributed system is difficult
- Counter operations are not *idempotent*
- Astra DB counters are *mostly accurate*
 - Counters should not be used when precision is required
 - Appropriate for behavior/trends (e.g. web clicks)



Hands-on Lab

LAB 17: Counters

- Create a table with a counter
- Increment the counter value



Static Columns

Deletions in Astra DB

Static Columns

- Keyword STATIC
- Only for tables with clustering columns
 - Support for multiple rows
- Same value for all rows in same partition
- Only stored once (per partition) in memory and on disk

Table With a Static Columns

- Track bowling scores
- Each bowler has a nickname

```
CREATE TABLE scores (
    first text,
    last text,
    session int,
    nickname text STATIC,
    score tuple<int,int,int>,
    PRIMARY KEY((first,last),session)
);
```

Table With a Static Columns

- Enter two scores for one bowler
 - Each score uses a different nickname

```
INSERT INTO scores (first, last, session, nickname, score)
    values ('Donald', 'Duck', 1, 'Don', (210, 198, 201));
INSERT INTO scores (first, last, session, nickname, score)
    values ('Donald', 'Duck', 2, 'DD', (205, 230, 219));
```

first	last	session	nickname	score
Donald	Duck	1	DD	(210, 198, 201)
Donald	Duck	2	DD	(205, 230, 219)

Set the Static Column Value

- Use UPDATE
 - WHERE clause specifies the partition

```
UPDATE scores set nickname='The OG Duck'  
WHERE first='Donald' AND last='Duck';
```

first	last	session	nickname	score
Donald	Duck	1	The OG Duck	(210, 198, 201)
Donald	Duck	2	The OG Duck	(205, 230, 219)

Static Column Values for Multiple Partitions

- Each partition has its own static value(s)

```
INSERT INTO scores (first, last, session, nickname, score)
    values ('Mickey', 'Mouse', 1, 'Mickey', (233, 210, 222));
INSERT INTO scores (first, last, session, score)
    values ('Mickey', 'Mouse', 2, (199, 222, 211));
```

first	last	session	nickname	score
Donald	Duck	1	The OG Duck	(210, 198, 201)
Donald	Duck	2	The OG Duck	(205, 230, 219)
Mickey	Mouse	1	Mickey	(233, 210, 222)
Mickey	Mouse	2	Mickey	(199, 222, 211)



Hands-on Lab



LAB 18: Static Columns

- Define static columns in a table
- Update static values
- Retrieve static values



Tombstones

Deletions in Astra DB

Deletions in Systems

- Deletions are challenging in distributed systems
- If all nodes are running the system could wait until all nodes acknowledge a delete
 - Deletes would fail if all nodes are not available
- The system could inform running nodes about a delete
 - If a node re-starts it might not know about the delete and could introduce a *zombie* into the system

Deletions in Astra DB

- Astra DB uses *tombstones* to mark deleted data
- Tombstones are treated as INSERTS or UPSERTS
 - Have timestamps to compare with data to prevent zombies
- Unlike Apache Cassandra™, Astra DB *cleans up tombstones automatically*

Best Practices

- Minimize deletes where possible
 - Too many tombstones is an anti-pattern
- Do not set values to null on initial INSERTS

```
/* inserts a tombstone for status */
INSERT INTO members (first, last, status) values ('Donald', 'Duck', null);

/* no tombstone */
INSERT INTO members (first, last) values ('Mickey', 'Mouse');
```

Data Modeling

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, sans-serif font with a stylized 'A', followed by "DB" in a smaller, regular sans-serif font.

Astra DB



Outline

- Query Patterns
- Methodology
- Optimization Techniques



Query Patterns

Data modeling fundamentals

Astra DB Data Model

- Tables
 - Rows, columns, primary keys
 - Partitions, partition keys
- Tables with single-row partitions
 - Primary key = partition key
- Tables with multi-row partitions
 - Primary key = partition key + clustering key

Implications for Data Modeling: Data Perspective

- Primary keys define data uniqueness
- Partition keys define data distribution
- Partition keys affect partition sizes
- Clustering keys define row ordering

Implications for Data Modeling: Query Perspective

- Primary keys define how data is retrieved
- Partition keys allow equality predicates
- Clustering keys allow inequality predicates and ordering
- Only one table per query, no joins
 - Accessing one partition (OLTP)
 - Accessing a few partitions
 - Accessing many partitions (OLAP)



Methodology

The Astra (Cassandra) way

What is Data Modeling

Collection and analysis of **data requirements**

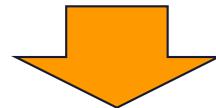
Identification of participating **entities** and relationships

Identification of data **access patterns**

A particular way of **organizing** and structuring data

Design and specification of a **database schema**

Schema **optimization** and data **indexing** techniques



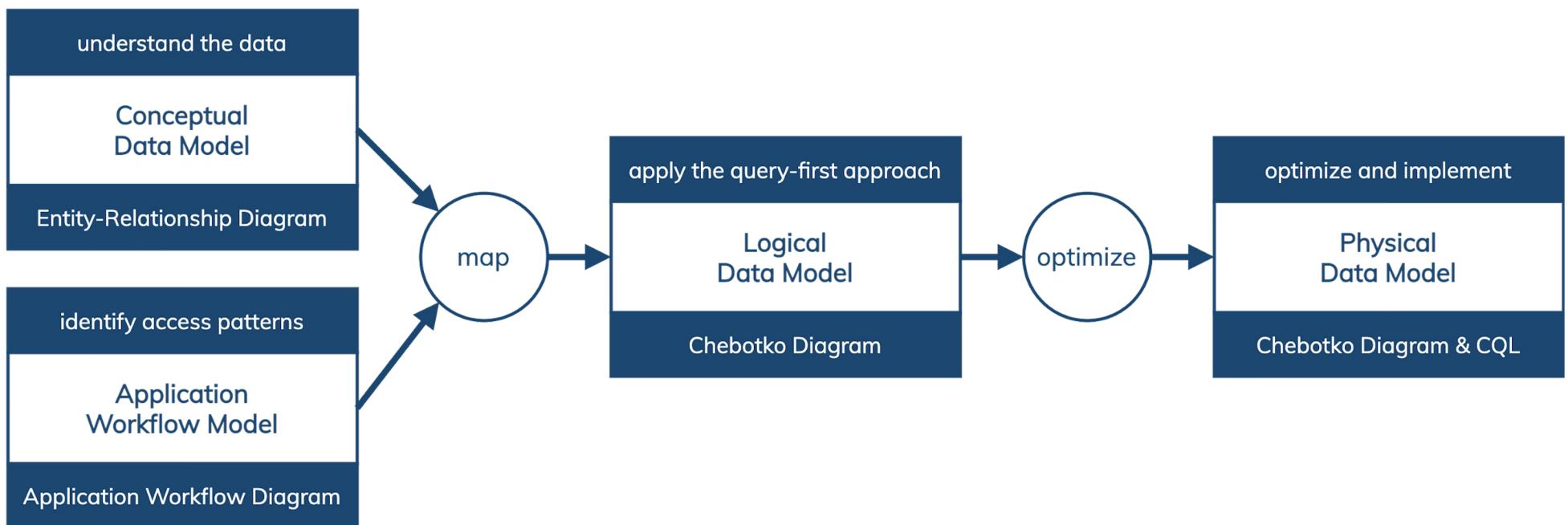
Data Quality: **completeness** **consistency** **accuracy**

Data Access: **queryability** **efficiency** **scalability**

Cassandra Data Modeling Principles

- Know your data
 - Key and cardinality constraints are fundamental to schema design
- Know your queries
 - Queries drive schema design
- Nest data
 - Data nesting is the main data modeling technique
- Duplicate data
 - Better to duplicate than to join

The Methodology: 4 Objectives, 4 Models, 2 Transitions



Mapping Rules

Mapping rule 1: “Entities and relationships”

- Entity and relationship types map to tables

Mapping rule 2: “Equality search attributes”

- Equality search attributes map to the beginning columns of a primary key

Mapping rule 3: “Inequality search attributes”

- Inequality search attributes map to clustering columns

Mapping rule 4: “Ordering attributes”

- Ordering attributes map to clustering columns

Mapping rule 5: “Key attributes”

- Key attributes map to primary key columns

Security

Astra DB for Cloud Native Applications

The Astra DB logo, featuring the word "Astra" in a bold, sans-serif font with a stylized 'A', followed by "DB" in a smaller, regular sans-serif font.



Outline

- Security Overview
- Astra Credentials



Security Overview

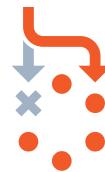
Security features of Astra

Astra DB Security Features



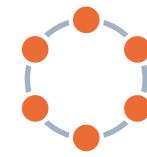
Secured Platform

- Secure shared infrastructure
 - Purpose built for multi tenancy
 - Hardened & continuously pen tested
 - Continuous intrusion detection & prevention
- Data at rest encryption with Bring Your Own Key



Secured Communications

- TLS encrypted communications to all HTTP interfaces
 - Astra GUI
 - DevOps API
- Private Link
- VPC peering
- End to end encryption for data in transit



Secured Access

- IP access list
- SAML based SSO
- Role Based Access Controls (RBAC)
- Organization level tenant separation
- Token based API Access for applications subject to RBAC
- Secure Connect Bundle for CQL access

Security Features 1/2

- **Encryption for Data in motion:** Users connect to Astra via a secure endpoint that provides in-transit encryption via industry-standard mutually authenticated TLS
- **Encryption for Data at Rest:** Persistent storage volumes are encrypted with KMS-managed keys
- **Backup Encryption:** Encrypted with KMS-managed keys in each cloud provider's respective blob store.

Security Features 2/2

- **Astra Login Access (Control Pane):** Built within purpose-hardened AWS accounts. Organization Service Control Policies (SCPs) provide guardrails, actions are logged and used for security monitoring, and the native AWS GuardDuty Intrusion Detection System is enabled and monitored to protect against malicious and unauthorized access.
- **Access to Compute Instance:** All access to production environments is granted by role and monitored, with further restrictions governing access to compute instances running customer clusters. No table-level access is required for troubleshooting, as recovery actions are via API. Responsibilities are separated and logs are audited and used for automated alerts to detect unauthorized activities.

Single Sign On (SSO)

SSO for Astra DB provides:

- Integration with Identity Providers via the SAML standard
- A new layer of authentication at the organization level in Astra DB
- Delegation of authentication and access control to an Identity Provider
- Adherence to password policy requirements enforced by an Identity Provider
- Rapid provisioning of new Astra accounts based on IdP account assignment (JIT provisioning)

The screenshot shows the DataStax Astra web interface. At the top, there's a dark header with the Astra logo, a search bar, and user navigation links. Below the header, the main content area has a left sidebar with 'Galactic Enterprises' selected from a dropdown menu. The sidebar includes links for User Management, Role Management, Token Management, Billing & Payments, and Security Settings. The main content area is titled 'Security Settings' and contains two sections: 'Single Sign-on (SSO)' and 'Audit Logs'. The 'Single Sign-on (SSO)' section has a sub-section titled 'Configure SSO' which explains that SSO enables users to log in through their identity provider. It includes a button to 'Add Identity Provider' and a note that no providers have been added yet. The 'Audit Logs' section includes a 'Download History' link and a button to 'Download last 30 days'.

Private Link

Private Link for Astra DB provides:

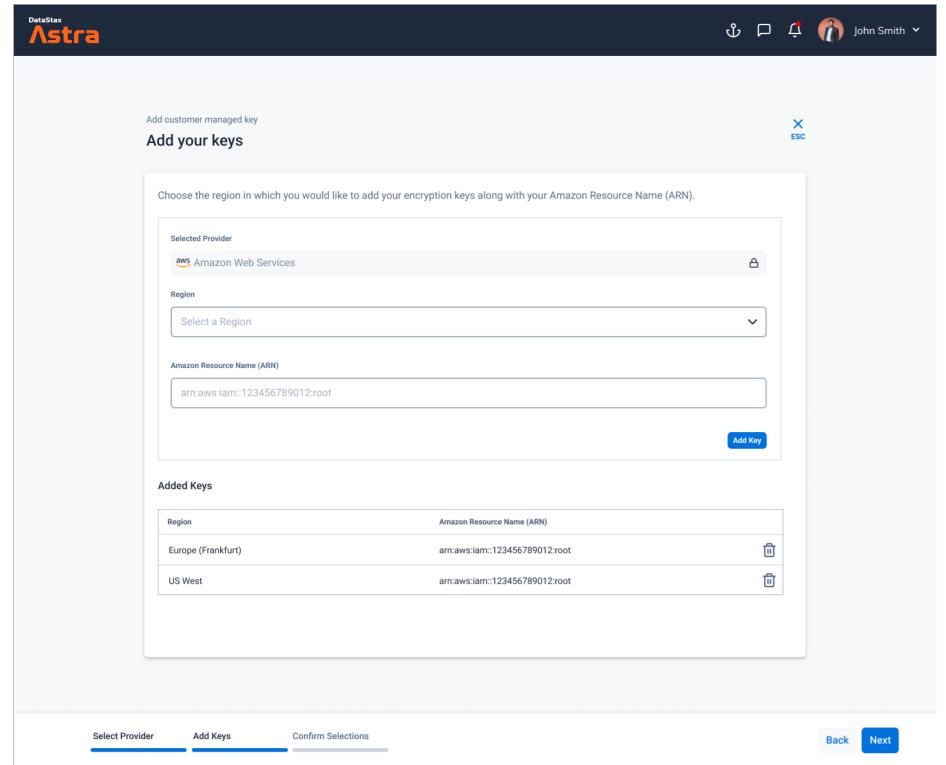
- Secure way to connect services with Astra DB
- Supported on all 3 cloud providers - AWS, Azure and GCP
- Favored alternative to VPC peering
 - Easier to manage and scalable
 - Overcomes the problems CIDR block conflicts
- Traffic remains in the private IP space
- Monitored by the cloud provider
- Possible Reduction in bandwidth cost for data leaving the enterprise VPC
- **Encryption for Data in transit** via a secure endpoint

The screenshot shows the DataStax Astra web interface. On the left, there's a sidebar with 'Current Organization' set to 'DataStax'. It lists databases: 'mars' (green dot), 'saturn' (orange dot), 'prod' (orange dot), 'service' (green dot), and 'ganymede' (green dot). Below this is a 'Sample App Gallery' button. On the right, the main area has a header 'Databases / mars' with tabs for Overview, Health, Connect, CQL Console, Explore, and Settings (which is selected). Under 'Settings', there's a section for 'Application Tokens' with a note about moving token management to organization settings. Below that is a 'Private Endpoint' section with a 'Premium' badge, showing a table with one row for 'aws Amazon Web Services' in 'us-east-2' region, with a cluster ID of 'f7497c6c-d624-4475-6...'. The status is 'Activated'. At the bottom, there's an 'IP Access List' section with a 'Restrict Public Access' toggle turned on, a note about public access being restricted, and a 'Manage Addresses' section with a note about protecting the database. A green toast message at the bottom says 'We've created your private endpoint. Check out our Connection Guide to accept the connection.' There's also a 'Suspend Database' button and a 'DataStax' logo.

Customer Managed Key for Data at Rest Encryption (BYOK)

BYOK for Astra DB provides:

- Support for envelope encryption via integration with AWS KMS
- AWS KMS provides support for key rotation, access monitoring/logging ..
- BYOK is optional, Astra DB supports data at rest encryption by default with DS managed keys- cloud native encryption





Astra Credentials

Authenticating and authorizing Astra connections

Secure Token

- Consists of:
 - clientID
 - clientSecret
 - token
- Use to access Astra database
- Associated with specific user permissions
- Generated at table creation
 - May be created later
 - May create multiple tokens with different permissions

Secure Token

Save your secure token details

Here's your auto-generated token for this database. It's got [the default permissions](#) assigned to it. Store it somewhere safe - it's your key to accessing this database.

! Save your auto-generated token and keep it somewhere safe. You can always [generate a new token](#), but you won't be able to access this one later.

Your Token

```
{  
  "clientID": "hDoyOnQiRzQXuMmOuJWvSonQ"  
  "clientSecret": "2nLS4K8vkHZK+DHQf,lfKZNC2Drb7vAS6M"  
  "token": "AstraCS:hDoyOnQiRzQXuMmOuJWvSonQ:f83d9aab"  
}
```

[Download Token Details ↴](#)

 [secure-token.json \(1KB\)](#)

Permissions (Default Token)

Organization permissions

- Write Token
- Read Token
- View DB
- Terminate DB
- Expand DB
- Create DB
- Add Peering
- Manage Region
- Manage Private Endpoint
- Write IP Access List
- Read IP Access List

Keyspace permissions

- Drop Keyspace
- Modify Keyspace
- Grant Keyspace
- Alter Keyspace
- Authorize Keyspace
- Create Keyspace
- Describe Keyspace

Table permissions (applies to all tables in selected Keyspace)

- Drop Table
- Grant Table
- Create Table
- Alter Table
- Authorize Table
- Select Table
- Modify Table
- Describe Table

API Access

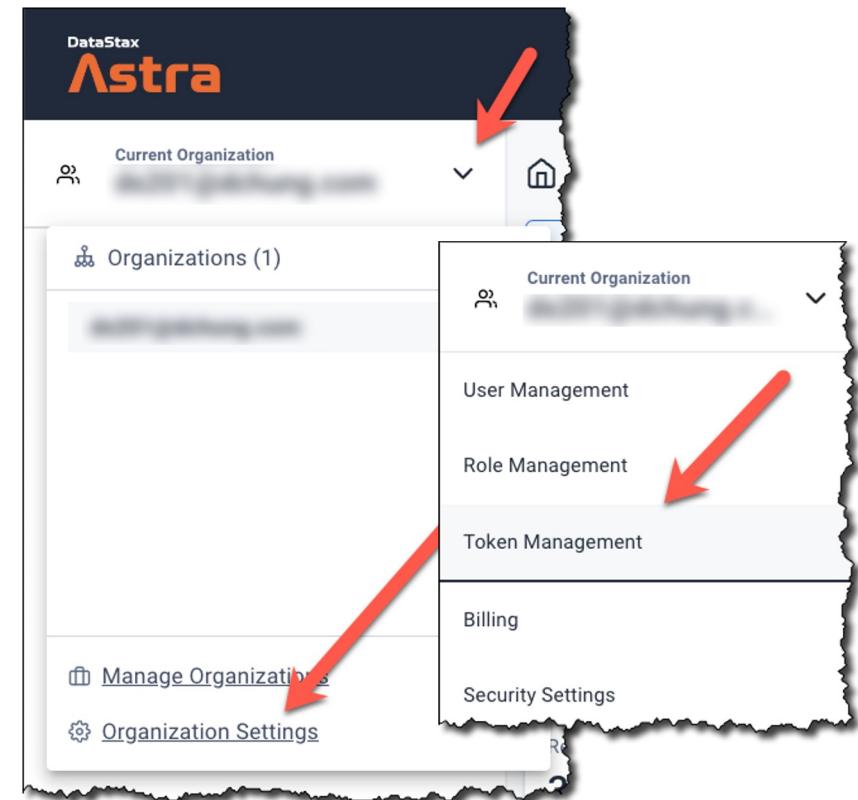
- Access Rest
- Access CQL
- Access Graphql

Best Practices

- Tokens are used for remote access
 - REST, GraphQL, gRPC, drivers, etc.
 - Default token likely has too many permissions
- Principle of *least privilege*
 - Create and use tokens with only the necessary and sufficient permissions required by applications
- Store tokens securely

Generate a Token

- Click on the organization dropdown
- Select *Organization Settings*
- Select *Token Management*



Assign a Role

- Choose a role from the dropdown
- Verify that the role has the necessary permissions
- Download or copy the token

The screenshot shows a user interface titled "Token Management" for generating application tokens. At the top, there's a section for "Application Tokens" with a "Generate a new token" button. Below it, a note says: "Select which role you want to attach to your token. The permissions for that role will be displayed before generating your token to ensure you give your application the right permissions." A "Select Role" dropdown menu is open, showing "API Read/Write User" as the selected option. Next to it is a "Generate Token" button. A large callout box displays the "Role Permissions for API Read/Write User" with the following items:

- Read IP Access List
- Describe Keyspace (All)
- Modify Table
- Access Rest
- View DB
- Describe Keyspace
- Describe Table
- Read User
- Select Table
- Access Graphql

At the bottom of the callout box, a link reads: "For more information about role permissions, see [User permissions](#)".



Hands-on Lab

LAB 19: Generate Credentials

- Generate a token for a database
- Verify that the token has the appropriate permissions