

Introduction

Astra DB for Cloud Native Applications

Astra DB



Outline

- Use Cases
- Features
- Fundamentals
- Getting Started



Astra DB Use Cases

Astra DB 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

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

Cloud Migration
Performance
Availability

Cloud Migration
Self managed operations
Availability

Managed Service
High Performance and scalability
Pay as You Go

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

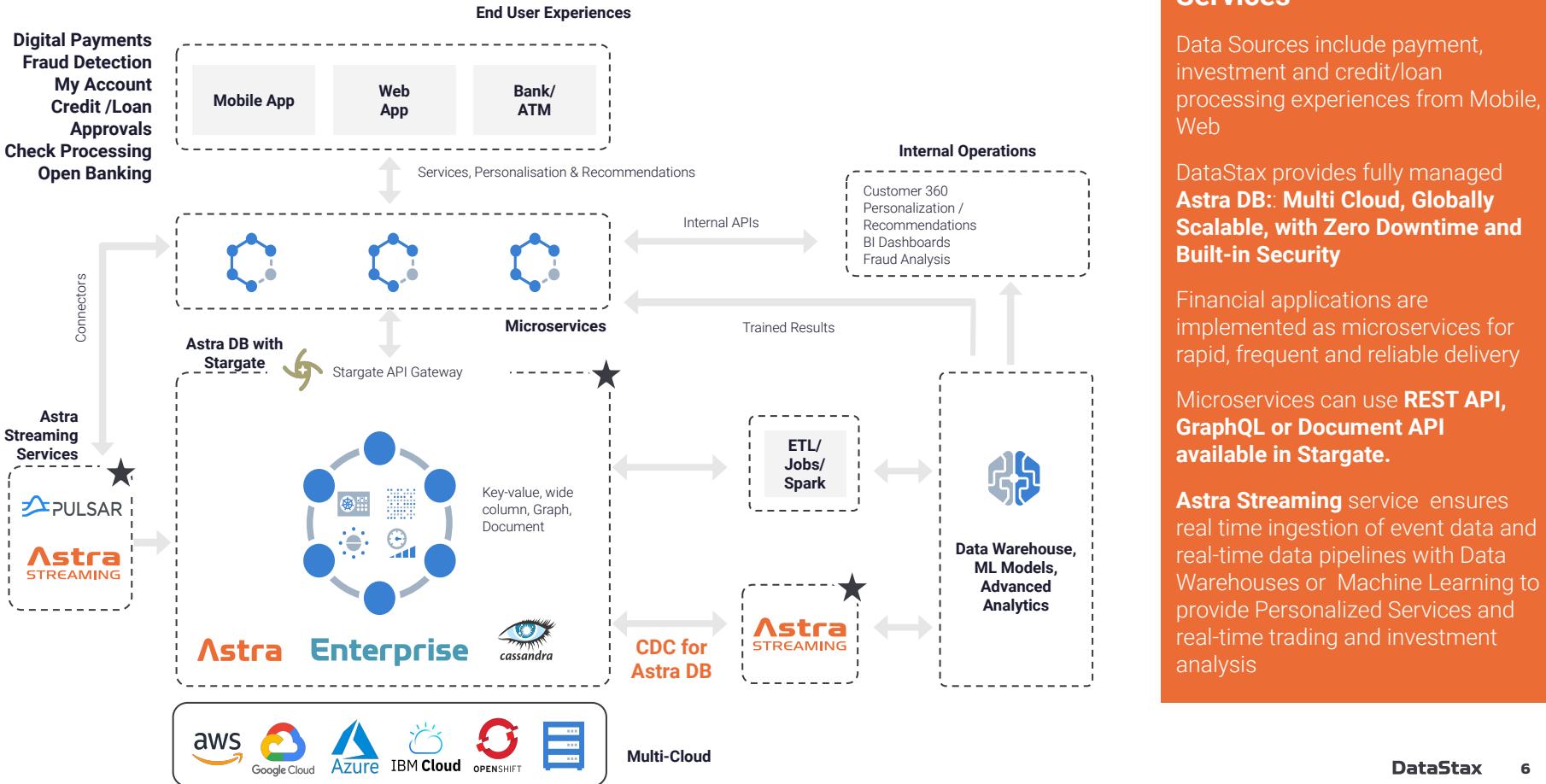
Cloud Migration
Self managed operations
Enterprise grade features not found in OSS C*

Cloud Agnostic
Reduced Operations
Developer Velocity

Astra DB Use Cases



Astra DB In Financial Services



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.

EndowUS

Industry
Finance

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



Enable developer adoption

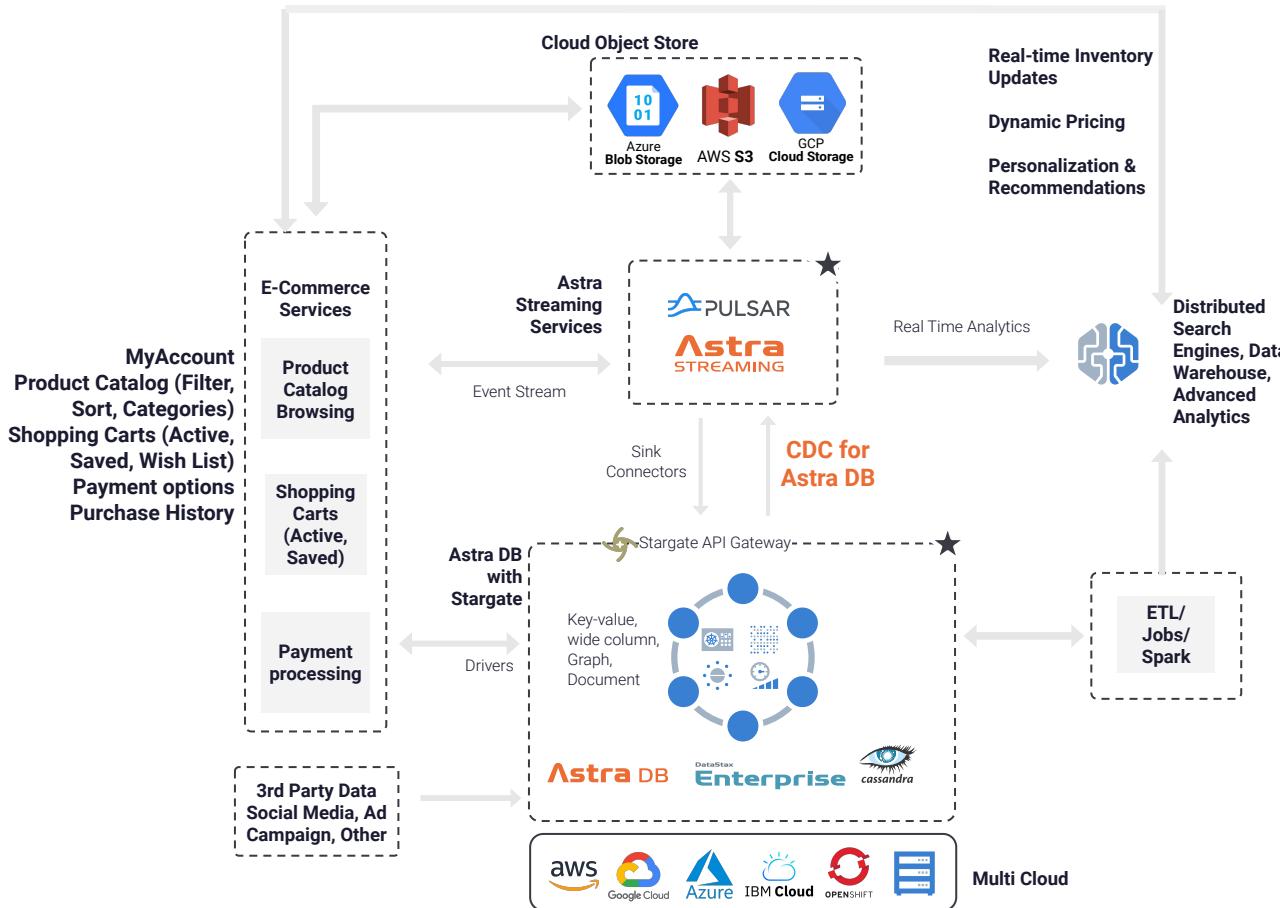


Data Driven

Zero

Operations and
Lock-In solution

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

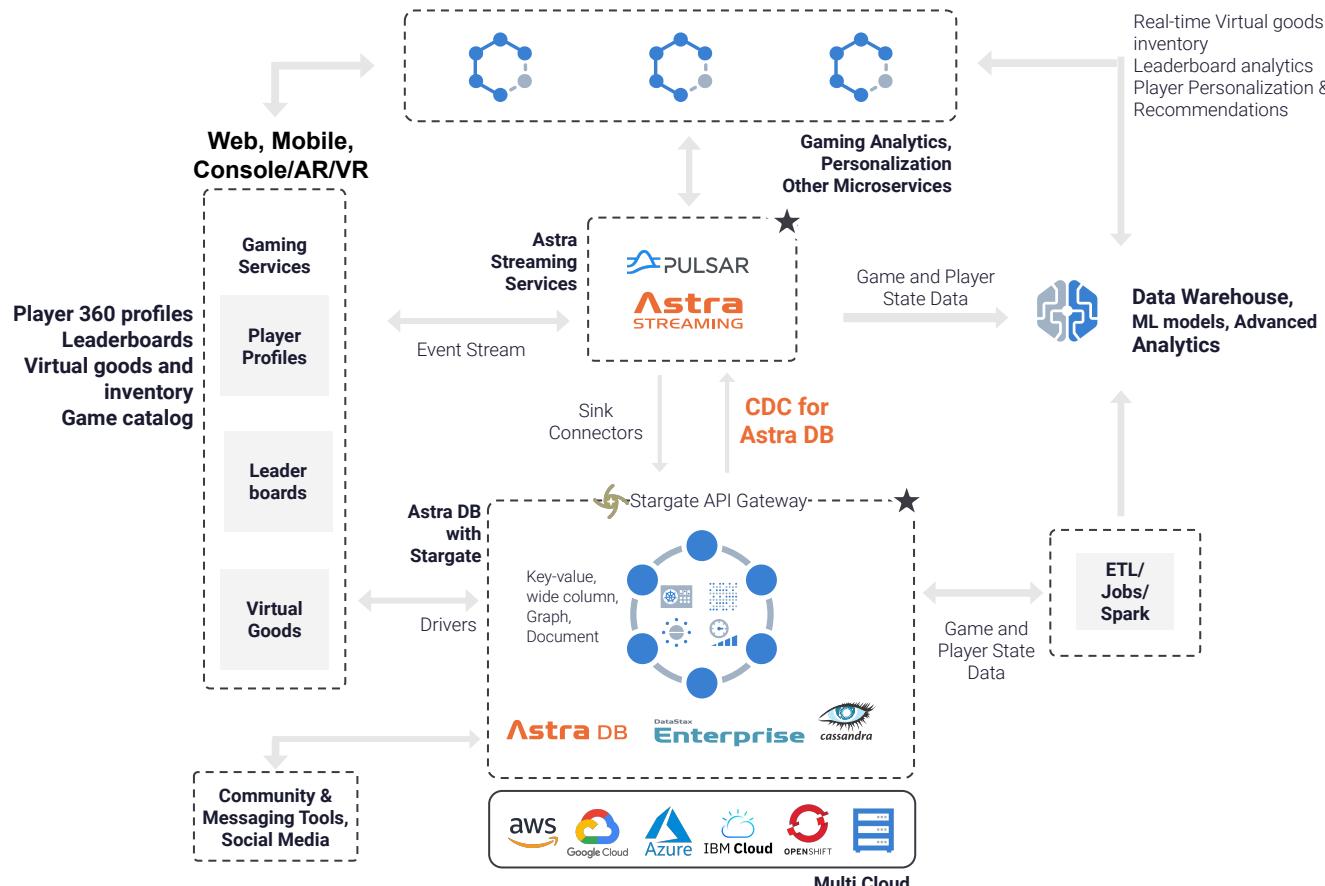


Zero

Millions of
personalized
recommendations

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



24
Astra DB cost savings and flexibility

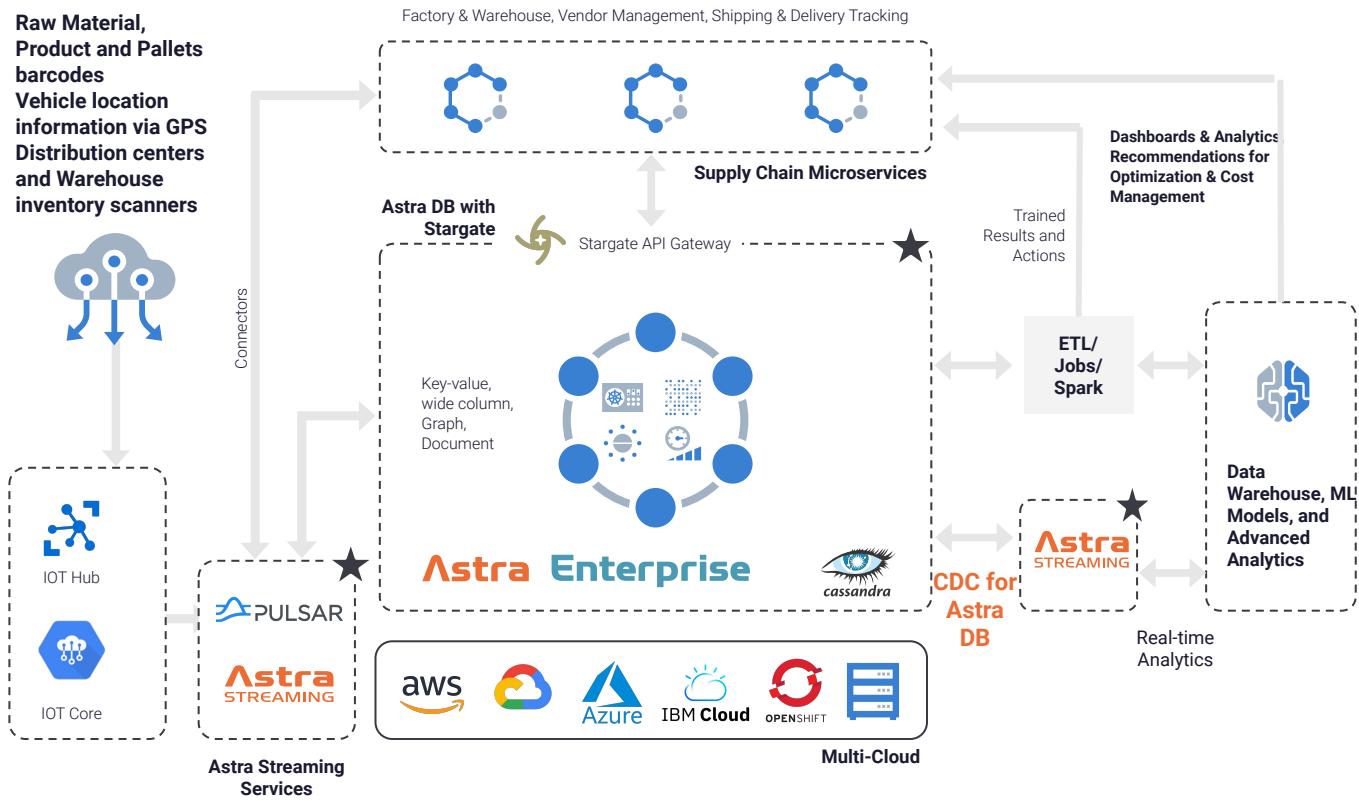


High reliability with scalability



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



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



Astra DB

Any API,
Any Language



Flexible
Data
Model



Column-
Family



Document

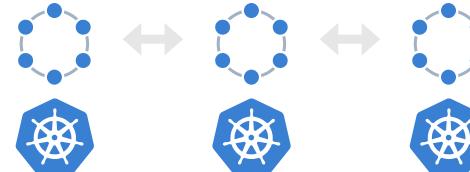


Key-
value



Storage
Attached Index

Cloud-Native
Cassandra
Dynamic
Elasticity

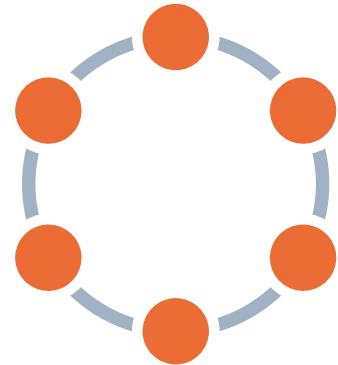


Any Cloud



Serverless

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



Multi-region

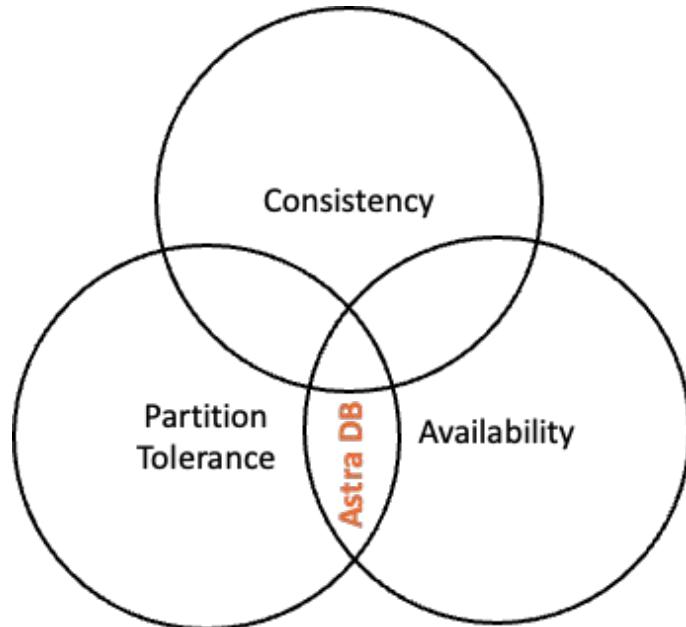
- Deploy across multiple AWS, GCP or Azure regions
- Simple configuration
- Benefits
 - High availability
 - Low latency (clients access data in region close to them)
 - Data sovereignty

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
 - *Tunable* Consistency
- Default data replication across 3 availability zones



Zero-ops

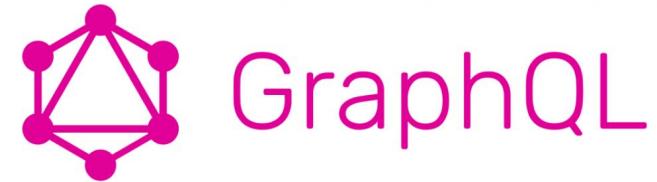
- Astra DB is a *managed service*
- DataStax *auto-magically*
 - Scales elastically
 - Bootstraps new nodes
 - Performs repairs
 - Creates backups

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 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	Normalized data (no duplication)

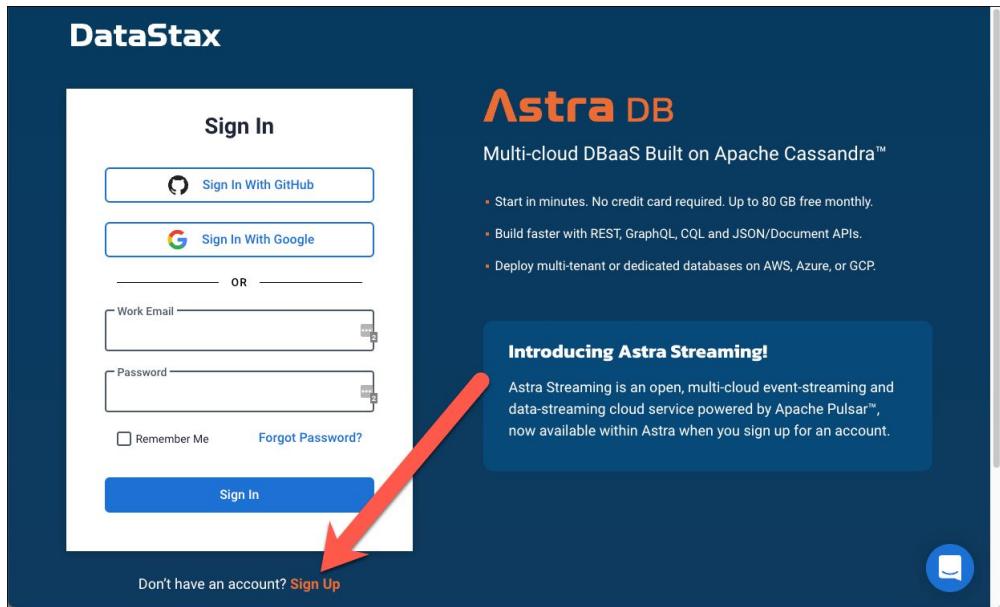


Getting Started

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

Astra Student

Live Chat

Get Started Now 🚀

Create your First Database

Get started in a matter of seconds. Deploy to Amazon Web Services, Google Cloud Platform, or Microsoft Azure. 99% uptime. Regional support. Liftoff!

Create Database

You currently have no databases. Get started by [creating your first one](#).

Streaming Create Stream

Sample App Gallery Looking to get up and running? Get started with a sample app!

Dashboard

Current Plan Free

Looking to use dsbulk for bulk uploads or perform load testing? Request a rate limit increase.

Request An Increase

Want to [upgrade your Astra serverless experience](#)? Add a credit card to get more.

Upgrade

Create Serverless Database

Usage Current Billing Period

Read Requests	Write Requests	Storage Consumed	Data Transfer
0	0	0.00	0.00

You haven't created a database yet. Once you create a database, it will appear here.

Invite Friends, Get Credits You and your friend both get \$25!

Other Resources Documentation Help Center

Live Chat

Sample App Gallery

The screenshot shows the DataStax Astra dashboard with the 'Sample App Gallery' section highlighted. The sidebar includes links for Dashboard, Databases (with 'Create Database'), Streaming, and Sample App Gallery. The main content area displays several sample applications:

- Netflix Clone using GraphQL and Astra DB**: Last Updated: Feb 11, 2022, 2 hours, Beginner. Description: Let's code a Netflix Clone with GraphQL Pagination! Tags: Javascript, GraphQL. Buttons: Learn More, Try Now.
- New Code With Me Series!**: Spring Boot Big Data Application Development. Description: Companion code for the Java Brains 'code with me' series that reads data from DataStax Astra DB. Tags: Java, Spring. Buttons: Learn More, Try Now.
- Building a Goodreads Clone**: BetterReads Spring App. Last Updated: Oct 5, 2021, 180 minutes, Advanced. Description: A workshop where you build a React Native Todo application using DataStax Astra DB. Tags: Javascript, Workshop. Buttons: Learn More, Try Now.
- React Native Todo List, an Astra DB + Netlify Workshop**: Last Updated: Feb 3, 2022, 40 minutes, Intermediate. Description: A workshop where you build a React Native Todo application using DataStax Astra DB. Tags: Javascript, Workshop. Buttons: Learn More, Try Now.
- Including Search**: Can you build a FULLSTACK APP in 30mins? React. Description: A circular graphic featuring a React logo and two people's faces.
- TikTok**: A screenshot of the TikTok mobile app interface showing a user profile and suggested accounts.
- Astra todos**: A screenshot of a mobile application interface titled 'Astra todos' showing a checklist with items like 'Walk my dog', 'Clean the dishes', and '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 DataStax 'Create a Database' interface. The top navigation bar includes 'Live Chat', a user icon, and 'Astra Student'. On the right, there are 'ESC' and 'X' buttons.

1 Enter the Basic Details

Database Name: AS201
Keyspace Name: class

Give it a memorable name - this can't be changed later.

2 Select a Provider and Region

You've got access to 3 free regions in GCP. Unlock all regions by [upgrading to the Pay as you go plan](#).

Google Cloud AWS Amazon Web Services Microsoft Azure

Select an Area: North America 1 of 6 regions selected Europe, Middle East, and Africa 0 of 2 regions selected Asia Pacific 0 of 4 regions selected South America 0 of 1 region selected

Select a Region: Moncks Corner, South Carolina (selected) Ashburn, Virginia The Dalles, Oregon Council Bluffs, Iowa Montreal, Quebec US West (Las Vegas)

Current Plan: Free

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.

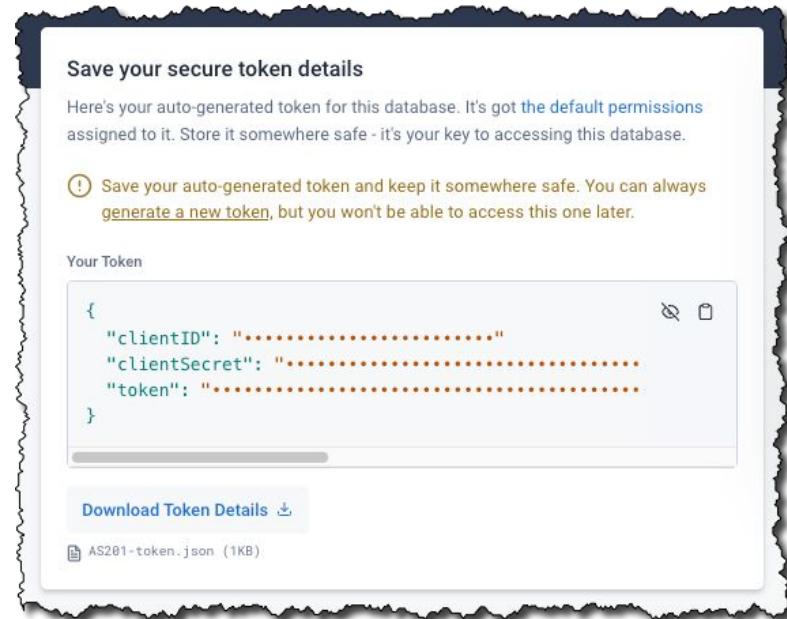
[Learn more](#) about our free accounts.

Create Database

A large red arrow points from the text 'Unlock all regions by [upgrading to the Pay as you go plan](#)' to the 'Select a Region' dropdown menu.

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 interface. At the top, there's a navigation bar with the Astra logo, user information (Current Organization: ds201@dchung.com), and links for Live Chat, Help, and Astra Student. Below the header, the main dashboard for "AS201" is displayed.

Dashboard / AS201

Key metrics shown:

- Read Requests: 0
- Write Requests: 0
- Storage Consumed: 0.00
- Data Transfer: 3.57 KB

Regions

Our Pay as you go plan allows you to add multiple regions. (Unlock Multi-Region)

Provider	Area	Region	Region Name	Datacenter ID	Region Availability
Google Cloud	North America	us-east1	Moncks Corner, South Carolina	6130b58b-56a6-441f-8b45-64f7461f8393-1	Online

Keyspaces

Managing multiple applications? Consider keeping each application in a separate keyspace – whatever a [keyspace](#) is. (Add Keyspace)

Keyspace
class

Other Resources

- Documentation
- Help Center

Invite Friends, Get Credits
You and your friend both get \$25!

Health Tab

DataStax Astra

Current Organization
ds201@dchung.com

Dashboard Databases AS201 Streaming Sample App Gallery

Looking to get up and running?
Get started with a sample app!

Other Resources Documentation Help Center

Invite Friends, Get Credits
You and your friend both get \$25!

Live Chat Astra Student

Load Data Connect

Last 30 minutes 30s

General / DSE Cluster Condensed

Request Details

Requests Combined

Request Errors

Read Failures Read Timeouts Read Unavailable Write Failures Write Timeouts Write Unavailable Range Failures Range Timeouts Range Unavailable

Write Details

Write Latency

P99 Current: P95 Current: P90 Current: P75 Current: P50 Current: Write Operations/sec Current: 0 s

Write Size Distribution

P99 Current: P95 Current: P90 Current: P75 Current: P50 Current: 0 B

Read Details

Read Latency

Range Latency

Connect

The screenshot shows the DataStax Astra dashboard for the organization `ds201@dchung.com`. The left sidebar includes links for Dashboard, Databases (with `AS201` selected), Streaming, Sample App Gallery, and Other Resources (Documentation, Help Center). The main content area is titled "Dashboard / AS201" and features a "Connect" tab. It provides instructions for connecting via REST API, GraphQL API, Document API, and various drivers (Node.js, Python, Java, C++, C#). It also shows how to use Swagger UI and gRPC clients. A sidebar on the right offers "Load Data" and "Connect" options.

Current Organization
ds201@dchung.com

Dashboard / AS201

Connect

You can connect to your database in a few different ways

Use the Astra DB REST API, GraphQL API, Document API or download the secure connect bundle to connect with DataStax drivers.

Connect using an API

Document API

GraphQL API

REST API

Using the Document API to connect to your database

The Document API allows you to store JSON documents in Astra DB without a schema.

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:

```
export ASTRA_DB_ID=6130b58b-56a6-441f-8b45-64f7461fb8393
export ASTRA_DB_REGION=us-east1
export ASTRA_DB_KEYSPACE=class
export ASTRA_DB_APPLICATION_TOKEN=<app_token>
```

3. Use `printenv` to ensure the environment variables were exported.

Launching Swagger UI

Visualize and interact with your database's REST API directly from your web browser with Swagger UI: <https://6130b58b-56a6-441f-8b45-64f7461fb8393-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:

```
curl --request POST \
--url https://$ASTRA_DB_ID-$ASTRA_DB_REGION.$ASTRA_DB_APPLICATION_TOKEN.firebaseioapp.com/$ASTRA_DB_KEYSPACE/collections/hello_docs \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN" \
-H "Content-Type: application/json" \
-d '{"title": "Some Stuff"}'
```

Live Chat

Load Data

Connect

Sample App Gallery

Looking to get up and running?
Get started with a sample app!

Other Resources

Documentation

Help Center

Invite Friends, Get Credits

You and your friend both get \$25!

CQL Console

The screenshot shows the Astra CQL Console interface. The top navigation bar includes the DataStax Astra logo, a "Live Chat" button, a user icon, and "Astra Student" dropdown. The left sidebar shows the current organization "ds201@dchung.com" and navigation links for Dashboard, Databases (with "Create Database" button), AS201, and Streaming (with "Create Stream" button). A "Sample App Gallery" section with a "Get started with a sample app!" button is also present. The main content area is titled "Dashboard / AS201" and contains tabs for Overview, Health, Connect, CQL Console (which is selected), CDC, and Settings. Below the tabs, a section titled "Connect to your CQL Console" provides instructions and a link to the "quick reference guide on CQL". A terminal window shows a successful connection to "cndb" at "cassandra.ingress:9042" using "cqlsh 6.8.0 | Cassandra 4.0.0,6816 | CQL spec 3.4.5 | Native protocol v4". The command "Use HELP for help." and the prompt "token@cqlsh>" are visible. At the bottom of the page, there are "Other Resources" links for Documentation and Help Center, and a "Invite Friends, Get Credits" section.

Current Organization
ds201@dchung.com

Dashboard

Databases [Create Database](#)

AS201

Streaming [Create Stream](#)

Sample App Gallery

Looking to get up and running?
Get started with a sample app!

Connected as ds201@dchung.com.
Connected to cndb at cassandra.ingress:9042.
[cqlsh 6.8.0 | Cassandra 4.0.0,6816 | CQL spec 3.4.5 | Native protocol v4]
Use HELP for help.
token@cqlsh> []

Documentation

Help Center

Invite Friends, Get Credits
You and your friend both get \$25!



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, featuring the word "Astra" in a bold, white, sans-serif font with a stylized 'A', and "DB" in a smaller, white, 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;
```

<code>id</code>	<code>make</code>	<code>model</code>	<code>year</code>
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

UPSETS

- An UPSET 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	1968
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

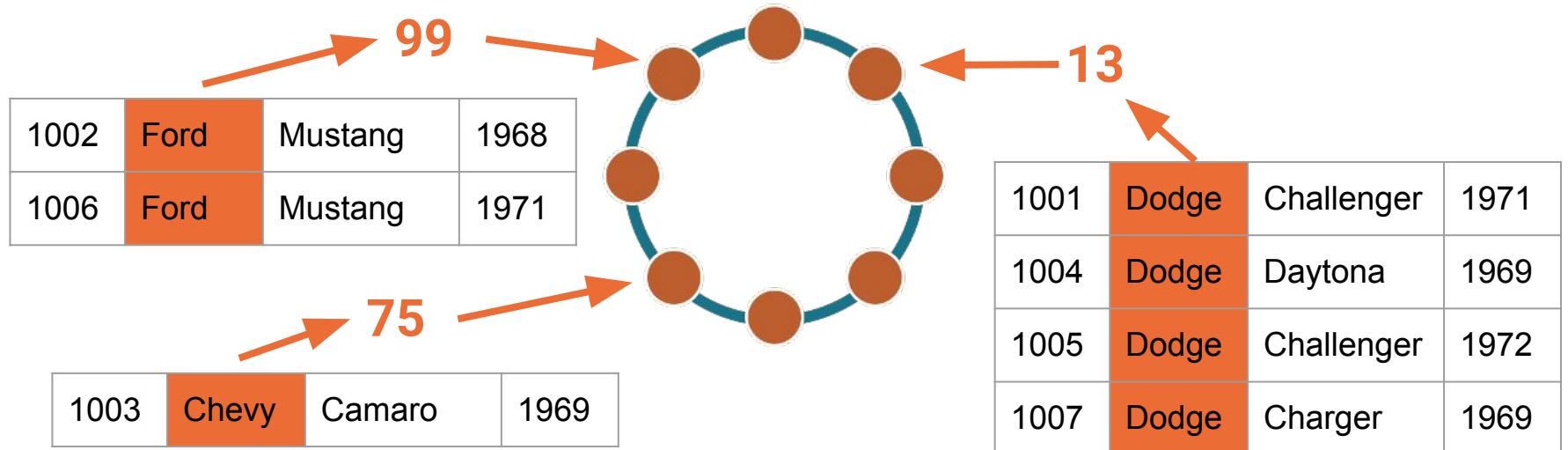
- In Astra (like an RDBMS) the primary key uniquely identifies a row
- Primary keys are made up of two parts:
 - Partition key (the column(s) in the inner parenthesis or first column if no inner parenthesis)
 - Clustering column(s) (the other columns in the primary key)

```
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 for 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

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

Partition Key

Clustering Column (Key)

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)
 - Clustering columns may be used in inequalities

```
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	31000	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 ...
- With the primary key shown below a valid query could use these columns in a *where clause*
 - *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 with 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

- 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, sans-serif font with a stylized 'A', and "DB" in a smaller, regular sans-serif font.



Outline

- Replication
- Consistency



Replication

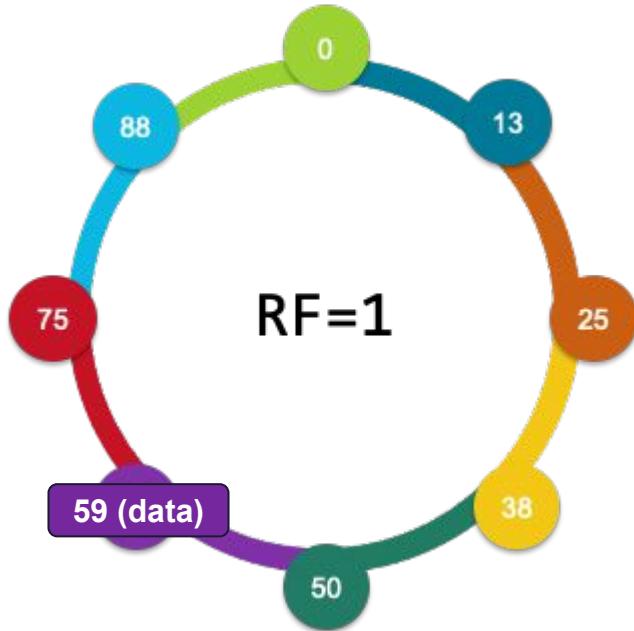
Data replication in Astra DB

Replication

- In Cassandra, data is normally replicated (copied) to multiple nodes in a cluster
- Astra automatically replicates data to three nodes
 - In a multi data center environment that is three nodes per datacenter
 - The three nodes are spread across *Availability Zones* within a *Region*

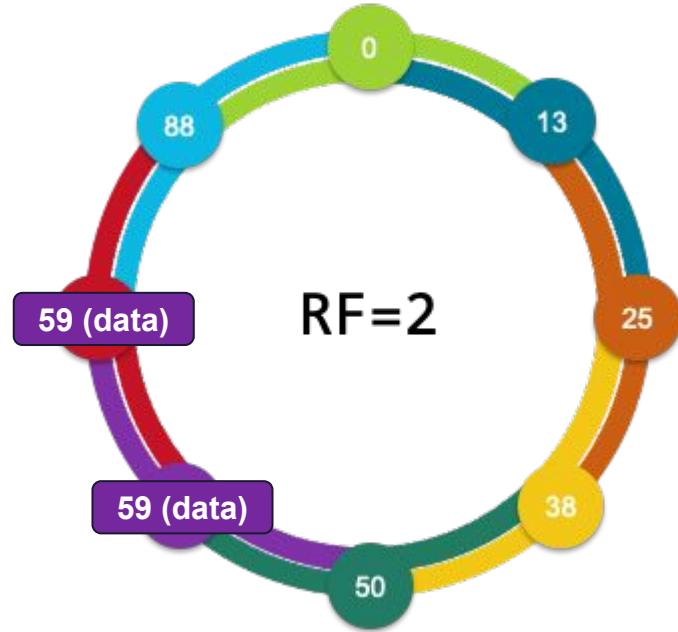
Replication Factor 1

- 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



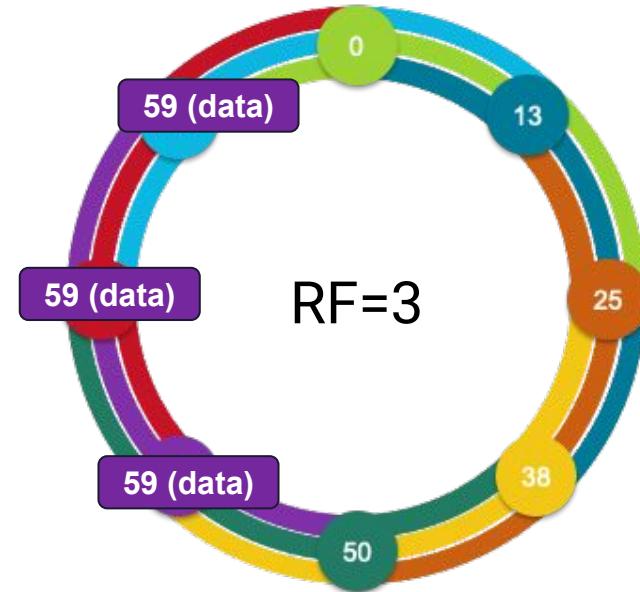
Replication Factor 2

- 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*

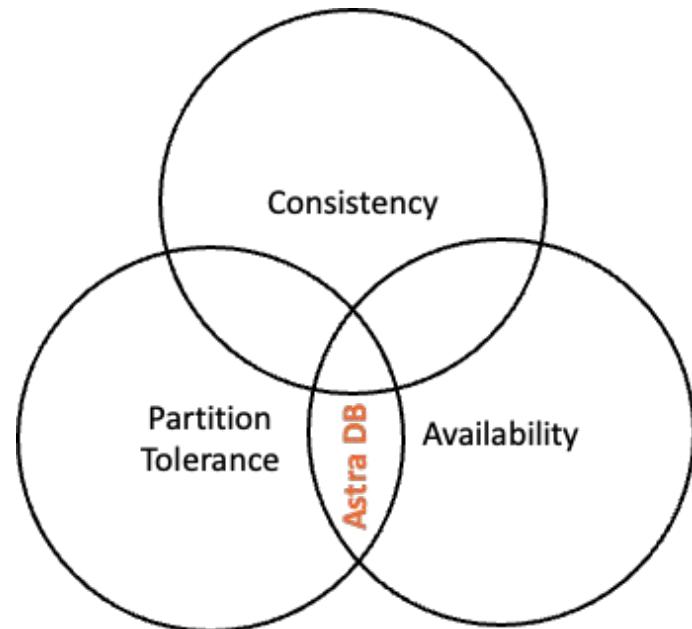


Consistency

Tunable consistency

Consistency

- Definition (simplified): the likelihood that client reads 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 of 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

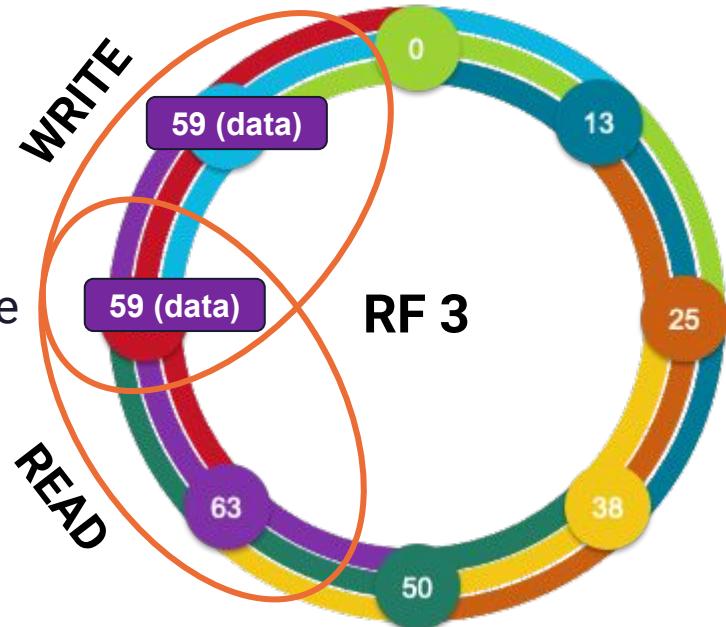
- In OSS Cassandra you need to:
 - Set consistency levels
 - Configure commit log timeouts
 - Define tombstone expiry
 - Perform periodic repairs
 - Do compaction
- Astra DB allows you to set consistency levels
 - No other configuration is required to manage consistency

Immediate Consistency

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

$2 + 2 > 3 \rightarrow$ 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

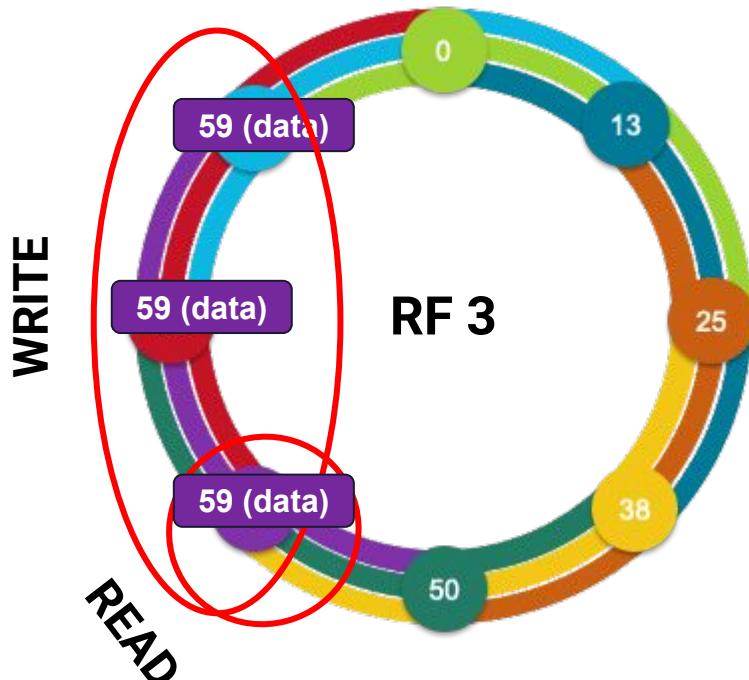


Immediate Consistency

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

$3 + 1 > 3 \rightarrow \text{Immediate Consistency}$

- Write at ALL
 - Guarantees that all of the replicas have the data
- Read at LOCAL_ONE
 - Since all replicas have the data the one chosen for the read will have 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', and "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	31000	1971	red
Chevy	Camaro	60000	1970	blue

Queries in the Cars Table

- Valid queries
 - Find all Chevy Camaros
 - Find all Ford Mustangs with less than 40,000 miles
 - Find the newest Chevy Camaro with less than 31,000 miles
- Invalid queries
 - Find all red Chevy Camaros
 - Find all blue cars

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

Supporting Additional Queries

Question: How do we support both of these queries?

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

Answer: Denormalization

Naming Convention

- RDBMS
 - Table names are nouns (singular or plural)
E.g. car, cars, color, colors
- Astra DB
 - Table names may be nouns (like RDBMS)
E.g. car, cars,
 - Denormalized tables <noun>_by_<partition key>
E.g. **cars_by_make_model**, **cars_by_color**

Denormalize the Cars Table

- Choose the partition key to support a specific query
- Both tables have *exactly* the same data
- Naming conventions includes partition key

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

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

Queries

`cars_by_make_model`

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`

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 WHERE make='Chevy' AND model='Camaro';
```

```
SELECT * FROM cars_by_color WHERE color='red'
```

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	31000	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';
```



A screenshot of the cqlsh command-line interface. The command entered is `SELECT * FROM cars WHERE year>1968 AND year <1970;`. The output shows a table with columns: make, model, miles, year, and color. The data returned is:

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 indexes 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', and "DB" in a smaller, white, sans-serif font.



Outline

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



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']);
```

<code>id</code>	<code>model</code>	<code>states_registered</code>	<code>year</code>
1001	Challenger	['CA', 'NV', 'NM']	1971

Modifying a List

- Append 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;
```

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

- Replace an entire LIST

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

<code>id</code>	<code>model</code>	<code>states_registered</code>	<code>year</code>
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'});
```

id	model	options	year
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;
```

<code>id</code>	<code>model</code>	<code>options</code>	<code>year</code>
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'});
```

id	model	service	year
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;
```

id	model	service	year
1001	Challenger	null	1971

- Update and set to an empty MAP

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

id	model	service	year
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 execu
te 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';
```

<code>id</code>	<code>model</code>	<code>options</code>	<code>year</code>
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 *customer* field of type *owner*

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

INSERTS in a Table with a UDT

```
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});
```

<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: 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,
    year int,
    vin text,
    PRIMARY KEY((collector), make, model, year, vin)
) ;
```

INSERTs

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

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

Collection and UDT in Primary Keys

- Cars partitioned by *collector* (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

Same Query without UDT

- Consider the car collection table without the UDT
- The partition key would be the *first* and *last* name of the collector

```
PRIMARY KEY((first, last), make, model, year, vin)
```

- Querying all of Jay Leno and Jerry Seinfeld's Porsches would now look like

```
SELECT * FROM cars WHERE first IN ('Jerry', Jay)  
    AND last IN ('Leno', 'SEINFELD') AND make = 'Porsche';
```

- This query would find all of Jay Leno and Jerry Seinfeld's Porsches
 - But, it would also look for Jay Seinfeld and Jerry Leno's Porsches
 - The UDT treats owner as a single type and fits this model better



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
 - Tuples are not named
 - Elements are not labeled
- FROZEN by default
 - May be used in partition and clustering keys
 - May be nested

Create a Table with a Tuple

- The `owner` tuple is the partition key for this table

```
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, white, sans-serif font with a stylized 'A', and "DB" in a smaller, white, sans-serif font.



Outline

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



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	1968	14075a-3-bd-2-4702-b-4-05-114-7b-a1	red
Chevy	Camaro	1969	38489fa8-b79c-4d0f-8a96-c62302dec0fd	yellow
Chevy	Camaro	1969	3fd519a2-dfcb-48db-b8f4-64e67f498efe	yellow
FORD	Mustang	1968	88c144b1-677d-4726-9698-c0a002887044	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



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

Shared data in a partition

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)



Tombstones

Deletions in Astra DB

Deletions in Distributed 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 reintroduce data into the system (this data is called a *zombie*)

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', and "DB" in a smaller, regular sans-serif font.



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



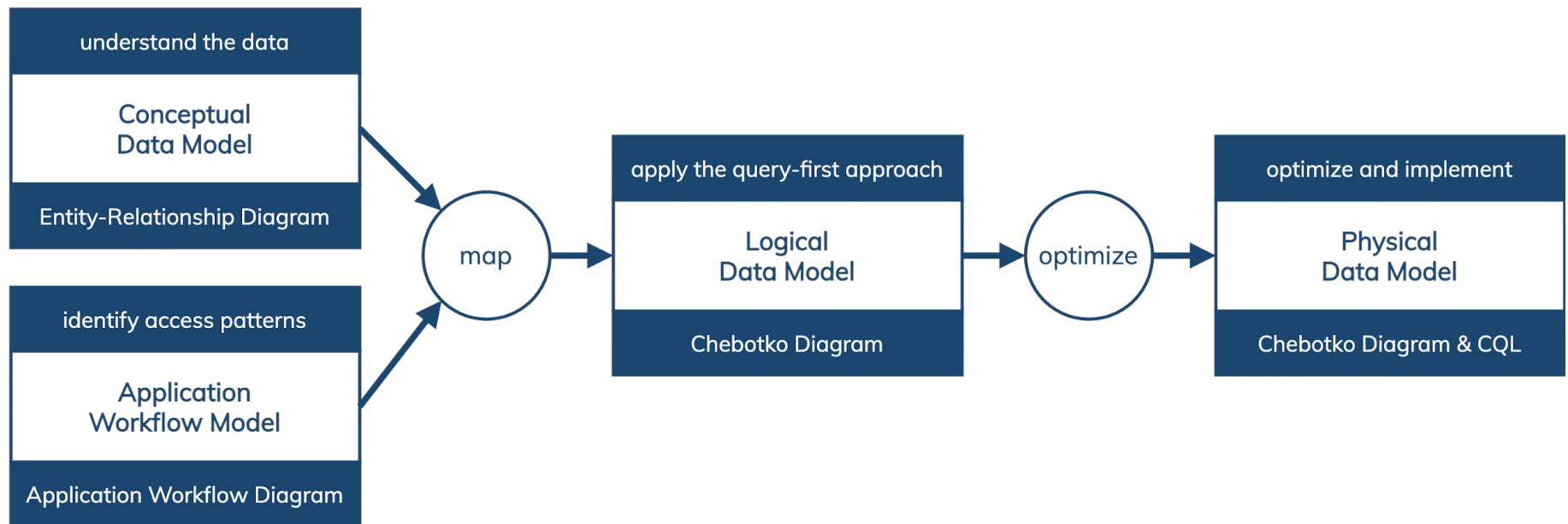
Data Quality: **completeness** **consistency** **accuracy**

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

Astra 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', and "DB" in a smaller, regular sans-serif font.



Outline

- Security Overview
- Astra Credentials



Security Overview

Security features of Astra

Astra DB Security



Private Endpoint

Provides a private channel between your VPC and Astra DB to improve security and latency.



Certification

SOC 2 Type II & PCI will be available soon.



Encryption at Rest

All Astra DB data is encrypted at rest.



Authentication/Authorization

Internal user and role management is available. Supports Role Based Access Controls (RBAC). Implementing SAML 2.0 is on the roadmap.



Encryption in Transit

All the internal and external communications to Astra DB use TLS with mutual authentication enabled.



IP Access Lists

Configure IPs, CIDR blocks that can access Astra DB hosted resources - REST, GraphQL endpoints, Swagger, CQLsh

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.

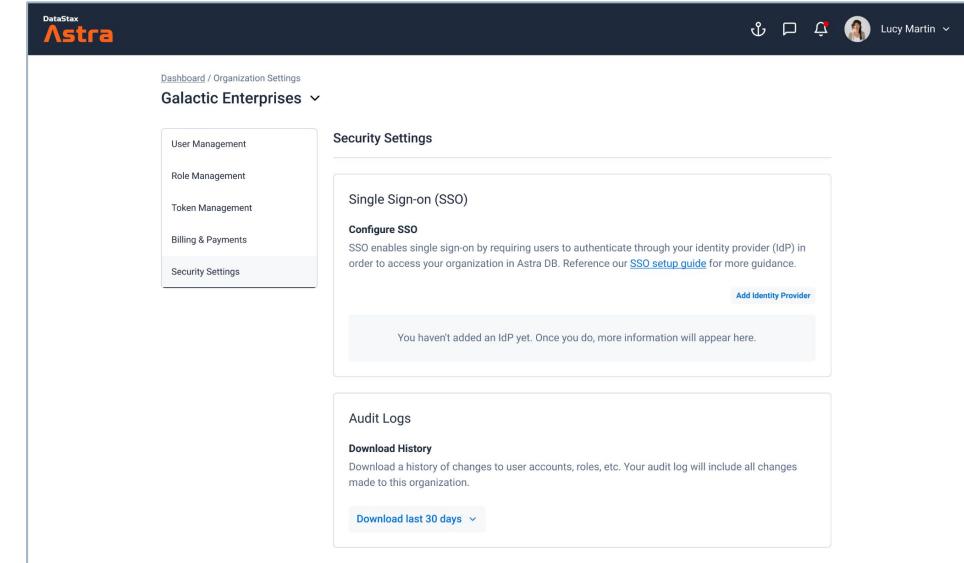
Security Features 2/2

- **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)



Private Link

Private Link for Astra DB provides:

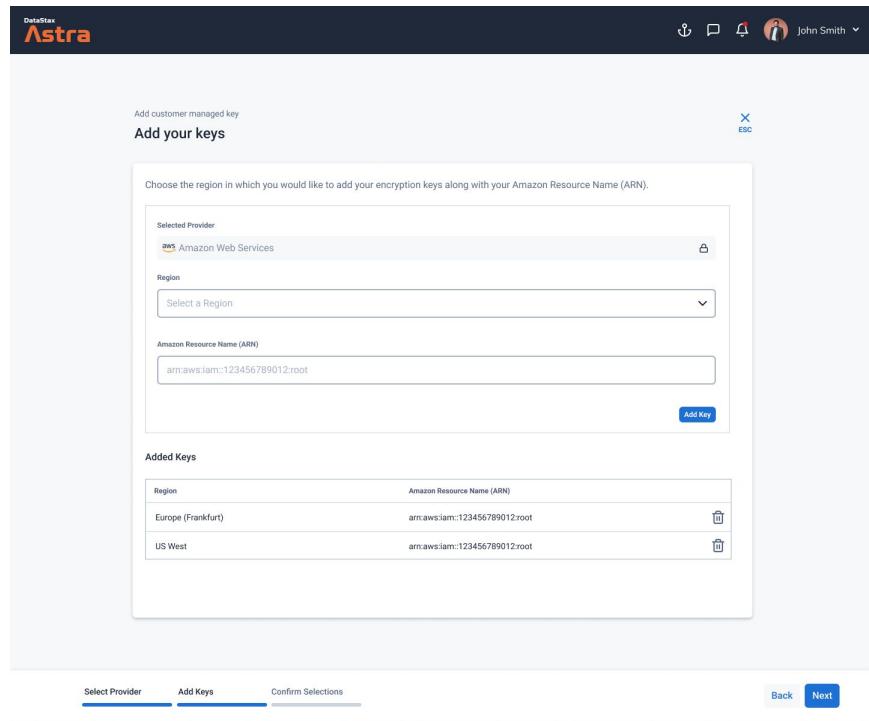
- Secure way to connect services with Astra DB
- Supported on all 3 cloud providers - AWS, Azure and GCP
- Favored alternative to VPC peering
 - Easier to manage and scalable
 - Overcomes the problems CIDR block conflicts
- Traffic remains in the private IP space
- Monitored by the cloud provider
- Possible 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 navigation links: Dashboard, Databases (listing 'mars', 'saturn', 'prod', 'service', 'ganymede'), Create Database, Sample App Gallery, Documentation, Give Feedback, and Get Support. The main content area has tabs for Overview, Health, Connect, CQL Console, Explore, and Settings (which is selected). In the 'Application Tokens' section, a message says "Looking to manage your tokens? We moved application token management to your [Organization settings](#)." Below that is the 'Private Endpoint' section, which is marked as 'Premium'. It lists a single entry: 'aws Amazon Web Services' under 'Provider', 'us-east-2' under 'Region', 'PrivateLink for app connection A' under 'Description', 'f7497c6c-d624-4475-6...' under 'Cluster ID', and 'Activated' under 'Status'. There's also an 'Add Private Endpoint' button. The 'IP Access List' section shows a note: "You haven't set up an access list yet. Once you add IP addresses, they will appear here." At the bottom, a green banner says "We've created your private endpoint. Check out our [Connection Guide](#) to accept the connection." To the right of the banner is a link "If you don't think you need to use your database, you may suspend it. If you suspend your database you" and a "Suspend Database" button. The bottom right corner of the interface has the 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": "hDoyOnQiRzQXuMm0uJWvSonQ"  
  "clientSecret": "2nLS4K8vkHZK+DHQf,lfKZNC2Drb7vAS6M"  
  "token": "AstraCS:hDoyOnQiRzQXuMm0uJWvSonQ: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

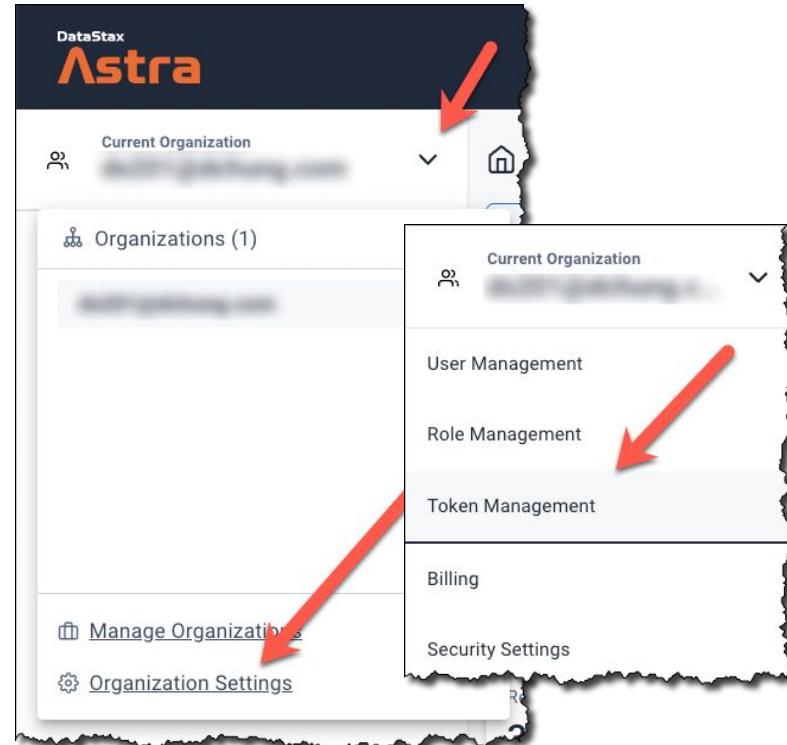
- Access Rest
- Access CQL
- Access Graphql

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

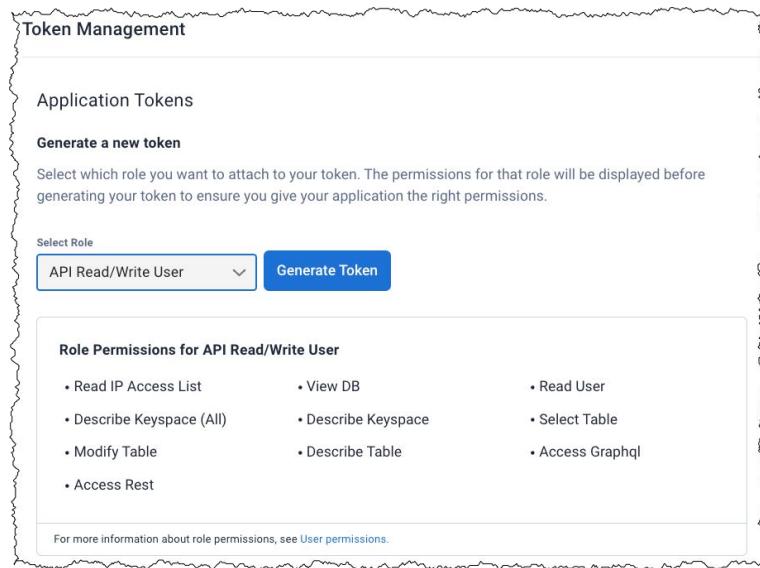
Generate a Token

- 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



Secure Connect Bundle

- Contains information to connect to Astra
 - Used by Drivers and Command Line tools
- Specific to a database and a cloud region
- Access via the Astra Console

Secure Connect Bundle

- Choose the *Connect* tab
- Choose a driver
(Node.js in this case)
- Click *Download Bundle*

The screenshot shows the Astra DB dashboard for cluster AS201. The 'Connect' tab is selected. On the left, there are two sections: 'Connect using an API' (Document API, GraphQL API, REST API) and 'Connect using a driver' (Node.js). The 'Node.js' link is highlighted with a pink box. On the right, a callout box highlights the 'Download your Secure Connect Bundle' button, which is also highlighted with a pink box. A red arrow points from the text 'Click Download Bundle' in the previous slide to this button. Below the callout, instructions for using the Node.js driver are provided.

You can connect to your database in a few different ways

Use the Astra DB REST API, GraphQL API, Document API or download the secure connect bundle to connect with DataStax drivers.

Connect using an API [Learn More](#)

Document API

GraphQL API

REST API

Connect using a driver [Learn More](#)

Node.js >

Download your Secure Connect Bundle
If you have multiple regions, you will have the option to download a bundle for each region.

Download Bundle

Using the Node.js driver to connect to your database

Use the Node.js driver to connect to your database and begin building your own application.

Prerequisites

1. Use the **Download Bundle** button at the top of this page to obtain connection

Bundle Contents

- The bundle contains
 - Certificates
 - Keystores (trust and identity)
- Keystores are encrypted (JKS)
 - The bundle also contains keystore passphrases
 - Keep the bundle secure!!!

secure-connect-as201		Today at 12:36 AM	--	Folder
ca.crt		Aug 7, 2022 at 9:11 PM	1 KB	certificate
cert		Aug 7, 2022 at 9:11 PM	2 KB	Document
cert.pfx		Aug 7, 2022 at 9:11 PM	4 KB	persona...nge file
config.json		Aug 7, 2022 at 9:11 PM	497 bytes	JSON Document
cqlshrc		Aug 7, 2022 at 9:11 PM	206 bytes	Document
identity.jks		Aug 7, 2022 at 9:11 PM	3 KB	Document
key		Aug 7, 2022 at 9:11 PM	2 KB	Document
trustStore.jks		Aug 7, 2022 at 9:11 PM	1 KB	Document



Hands-on Lab

LAB 19: Generate Credentials

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

Data Loading

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', and "DB" in a smaller, white, sans-serif font.



Outline

- DataStax Bulk Loader
- Astra DB Data Loader



DataStax Bulk Loader

Standalone Data Loader

DataStax Bulk Loader (dsbulk)

- Command line tool
 - load data
 - unload data
 - count data
- Formats
 - JSON
 - CSV
- Create table before loading

Install dsbulk

- Download dsbulk
- Untar the file
- Verify the version

```
curl -OL https://downloads.datastax.com/dsbulk/dsbulk-1.9.0.tar.gz  
tar -xzvf dsbulk-1.9.0.tar.gz
```

```
cd dsbulk-1.9.0/bin  
. ./dsbulk --version
```

```
[bin]> ./dsbulk --version  
DataStax Bulk Loader v1.9.0
```

Create the Table

```
CREATE TABLE happiness (
    country_name TEXT,
    regional_indicator TEXT,
    ladder_score DECIMAL,
    gdp_per_capita DECIMAL,
    social_support DECIMAL,
    healthy_life_expectancy DECIMAL,
    generosity DECIMAL,
    PRIMARY KEY((regional_indicator) , country_name)
);
```

Run dsbulk

```
dsbulk-1.9.0/bin/dsbulk load -url <path-to-csv-file> \
-k <keyspace_name> -t <table_name> -b <path-to-secure-connect-bundle> \
-u <client_id> -p <client_secret>
```

```
[AS201]>
[AS201]> dsbulk-1.9.0/bin/dsbulk load -url ./world-happiness-report-2021.csv -k class -t happiness -b ./secure-connect-as201.zip -u dDOdxjRziyeZRkMYrSxzjWMR -p 11jHa5x1M0lrLiqcjh-,02gm7H-joRqCyWUJ8tLxz00e2B3+7cpBL-HQ,evsoMiqCky6izeTmSSQSCHWR.k5BaivUHZ_ET0ajvCe,t6hmMFOeQCjX774,ZGKF+8OZhPZ
Username and password provided but auth provider not specified, inferring PlainTextAuthProvider
A cloud secure connect bundle was provided: ignoring all explicit contact points.
A cloud secure connect bundle was provided and selected operation performs writes: changing default consistency level to LOCAL_QUORUM.
Operation directory: /Users/dohung/Classes/DataStax/as201/logs/LOAD_20220819-061450-140938
total | failed | rows/s | p50ms | p99ms | p999ms | batches
 149 |     0 |    227 | 158.72 | 274.73 | 274.73 |   10.64
Operation LOAD_20220819-061450-140938 completed successfully in less than one second.
Last processed positions can be found in positions.txt
AS201>
```

Data Loaded

```
token@cqlsh:class>
token@cqlsh:class> SELECT * FROM happiness;
```

regional_indicator	country_name	gdp_per_capita	generosity	healthy_life_expectancy	ladder_score	social_support
North America and ANZ	Australia	10.796	0.041	7.265	7.183	0.94
North America and ANZ	Canada	10.776	0.042	7.185	7.103	0.926
North America and ANZ	New Zealand	10.643	0.04	7.355	7.277	0.948
North America and ANZ	United States	11.023	0.049	7.047	6.951	0.92
Middle East and North Africa	Algeria	9.342	0.053	4.991	4.887	0.802
Middle East and North Africa	Bahrain	10.669	0.068	6.779	6.647	0.862
Middle East and North Africa	Egypt	9.367	0.045	4.371	4.283	0.75
Middle East and North Africa	Iran	9.584	0.055	4.828	4.721	0.71
Middle East and North Africa	Iraq	9.24	0.059	4.97	4.854	0.746
Middle East and North Africa	Israel	10.575	0.034	7.224	7.157	0.939
Middle East and North Africa	Jordan	9.182	0.062	4.516	4.395	0.767
Middle East and North Africa	Kuwait	10.817	0.066	6.235	6.106	0.843
Middle East and North Africa	Lebanon	9.626	0.055	4.691	4.584	0.848
Middle East and North Africa	Libya	9.622	0.076	5.558	5.41	0.827
Middle East and North Africa	Morocco	8.903	0.06	5.036	4.918	0.56
Middle East and North Africa	Palestinian Territories	8.485	0.067	4.649	4.517	0.826
Middle East and North Africa	Saudi Arabia	10.743	0.056	6.604	6.494	0.891
Middle East and North Africa	Tunisia	9.266	0.058	4.709	4.596	0.691
Middle East and North Africa	Turkey	10.24	0.046	5.038	4.948	0.822
Middle East and North Africa	United Arab Emirates	11.085	0.039	6.637	6.561	0.844
Middle East and North Africa	Yemen	7.578	0.07	3.794	3.658	0.832
South Asia	Afghanistan	7.695	0.038	2.596	2.523	0.463
South Asia	Bangladesh	8.454	0.046	5.115	5.025	0.693
South Asia	India	8.755	0.026	3.869	3.819	0.603
South Asia	Maldives	9.826	0.072	5.339	5.198	0.913
South Asia	Nepal	8.12	0.07	5.406	5.269	0.774



Astra DB Data Loader

Built in Data Loader

Astra DB Data Loader

- Included in Astra UI
- Load CSV files
 - Limited to files < 40 MB
- Load data from DynamoDB
 - Exported to AWS S3
- Creates Table
 - Specify partition and clustering keys
- Email notification of success or failure

Link on the Dashboard

The screenshot shows the DataStax Astra dashboard for cluster AS201. At the top, there's a navigation bar with links for Overview, Health, Connect, CQL Console, CDC, and Settings. Below the navigation bar, a section titled "Usage For Current Billing Period for AS201" displays four key metrics:

- Read Requests: 607
- Write Requests: 95.6k
- Storage Consumed: 409.74 KB
- Data Transfer: 78.62 MB

A red arrow points to the "Load Data" button in the top right corner of the dashboard header. To the right of the "Load Data" button, there's a "Connect" button and a status indicator showing "Status Active".

Below the usage metrics, there's a "Regions" section with a note about adding multiple regions. It includes a "Unlock Multi-Region" button and a table showing the current region configuration:

Provider	Area	Region	Region Name	Datacenter ID	Region Availability
Google Cloud	North America	us-east1	Moncks Corner, South Carolina	6130b58b-56a6-441f-8b45-64f7461f8393-1	Online

Drag and Drop CSV File

The screenshot shows the Astra Data Loader interface. At the top, there is a navigation bar with the Astra logo, a Live Chat button, and an Astra Student dropdown. Below the navigation bar, the URL is shown as [Dashboard](#) / [AS201](#) / [Data Loader](#). The main area has a heading "1 Add a Dataset" and a radio button labeled "Upload your own dataset" which is selected. Below this is a dashed rectangular area with a red border containing a cloud icon and the text "Release to drop the file here. Only CSV files are supported.". A file named "world-happiness-report-2021.csv" is shown being dropped into this area. To the left of the file is its thumbnail and name, and to the right is a trash bin icon. Below the dashed area, the file name "world-happiness-report-2021.csv" is listed again with a trash bin icon. To the right, the text "Upload Successful" is displayed. At the bottom, there is another radio button labeled "Load a sample dataset" and a blue speech bubble icon.

Astra DB Data Loader

- Name the table
- Make sure the loader has inferred the correct column types

The screenshot shows the Astra Data Preview and Types interface. At the top, there's a navigation bar with the Astra logo, a Live Chat button, and user account information. Below the navigation bar, the title "Data Preview and Types" is displayed, followed by a "Table Name*" input field containing "happiness", which is highlighted with a red arrow. The main area contains a table with columns: country_name, regional_indicator, ladder_score, gdp_per_capita, social_support, healthy_life_expectancy, and generosity. Each column has a dropdown menu for selecting data types (text or decimal). The first two rows of data are shown: Finland (Western Europe) with values 7.842, 10.775, 0.954, 7.904, and 0.032; and Denmark (Western Europe) with values 7.62, 10.933, 0.954, 7.687, and 0.025. A blue circular button with a white square icon is located in the bottom right corner of the preview area.

country_name	regional_indicator	ladder_score	gdp_per_capita	social_support	healthy_life_expectancy	generosity
Finland	Western Europe	7.842	10.775	0.954	7.904	0.032
Denmark	Western Europe	7.62	10.933	0.954	7.687	0.025

Select Key Columns

- Partition Keys
- Clustering Columns

Keys and Clustering

Partition Keys*

X Search ▼

Clustering Columns

X Search X ▼

Back Next



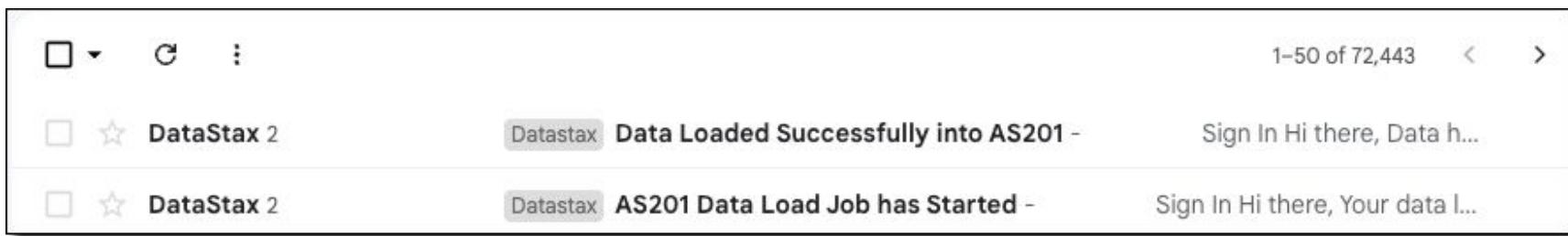
Choose Target

- Database
- Keyspace

The screenshot shows a user interface for configuring a database load. At the top left, there is a blue circular icon with the number '3'. To its right, the text 'Load onto Database' is displayed. Below this, there are two dropdown menus. The first dropdown is labeled 'Target Database*' and contains the value 'AS201'. The second dropdown is labeled 'Target Keyspace*' and contains the value 'class'. At the bottom left of the form, there are two buttons: a light gray 'Back' button and a blue 'Finish' button.

Email Notification

- Upload started
- Upload complete
 - Success or failure



Examine the Table

```
token@cqlsh:class>
token@cqlsh:class> SELECT country_name, generosity FROM happiness;
```

country_name	generosity
Australia	0.041
Canada	0.042
New Zealand	0.04
United States	0.049
Algeria	0.053
Bahrain	0.068
Egypt	0.045
Iran	0.055
Iraq	0.059
Israel	0.034
Jordan	0.062
Kuwait	0.066
Lebanon	0.055



Hands-on Lab

LAB 18: Use the Astra DB Data Loader

- Load data
- Select key columns
- Run queries

Stargate

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', and "DB" in a smaller, white, sans-serif font.



Outline

- Stargate
- Document API
- GraphQL API
- REST API
- gRPC API



Stargate

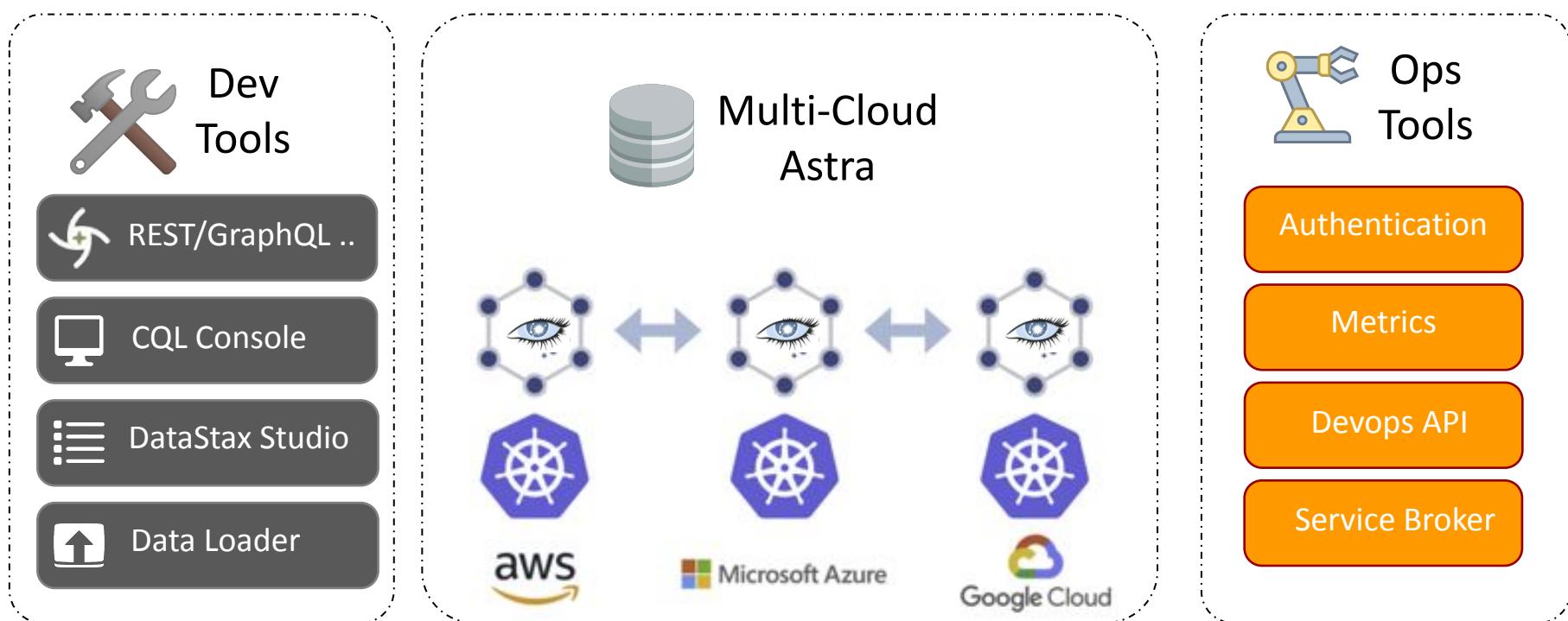
An Open Source Data API Gateway

Stargate

- Open source data API Gateway
 - DB agnostic
 - Integrated with Astra DB
- Supports multiple data API access to data
 - Document
 - GraphQL
 - REST
 - gRPC



Stargate is Embedded in Astra



Compare Astra DB Access Methods

Drivers

Open API



gRPC



{REST}



Cassandra Query Language

gRPC

GraphQL

REST

Document

SQL like Table Model

Structured Data (CQL)

Hierarchy of types and fields

Row based

JSON Documents

Structured Data

Lighter weight

Structured Data

Semi-Structured Data

Key-Value Data

Native driver alternative

Structured Data

Weaker Types

Strong Types

Low query overhead

Key-Value Data

High query overhead

Minimal query overhead

Low query overhead

High query overhead



More Performant

More Flexible



Document API

Schemaless Data in Astra

Document API

- Supports handling JSON documents
 - Loosely coupled
 - Schemaless
- Data *shredded*
 - Stored in *normal* Astra tables in row/column form
 - Accessed as JSON documents



Connection Information

- The Document API uses REST
- To connect you will need:
 - Database ID
 - Cloud Region
 - Keyspace Name
 - Application Token

```
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>
```

Create a Document

- A POST creates a document
- Returns the documentID (UUID)

```
curl --request POST --url \
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astra.datastax.com/api/rest/v2/nam
espaces/$ASTRA_DB_KEYSPACE/collections/cars \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN" \
-H 'Content-Type: application/json' \
-d '{ "vehicle": { "make": "Chevy", "model": "Camaro", "year": 1969 },
      "condition": "good" }'

{
  "documentId": "6076e9e2-33ec-4148-a211-569a3d3f800e"
}
```

The Document

- Contains a *sub-document*: vehicle

```
{  
  "vehicle": {  
    "make": "Chevy",  
    "model": "Camaro",  
    "year": 1969  
  },  
  "condition": "good"  
}
```

Update a Document

- A PUT updates a document
 - Add *price*
- Since this is REST use the document ID in the URI

```
curl --request PUT --url \
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astrastax.com/api/rest/v2/nam
espaces/$ASTRA_DB_KEYSPACE/collections/cars/6076e9e2-33ec-4148-a211-569a3d3f8
00e \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN" \
-H 'Content-Type: application/json' \
-d '{"vehicle": {"make": "Chevy", "model": "Camaro", "year": 1969},
  "condition": "good", "price": 25000}'
```

Get a Specific Document

- A GET with an id retrieves a document

```
curl --request GET --url \
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astra.datastax.com/api/rest/v2/nam
espaces/$ASTRA_DB_KEYSPACE/collections/cars/6076e9e2-33ec-4148-a211-569a3d3f8
00e \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN"
```

```
{"documentId": "6076e9e2-33ec-4148-a211-569a3d3f800e", "data": { "condition": "go
d", "price": 25000, "vehicle": { "make": "Chevy", "model": "Camaro", "year": 1969} }}
```

Create Another Document

- The previous document had price
 - This document does not have a price entry
 - The Document API is schemaless and flexible

```
curl --request POST --url \
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astrastax.com/api/rest/v2/nam
espaces/$ASTRA_DB_KEYSPACE/collections/cars \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN" \
-H 'Content-Type: application/json' \
-d '{ "vehicle": { "make": "Chevy", "model": "Corvette", "year": 1963 },
      "condition": "good" }'

{"documentId": "7e092ac0-2a6f-482f-aecd-954108cdf556"}
```

Get all Documents

- A GET with no id retrieves all documents

```
curl --request GET --url  
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astrastore.com/api/rest/v2/nam  
espaces/$ASTRA_DB_KEYSPACE/collections/cars \  
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN" \  
-H 'Content-Type: application/json'
```

```
{"data": {"7e092ac0-2a6f-482f-aecd-954108cdf556": {"condition": "good", "vehicle":  
{"make": "Chevy", "model": "Corvette", "year": 1963}}, "6076e9e2-33ec-4148-a211-56  
9a3d3f800e": {"condition": "good", "price": 25000, "vehicle": {"make": "Chevy", "mode  
l": "Camaro", "year": 1969}}}}
```

Use a WHERE Clause

- Find all (both) Chevys

```
curl --request GET --url  
https://${ASTRA_DB_ID}-${ASTRA_DB_REGION}.apps.astra.datastax.com/api/rest/v2  
/namespaces/${ASTRA_DB_KEYSPACE}/collections/cars' ?where=\{ "vehicle.make":\{ "  
$eq": "Chevy"\}\}\}' -H 'accept: application/json' -H "X-Cassandra-Token:  
${ASTRA_DB_APPLICATION_TOKEN}"
```

```
{"data": {"7e092ac0-2a6f-482f-aecd-954108cdf556": {"condition": "good", "vehicle":  
{"make": "Chevy", "model": "Corvette", "year": 1963}}, "6076e9e2-33ec-4148-a211-56  
9a3d3f800e": {"condition": "good", "price": 25000, "vehicle": {"make": "Chevy", "mode  
l": "Camaro", "year": 1969}}}}
```

Delete a Document

- A DELETE deletes a document

```
curl --request DELETE -- url \
https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astra.datastax.com/api/rest/v2/nam
espaces/$ASTRA_DB_KEYSPACE/collections/cars/7e092ac0-2a6f-482f-aecd-954108cdf
556 \
-H "X-Cassandra-Token: $ASTRA_DB_APPLICATION_TOKEN"
```

Swagger and the Document API

- The Document API can also be accessed using the Swagger UI
- Swagger is a Web interface to access HTTP based APIs
 - Exposes all the functions of the API
 - Has the connection information already configured

Accessing the Swagger UI

- Select the Connect tab
- Choose the Document API
- Follow the Swagger UI link

The screenshot shows the DataStax Astra web interface. At the top, there's a navigation bar with 'Live Chat', a search icon, a help icon, and a user profile 'Astra Student'. Below the navigation bar, the 'Connect' tab is selected, highlighted by a red box. On the left, a sidebar shows 'Current Organization' as 'ds201@dchung.com'. Under 'Databases', there are three entries: 'AS201' (selected), 'secure', and 'graphql'. Other menu items include 'Documentation' and 'Help Center'. A promotional message at the bottom left says 'Invite Friends, Get Credits' with a note about getting \$25. The main content area has a heading 'You can connect to your database in a few different ways' and instructions for using the REST API, GraphQL API, or Document API. A 'Learn More' link is available. Under 'Connect using an API', the 'Document API' button is highlighted with a red box. Below it, another section titled 'Using the Document API to connect to your database' describes how it allows storing JSON documents in Astra DB without a schema. A note at the bottom says '3. Use printenv to ensure the environment variables were exported.' A third red box highlights the 'Launching Swagger UI' section, which provides a URL: <https://6130b5bb-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com/api/rest/swagger-ui/>. There's also a 'Write a document' button.

Operations Exposed in Swagger

The screenshot shows the Swagger UI interface for a REST API. The top navigation bar includes the Swagger logo, a search bar containing '/swagger.json', and an 'Explore' button. The main content area is titled 'documents' and lists various API operations:

- GET /v2/namespaces/{namespace-id}/collections** List collections in namespace
- POST /v2/namespaces/{namespace-id}/collections** Create a new empty collection in a namespace
- GET /v2/namespaces/{namespace-id}/collections/{collection-id}** Search documents in a collection
- POST /v2/namespaces/{namespace-id}/collections/{collection-id}** Create a new document
- DELETE /v2/namespaces/{namespace-id}/collections/{collection-id}** Delete a collection in a namespace
- POST /v2/namespaces/{namespace-id}/collections/{collection-id}/upgrade** Upgrade a collection in a namespace
- GET /v2/namespaces/{namespace-id}/collections/{collection-id}/json-schema** Get a JSON schema from a collection
- PUT /v2/namespaces/{namespace-id}/collections/{collection-id}/json-schema** Assign a JSON schema to a collection. This will overwrite any schema that already exists.
- GET /v2/schemas/namespaces** Get all namespaces
- POST /v2/schemas/namespaces** Create a namespace
- GET /v2/schemas/namespaces/{namespace-id}/functions** View all built-in namespace functions

Choose an Operation

- Select an operation
- Click the *Try it out* button
 - Configure the request

The screenshot shows a REST API endpoint for creating a collection in a namespace. The endpoint is `POST /v2/namespaces/{namespace-id}/collections`. The description is "Create a new empty collection in a namespace". A red arrow points to the "Try it out" button in the top right corner of the interface.

Name	Description
X-Cassandra-Token <small>* required</small>	The token returned from the authorization endpoint. Use this token in each request. string (header) Default value : eyJhbGciOiJSUzI1NiIsInR5cClgOiAiSldUliwia2IkIiA6ICJsYmJUcDRObzdfUHBjLUXOWFZJbUFmY1pWQWs3TFNwdnY5Z0RFVF80M2Zrln0.eyJqdGkiOiIxZjlyNDNIYS0wYTc3LT

Populate the Request

POST /v2/namespaces/{namespace-id}/collections Create a new empty collection in a namespace

Parameters

Name Description

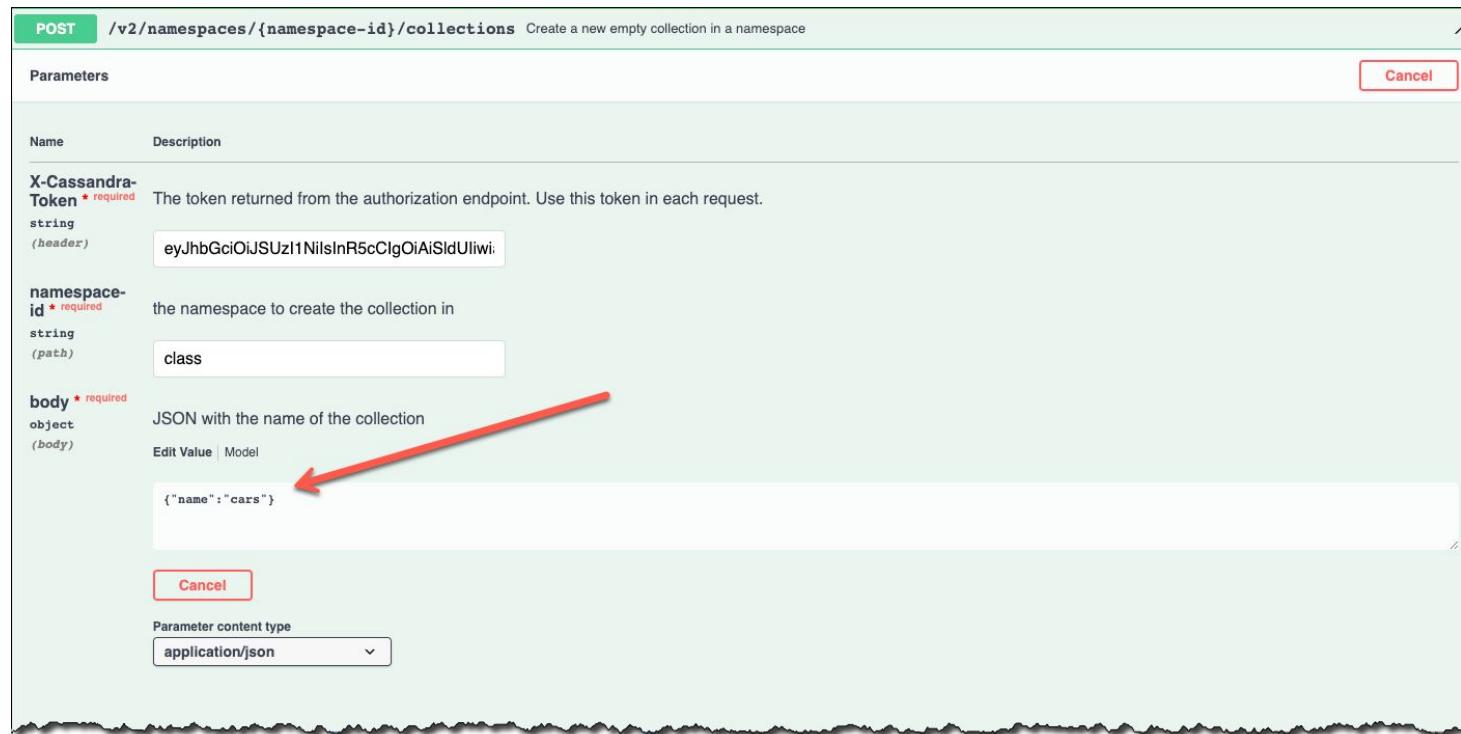
X-Cassandra-Token * required The token returned from the authorization endpoint. Use this token in each request.
string
(header)
eyJhbGciOiJSUzI1NlslR5cClgOiAiSlldUiwi.

namespace-id * required the namespace to create the collection in
string
(path)
class

body * required JSON with the name of the collection
object
(body)
Edit Value | Model
{"name": "cars"}


Cancel

Parameter content type
application/json



Execute the Request

- Click *Execute*
 - You should see then HTTP Response code 201 Success

The screenshot shows a REST API testing interface with the following details:

- Execute** and **Clear** buttons at the top.
- Responses** section on the left.
- Response content type**: application/json
- Curl** command:

```
curl -X 'POST' \
  'https://6130b58b-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com/api/rest/v2/namespaces/class/collections' \
-H 'accept: application/json' \
-H 'X-Cassandra-Token: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwid2lkIiA6ICJyMjUCDR0bzdfUHBjLJLxOWFZjbUFmY1pWQWs3TFNwdnYSZ0RFVF80M2ZrIn0.eyJqdGkiOiIxZjIyNDIyS0wYTC3LTQxY2MtYjoxNi02NDgxZmEy0d' \
-H 'Content-Type: application/json' \
-d '{"name": "cars"}'
```
- Request URL**: https://6130b58b-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com/api/rest/v2/namespaces/class/collections
- Server response** section:
 - Code**: 201 (highlighted with a red arrow)
 - Details** button
 - Response headers**:
 - access-control-allow-credentials: true
 - access-control-allow-origin: https://6130b58b-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com
 - access-control-expose-headers: Date
 - Date: Fri, 19 Aug 2022 07:59:57 GMT
 - Vary: Origin
- Responses** table:

Code	Description
201	Created



Hands-on Lab



LAB 19: Document API

- Create some documents
- Run queries to retrieve documents

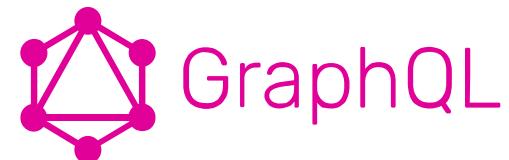


GraphQL API

Graph data for Astra DB

GraphQL API

- Open source data query and manipulation language for APIs
- Developed at Facebook
- Clients define structure of data to be returned
 - Strongly typed
 - Return only data required (not complete *documents or rows*)



GraphQL vs REST

GraphQL	REST
Endpoints for DDL and DML	Individual endpoints for resources
Specify amount of data to receive	API definition pre-defines amount of data returned
HTTP POST action defined in payload	HTTP verbs to define actions

URLs

- Endpoint URLs begin with:

`https://$ASTRA_DB_ID-$ASTRA_DB_REGION.apps.astra.datastax.com`

- DDL URLs end with
`/api/graphql-schema`
- DML URLs end with
`/api/graphql/@ASTRA_DB_KEYSPACE`

Create a Table

```
mutation {
  cars: createTable(
    keyspaceName: "class"
    tableName: "cars"
    partitionKeys: [
      { name: "make", type: { basic: TEXT } },
      { name: "model", type: { basic: TEXT } }
    ]
    clusteringKeys: [{ name: "id", type: { basic: INT }, order: "ASC" }]
    values: [{ name: "color", type: { basic: TEXT } }]
  )
}
```

GraphQL DDL API

The screenshot shows a GraphQL playground interface with the following details:

- Header:** graphql-schema (selected), graphql, Settings, +
- URL:** https://b4c8a8b6-3ef8-421e-965a-eb73b4aa6c48-us-east1.apps.astrastax.com/api/graphql-schema
- Buttons:** PRETTIFY, HISTORY, COPY CURL, SHARE PLAYGROUND
- Left Panel (Query Editor):**

```
1 v query GetTables {  
2 v   keyspace(name: "class") {  
3   name  
4 v     tables {  
5     name  
6 v       columns {  
7       name  
8       kind  
9 v         type {  
10        basic  
11        info {  
12          name  
13        }  
14      }  
15    }  
16  }  
17}  
18 }
```
- Right Panel (Results):** A large JSON object representing the response to the query. It includes fields for data, keyspace, and tables, with detailed information about each table's columns and types.
- Side Navigation:** DOCS, SCHEMA
- Bottom Navigation:** QUERY VARIABLES, HTTP HEADERS (1), TRACING

GraphQL DDL API

The screenshot shows a GraphQL playground interface with the following details:

- Header:** graphql-schema (selected), graphql, Settings, +
- URL:** https://b4c8a8b6-3ef8-421e-965a-eb73b4aa6c48-us-east1.apps.astradb.datastax.com/api/graphql-schema
- Buttons:** PRETTIFY, HISTORY, COPY CURL, SHARE PLAYGROUND
- Left Panel (Query Editor):**

```
1 v query GetTables {  
2 v   keyspace(name: "class") {  
3 v     tables {  
4 v       name  
5 v       columns {  
6 v         name  
7 v       }  
8 v     }  
9 v   }  
10 }
```
- Right Panel (Results):** A large JSON tree view showing the response to the query. The results include a "data" object with a "keyspace" object containing a "tables" array. Each table object has a "name" field (e.g., "cars") and a "columns" array. Each column object has a "name" field (e.g., "make", "model", "id", "color").
- Side Panels:** SCHEMA (selected), DOCS
- Bottom Navigation:** QUERY VARIABLES, HTTP HEADERS (1), TRACING

Insert Data with DML API

```
mutation {
  car1001: insertcars(
    value: { make: "Dodge", model: "Challenger", id:1001, color:"yellow" }
  ) { value { id } }
  car1002: insertcars(
    value: { make: "Ford", model: "Mustang", id:1002, color:"green" }
  ) { value { id } }
  car1003: insertcars(
    value: { make: "Ford", model: "Mustang", id:1003, color:"blue" }
  ) { value { id } }
}
```

make	model	id	color
Dodge	Challenger	1001	yellow
Ford	Mustang	1002	green
Ford	Mustang	1003	blue

Query Data with DML API

```
query mustangs {  
    cars(value: { make: "Ford", model: "Mustang" }) { values { color } }  
}  
  
{  
    "data": { "cars": { "values": [ {"color": "green" },  
                                {"color": "blue"} ] } }  
}
```

Update Data with DML API

```
mutation {
  aaa: updatecars(
    value: { make: "Ford", model: "Mustang", id:1003, color:"orange" }
  ) { value { id color } }
}

{ "data": { "aaa": { "value": { "id": 1003, "color": "orange" } } } }
```

Delete Data with DML API

```
mutation deleteCars {  
  mustangs: deletecars(  
    value: { make: "Ford", model:"Mustang" }  
  ) { value { id } } }
```

make	model	id	color
Dodge	Challenger	1001	yellow

GraphQL Playground

- Web based GUI for executing GraphQL command
 - Integrated into Astra UI
- Click on *Connect* tab
- Select *GraphQL API*
- Clink on the link

The GraphQL API allows you to interact with your data using GraphQL types, queries, and mutations. For every table in the database, a series of GraphQL objects are generated, along with queries and mutations that allow you to search and modify the table data.

Prerequisites

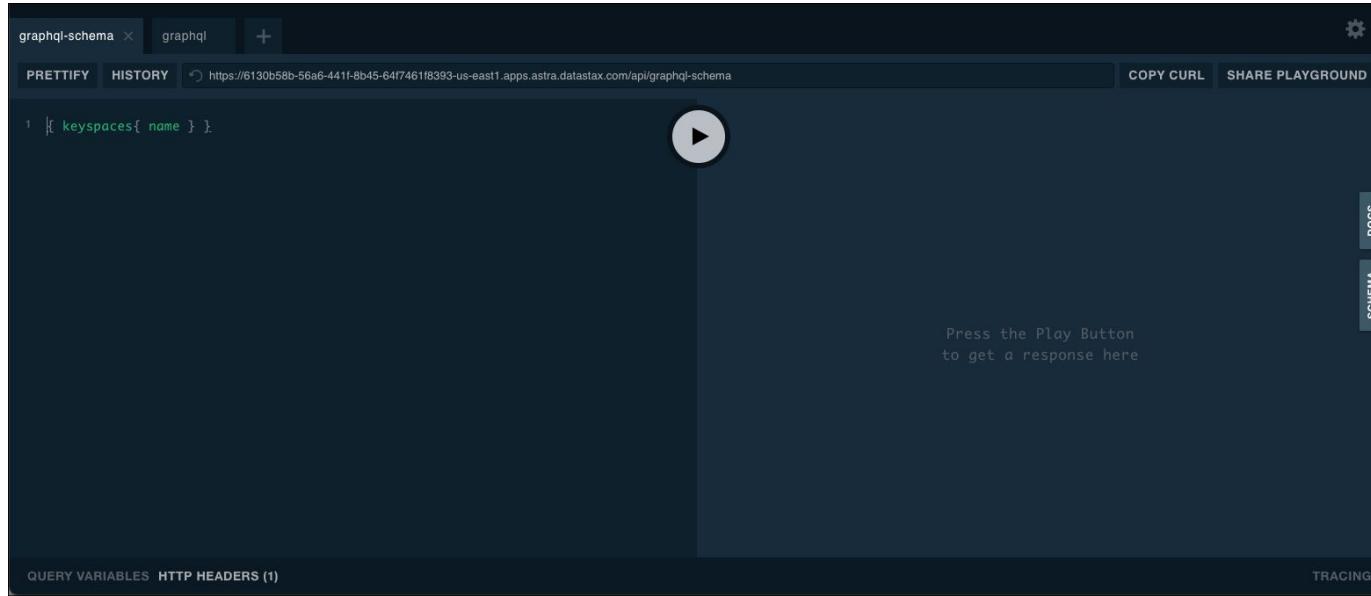
1. An Application Token (create a new one [here](#)) with the appropriate role set (API Admin User is needed for example below).

Launching GraphQL Playground

1. Open the GraphQL Playground URL at <https://6130b58b-56a6-441f-8b45-64f7461f8393-us-east1.apps.astra.datastax.com/api/playground>
2. Add your **Token** from your Application Token to the HTTP HEADERS section at the bottom of the GraphQL Playground window

GraphQL Playground

- Enter GraphQL commands on the left
- Responses appear on the right





Hands-on Lab

LAB 20: GraphQL API

- Create a table with the GraphQL API
- Populate the table and retrieve data



REST API

RESTful data access

REST API

- REpresentational State Transfer
- Roy Fielding of Apache Foundation
 - First mentioned in doctoral thesis
 - Based on common Internet standards
- Not a standard more of a philosophy for services using
 - HTTP as transport
 - HTTP verbs for operations
 - URIs to represent resources
 - MIME types for content typing
 - HATEOS

{REST}

Connection Information

- To connect you will need:
 - Database ID
 - Cloud Region
 - Keyspace Name
 - Application Token

```
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>
```



CQL Table Definition

- Composite partition key
 - *id* for uniqueness
- We will use the REST API to create this table

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

Create a Table

```
curl --request POST \
--url
https://${ASTRA_DB_ID}-${ASTRA_DB_REGION}.apps.astra.datastax.com/api/rest
/v2/schemas/keyspaces/${ASTRA_DB_KEYSPACE}/tables \
--header 'content-type: application/json' \
--header "x-cassandra-token: ${ASTRA_DB_APPLICATION_TOKEN}" \
--data '{"name": "cars", "columnDefinitions": [
    {"name": "id", "typeDefinition": "int", "static": false},
    {"name": "make", "typeDefinition": "text", "static": false},
    {"name": "model", "typeDefinition": "text", "static": false},
    {"name": "year", "typeDefinition": "int", "static": false}],
"primaryKey": {"partitionKey": ["make", "model"],
"clusteringKey": ["id"]}}'
```

Add Some Data

```
curl --request POST \
--url
https://${ASTRA_DB_ID}-${ASTRA_DB_REGION}.apps.astra.datastax.com/api/rest
/v2/keyspaces/${ASTRA_DB_KEYSPACE}/cars \
--header 'content-type: application/json' \
--header "x-cassandra-token: ${ASTRA_DB_APPLICATION_TOKEN}" \
--data '{
    "id":1001,
    "make":"Dodge",
    "model":"Challenger",
    "year":1971
}'
```

Get All Rows

```
curl --request GET \
--url
https://${ASTRA_DB_ID}-${ASTRA_DB_REGION}.apps.astra.datastax.com/api/rest
/v2/keyspaces/${ASTRA_DB_KEYSPACE}/cars/rows \
--header "x-cassandra-token: ${ASTRA_DB_APPLICATION_TOKEN}"
```

```
{"count":7, "data": [
  {"model":"Challenger","id":1001,"make":"Dodge","year":1971},
  {"model":"Challenger","id":1005,"make":"Dodge","year":1972},
  {"model":"Mustang","id":1002,"make":"Ford","year":1968},
  {"model":"Mustang","id":1006,"make":"Ford","year":1971},
  {"model":"Daytona","id":1004,"make":"Dodge","year":1969},
  {"model":"Charger","id":1007,"make":"Dodge","year":1969},
  {"model":"Camaro","id":1003,"make":"Chevy","year":1969}]}]
```

Get All Rows

```
curl --request GET \
--url
https://${ASTRA_DB_ID}-${ASTRA_DB_REGION}.apps.astra.datastax.com/api/rest
/v2/keyspaces/${ASTRA_DB_KEYSPACE}/cars/rows \
--header "x-cassandra-token: ${ASTRA_DB_APPLICATION_TOKEN}"
```

```
{"count":7, "data": [
  {"model":"Challenger","id":1001,"make":"Dodge","year":1971},
  {"model":"Challenger","id":1005,"make":"Dodge","year":1972},
  {"model":"Mustang","id":1002,"make":"Ford","year":1968},
  {"model":"Mustang","id":1006,"make":"Ford","year":1971},
  {"model":"Daytona","id":1004,"make":"Dodge","year":1969},
  {"model":"Charger","id":1007,"make":"Dodge","year":1969},
  {"model":"Camaro","id":1003,"make":"Chevy","year":1969}]}]
```

Get a Partition

```
curl --request GET \
--url
https://${{ASTRA_DB_ID}}-${{ASTRA_DB_REGION}}.apps.astra.datastax.com/api/rest/v2
/keyspaces/${{ASTRA_DB_KEYSPACE}}/cars/Ford/Mustang \
--header "x-cassandra-token: ${ASTRA_DB_APPLICATION_TOKEN}"
```

```
{"count":2,"data":[
    {"model":"Mustang","id":1002,"make":"Ford","year":1968},
    {"model":"Mustang","id":1006,"make":"Ford","year":1971}]}  

```

Use a WHERE clause

- Composite partition key
 - Finds all rows in *Dodge/Challenger* and *Dodge/Charger* partitions

```
curl --request GET --url  
https://${{ASTRA_DB_ID}}-${{ASTRA_DB_REGION}}.apps.astra.datastax.com/api/rest/v2  
/keyspaces/${{ASTRA_DB_KEYSPACE}}/cars '?where=\{"make":\{"$eq":"Dodge"\}, "model  
":\{"$in":\["Challenger", "Charger"\]\}\}' -H 'accept: application/json' -H  
"X-Cassandra-Token: ${ASTRA_DB_APPLICATION_TOKEN}"  
  
{"count":3,"data": [  
    {"model":"Challenger","id":1001,"make":"Dodge","year":1971},  
    {"model":"Challenger","id":1005,"make":"Dodge","year":1972},  
    {"model":"Charger","id":1007,"make":"Dodge","year":1969}]}  
}
```

Swagger UI

- The Swagger UI supports Stargate's REST API
- Access on the *Connect* tab
- Select REST API
- Click on the link

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/>

Swagger UI

- Separate sections for:
 - DDL - called *schemas*
 - DML - called *data*
- The UI gives sample request payloads
 - This example is for creating a table

```
{  
  "name": "string",  
  "primaryKey": {  
    "partitionKey": [  
      "string"  
    ],  
    "clusteringKey": [  
      "string"  
    ]  
  },  
  "columnDefinitions": [  
    {  
      "name": "emailaddress",  
      "typeDefinition": "text",  
      "static": true  
    }  
  ],  
  "ifNotExists": true,  
  "tableOptions": {  
    "defaultTimeToLive": 0,  
    "clusteringExpression": [  
      {  
        "column": "string"  
      }  
    ]  
  }  
}
```



Hands-on Lab



LAB 21: REST API

- Create a table with the REST API
- Populate and retrieve data



gRPC API

Google's RPC API with Protobuf

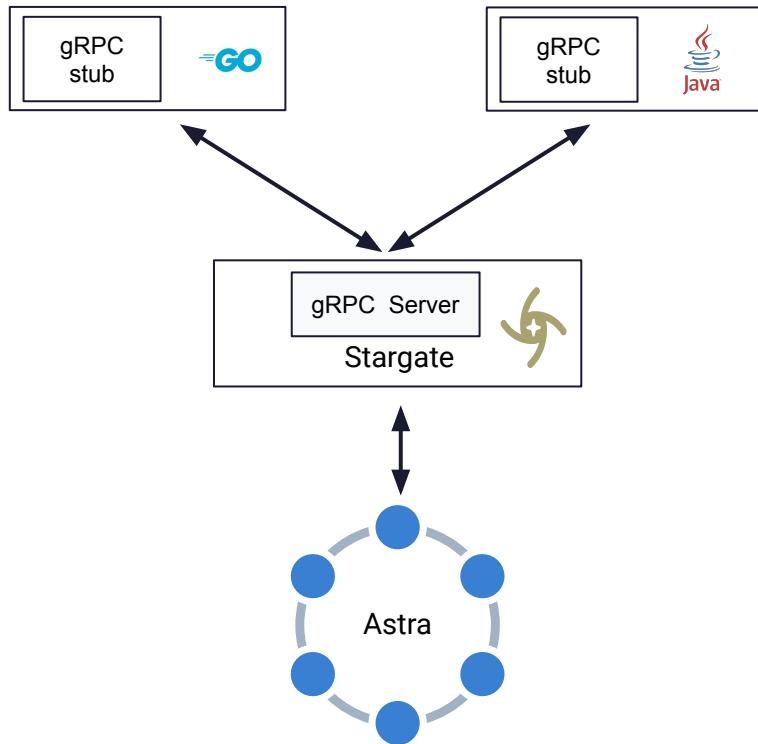
gRPC API

- Open Source Remote Procedure Call (RPC) framework
- Data passed using Protocol Buffers (protobuf)
 - Language independent binary serialization framework
 - Developed by Google
- Supported in:
 - C#, C++, Dart, Go, Java, Kotlin, Node, Objective C, PHP, Python, Ruby



gRPC and Stargate

- gRPC stub in client
- gRPC server in Stargate
- Requests and responses in protobuf
- Synchronous and Asynchronous API



gRPC Client: Java – Maven Dependencies

- These are the required Maven dependencies

```
<dependencies>
  <dependency>
    <groupId>io.stargate.grpc</groupId>
    <artifactId>grpc-proto</artifactId>
    <version>1.0.41</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.41.0</version>
  </dependency>
</dependencies>
```

gRPC Client: Java – Imports

- Requires same connection details as other methods

```
private static final String ASTRA_DB_ID      = "<id>" ;
private static final String ASTRA_DB_REGION   = "<region>" ;
private static final String ASTRA_TOKEN       = "<token>" ;
private static final String ASTRA_KEYSPACE    = "<keyspace>" ;
```

gRPC Client: Java - Create a Blocking Stub

- Create the channel
- Use the channel to create the *blockingStub*

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress(ASTRA_DB_ID + "-" + ASTRA_DB_REGION +
                ".apps.astra.datastax.com", 443)
    .useTransportSecurity()
    .build();

StargateGrpc.StargateBlockingStub blockingStub =
    StargateGrpc.newBlockingStub(channel)
    .withDeadlineAfter(10, TimeUnit.SECONDS)
    .withCallCredentials(new StargateBearerToken(ASTRA_TOKEN)) ;
```

gRPC Client: Java - Execute a Query

- Define and build the query object
- Use the blockingStub
 - Execute the query
- Get the response

```
QueryOuterClass.Response response = blockingStub.executeQuery(  
    QueryOuterClass.Query.newBuilder()  
        .setCql("SELECT * FROM " + ASTRA_KEYSPACE + ".cars")  
        .build());
```

gRPC Client: Java - Process Results

- The `getResultSet()` method converts the response to a Java `ResultSet`

```
QueryOuterClass.ResultSet rs = queryString.getResultSet();
for (Row row : rs.getRowsList()) {
    System.out.println(
        + "id = " + row.getValues(0).getInt() +
        + ", make = " + row.getValues(1).getString() +
        + ", model = " + row.getValues(2).getString() +
        + ", year = " + row.getValues(3).getInt());
}
```

Java Driver

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

- Sessions/Connections
- Resultsets and Rows
- Core Driver
- Query Builder
- Mapper



Sessions/Connections

Connecting to Astra DB

Core Driver

- DataStax Driver for Apache Cassandra™ 4.x
 - New interface since version 3.x
- Supports
 - Astra DB
 - OSS Cassandra
 - DataStax Enterprise

Getting the Driver

- Published in Maven Central
- Group id: com.datastax.oss
- Maven coordinates

```
<dependency>
  <groupId>com.datastax.oss</groupId>
  <artifactId>java-driver-core</artifactId>
  <version>${driver.version}</version>
</dependency>
```

Prerequisites

- Secure token
 - Sufficient permissions for the client application
 - clientID
 - clientSecret
- Secure connect bundle

Session

- CqlSession object
- Pool of connections
- Fluent Builder pattern (Fluent Interface)
 - Call to `build()` attempts to connect to Astra DB
- Executes CQL statements against database

Making a Connection

1. Wrap in try or *try with resources*
2. Get the builder
3. Use the secure connect bundle
4. Provide credentials
5. Build

```
try( CqlSession session = CqlSession.builder()
    .withCloudSecureConnectBundle(Paths.get("{{PATH/TO/bundle.zip}}"))
    .withAuthCredentials("{{CLIENT ID}}", "{{CLIENT SECRET}}")
    .build())
{
    // use the session
}
```



Resultsets and Rows

Handling query results

Resultsets and Rows

- Query results are wrapped in `ResultSet`
 - Similar to `java.sql.ResultSet`
 - Contains `Row`s
- `Rows` are encapsulated in `Row`
 - Columns accessible by name or position
 - Strongly typed getters

Extract One Row From ResultSet

- This query gets all rows from the table
 - The `one()` method extracts one Row from the ResultSet

```
ResultSet resultSet = session.execute("SELECT * FROM examples.cars");

Row row = resultSet.one(); // extract the first row
System.out.printf(
    "%4d %-6s %-10s %4d\n",
    row.getInt("id"),           // positional equivalent - row.getInt(0)
    row.getString("make"),      // positional equivalent - row.getString(1)
    row.getString("model"),     // positional equivalent - row.getString(2)
    row.getInt("year")         // positional equivalent - row.getInt(3)
);
```

Iterate Through the ResultSet

- Use an *enhanced for loop* to extract the Rows from the ResultSet
- ```
ResultSet resultSet = session.execute("SELECT * FROM examples.cars");

for(Row row : resultSet) {
 System.out.printf(
 "%4d %-6s %-10s %4d\n",
 row.getInt("id"), // positional equivalent - row.getInt(0)
 row.getString("make"), // positional equivalent - row.getString(1)
 row.getString("model"), // positional equivalent - row.getString(2)
 row.getInt("year") // positional equivalent - row.getInt(3)
);
}
```



# Core Driver

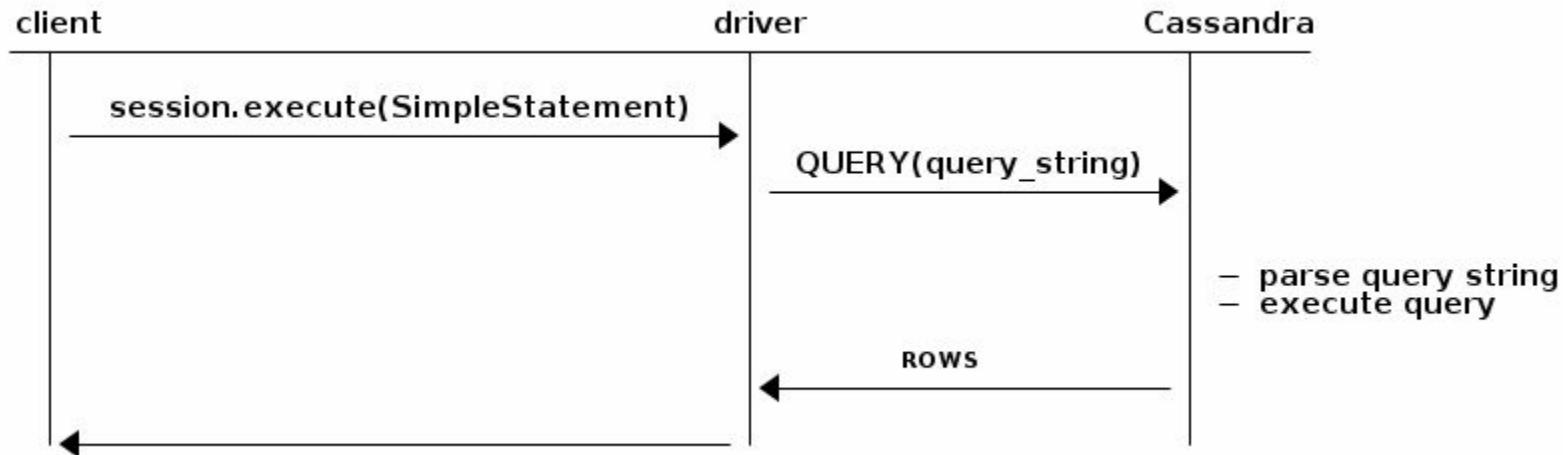
CQL queries in Java

# Statements

- Simple statements
  - One-off executions of a literal query string
- Bound statements
  - Reusable query
  - Configurable values
- Batch statements
  - Group statements into an atomic operation
- Run statement using the `session.execute()` function
  - Returns a `ResultSet`

# Simple Statements

- No caching
- Query string parsed on each request



# Simple Statements – Literal Query String

- Shorthand to execute a query
- With or without explicit statement creation

```
session.execute("SELECT * FROM examples.cars"); // no Statement object

SimpleStatement statement =
 SimpleStatement.newInstance("SELECT * FROM examples.cars");
session.execute(statement);
```

# Simple Statements – Fluent Builder

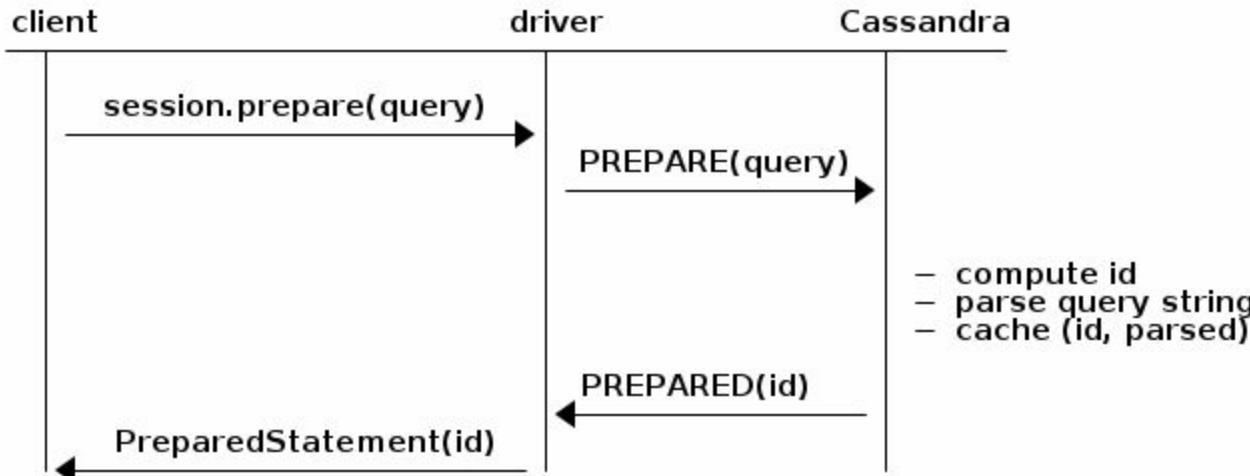
- Use *bind markers*
- Add values separately

```
SimpleStatement statement =
 SimpleStatement.builder("SELECT * FROM examples.cars WHERE make = :m");
 .addNamedValue("m", "Ford")
 .build();

session.execute(statement);
```

# Bound Statements

- Created from a PreparedStatement
  - PreparedStatement created from a SimpleStatement
- Parsed query cached on server



# Bound Statements

- Use *bind markers*
- Add values separately

```
SimpleStatement simpleStatement = SimpleStatement
 .newInstance("SELECT * FROM examples.cars WHERE make = :m");

PreparedStatement preparedStatement = session.prepare(simpleStatement);

BoundStatement boundStatement = preparedStatement.bind()
 .setString("m", "Ford");

session.execute(boundStatement);
```

# Batch Statements

- Combine multiple statements
  - Single partition batches are *atomic* by default
  - Multi partition batches require a batchlog for atomicity
- Batched statements are *not* isolated
  - Other queries may read *in-flight* batch data
- Batches may result in significant performance reductions

# Batch Statements

```
SimpleStatement insert = SimpleStatement
 .newInstance("INSERT INTO examples.cars (id, make, model, year) " +
 "VALUES (:id, :make, model, :year))";

PreparedStatement psInsert = session.prepare(insert);

SimpleStatement simpleInsert = SimpleStatement
 .newInstance("INSERT INTO examples.cars (id, make, model, year) " +
 "VALUES (1101, 'Chevy', 'Camaro', 1970)");

BatchStatement batch = BatchStatement.newInstance (
 DefaultBatchType.LOGGED,
 psInsert.bind(1100, 'Chevy', 'Corvette', 1975),
 simpleInsert);

session.execute(batch);
```



# Query Builder

Fluent interface for Queries

# Query Builder

- Build queries programmatically
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - TRUNCATE
- Fluent API
- Ideal for dynamic queries
- Query Builder queries can be turned into prepared statements
  - Takes advantage of caching

# Using the Query Builder

- Published in Maven Central
- Group id: com.datastax.oss
- Maven coordinates

```
<dependency>
 <groupId>com.datastax.oss</groupId>
 <artifactId>java-driver-query-builder</artifactId>
 <version>${driver.version}</version>
</dependency>
```

# Create a Simple Statement with the Query Builder

- SimpleStatement
- Hard-coded where value

```
SELECT * FROM examples.cars WHERE make='Ford'
```

```
SimpleStatement simpleStatement =
 selectFrom("examples", "cars")
 .all()
 .whereColumn("make")
 .isEqualTo(literal("Ford"))
 .build();

ResultSet rs = session.execute(simpleStatement);
```

# Create a Prepared Statement with the Query Builder

```
SimpleStatement statement =
 selectFrom("examples", "cars")
 .all()
 .whereColumn("make")
 .isEqualTo(bindMarker())
 .build();

PreparedStatement preparedStatement = session.prepare(statement);

BoundStatement boundStatement = preparedStatement.bind("Ford");

ResultSet rs = session.execute(boundStatement);
```



# Mapper

Annotated Object to Table Mapping

# Mapper

- Uses Java annotations
- Generates code to
  - Execute queries
  - Convert results into Java objects

# Components

- Entities
  - Annotated classes that represent table data
- Data Access Objects (DAOs)
  - Annotated interfaces
  - Define (CRUD) operations on database
- Mapper
  - Gives access to generated DAO instance

# Entities

- Encapsulate row data
- Requires *no-arg* constructor
- Annotations
  - `@Entity` - identifies the class as entity
  - `@PartitionKey` - marks partition key column(s)
  - `@ClusteringColumn` - marks clustering columns in order
  - `@CqlName` - maps class/field name to table/column name

# Car Entity

```
@Entity
@CqlName("cars")
public class Car {

 @PartitionKey private String make;
 @ClusteringColumn(1) private String model;
 @ClusteringColumn(2) private int id;
 private int year;

 public Car(String make, String model, int id, int year) { ... }
 public Car() {}

 // getters and setter not shown
}
```

# DAOs

- Interface
- Declares CRUD (and other) operations
- Annotations
  - `@Dao` - declares interface is a DAO
  - `@Insert` - insert method (auto-generated)
  - `@Select` - select method (auto-generated)
  - `@Update` - update method (auto-generated)
  - `@Delete` - delete method (auto-generated)
  - `@Query` - arbitrary operation (user-defined)

# Car DAO

```
@Dao public interface CarDAO {

 @Select PagingIterable<Car> findAllCars();

 @Select PagingIterable<Car> findCarsByMake(String make, String model);

 @Query("SELECT count(*) FROM examples.cars WHERE make = :make")
 long countMake(String make);

 @Insert void save(Car car);

}
```

# Car Mapper

- Interface
- Contains *factory* method for acquiring DAO instance

```
@Mapper
public interface CarMapper {
 @DaoFactory
 CarDAO carDAO(@DaoKeyspace CqlIdentifier keyspace);
}
```

# Use the Mapper API

```
CarMapper carMapper = new CarMapperBuilder(session).build();
CarDAO carDAO = carMapper.carDAO(CqlIdentifier.fromCql("examples"));

carDAO.save(new Car("Chevy", "Nova", 2000, 1974));

PagingIterable<Car> cars = carDAO.findCarsByMake("Chevy", "Nova");

for(Car car : cars) {
 System.out.printf("%4d %-6s %-10s %4d\n",
 car.getId(), car.getMake(), car.getModel(), car.getYear()
);
}

String make = "Dodge";
long count = carDAO.countMake(make);
System.out.printf("%d %ss in the table", count, make);
```

**DataStax**

**Thank  
You!**