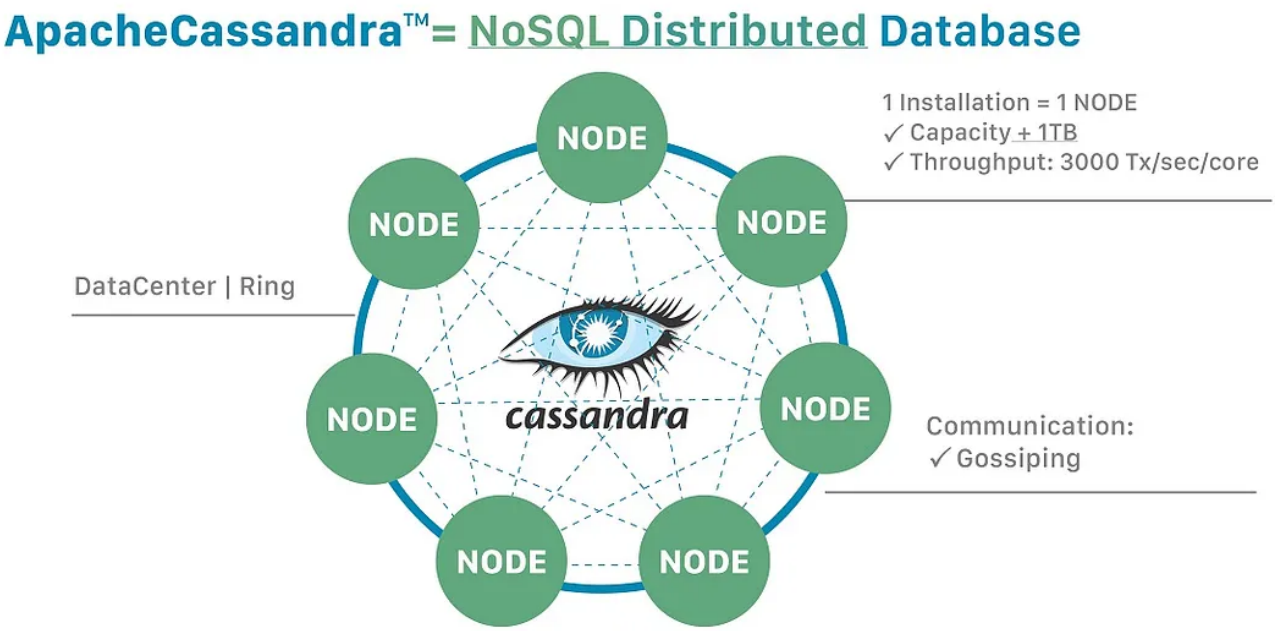
Deploying Apache Cassandra Cluster (3 Nodes) with Docker Compose

Keivan Soleimani · Follow

6 min read · May 8, 2024



Apache Cassandra is an open source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data.

Masterless architecture and low latency means Cassandra will withstand an entire data center outage with no data loss and Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the

peace of mind of knowing that you can survive regional outages. Failed nodes can be replaced with no downtime.

Cassandra is suitable for applications that can't afford to lose data, even when an entire data center goes down. There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical.

Cassandra streams data between nodes during scaling operations such as adding a new node or datacenter during peak traffic times. Zero Copy Streaming makes this up to 5x faster without vnodes for a more elastic architecture particularly in cloud and Kubernetes environments.

We want to deploy a 3 node cluster in Docker Desktop ...

Pull the docker image of cassandra :

```
docker pull cassandra
```

Some Notices and roles about deployment :

- Controlling the startup order of the nodes in the Compose file, such that Compose first makes sure that the seed node cassandra-1 is up and healthy, then starts cassandra-2 and also makes sure that cassandra-2 node is up and healthy, then starts cassandra-3, and so on. Basically, preventing nodes from all starting simultaneously, especially the nodes after the seed node. When nodes are started simultaneously with Compose, it can lead to errors such as a conflict with token ranges, causing some of the nodes to fail to join the cluster.
- Using a Snitch configuration that more resembles your production environment, which is usually when a multi-node or multi-cluster or multi-datacenter becomes necessary. For example, you can use a GossipingPropertyFileSnitch,

which is also the same Snitch type used in the Cassandra tutorial for Initializing a multiple node cluster (multiple datacenters).

- Explicitly setting the CASSANDRA_CLUSTER_NAME and CASSANDRA_DC environment variables, which correspondingly sets the cluster_name on the cassandra.yaml config and the dc option on the cassandra-rackdc.properties file. This allows you explicitly tell the nodes to join the same datacenter and cluster. These options are only relevant for GossipingPropertyFileSnitch.

docker compose file with network cassandra-net :

```
version: "3.3"

networks:
  cassandra-net:
    driver: bridge

services:

  cassandra-1:
    image: "cassandra:latest"  # cassandra:4.1.3
    container_name: "cassandra-1"
    ports:
      - 7000:7000
      - 9042:9042
    networks:
      - cassandra-net
    environment:
      - CASSANDRA_START_RPC=true       # default
      - CASSANDRA_RPC_ADDRESS=0.0.0.0  # default
      - CASSANDRA_LISTEN_ADDRESS=auto  # default, use IP addr of container # =
      - CASSANDRA_CLUSTER_NAME=my-cluster
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
      - CASSANDRA_DC=my-datacenter-1
    volumes:
      - cassandra-node-1:/var/lib/cassandra:rw
    restart:
      on-failure
    healthcheck:
      test: ["CMD-SHELL", "nodetool status"]
      interval: 2m
      start_period: 2m
      timeout: 10s
```

```
    retries: 3

cassandra-2:
  image: "cassandra:latest"  # cassandra:4.1.3
  container_name: "cassandra-2"
  ports:
    - 9043:9042
  networks:
    - cassandra-net
  environment:
    - CASSANDRA_START_RPC=true       # default
    - CASSANDRA_RPC_ADDRESS=0.0.0.0  # default
    - CASSANDRA_LISTEN_ADDRESS=auto  # default, use IP addr of container # =
    - CASSANDRA_CLUSTER_NAME=my-cluster
    - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
    - CASSANDRA_DC=my-datacenter-1
    - CASSANDRA_SEEDS=cassandra-1
  depends_on:
    cassandra-1:
      condition: service_healthy
  volumes:
    - cassandra-node-2:/var/lib/cassandra:rw
  restart:
    on-failure
  healthcheck:
    test: ["CMD-SHELL", "nodetool status"]
    interval: 2m
    start_period: 2m
    timeout: 10s
    retries: 3

cassandra-3:
  image: "cassandra:latest"  # cassandra:4.1.3
  container_name: "cassandra-3"
  ports:
    - 9044:9042
  networks:
    - cassandra-net
  environment:
    - CASSANDRA_START_RPC=true       # default
    - CASSANDRA_RPC_ADDRESS=0.0.0.0  # default
    - CASSANDRA_LISTEN_ADDRESS=auto  # default, use IP addr of container # =
    - CASSANDRA_CLUSTER_NAME=my-cluster
    - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
    - CASSANDRA_DC=my-datacenter-1
    - CASSANDRA_SEEDS=cassandra-1
  depends_on:
    cassandra-2:
```

```
        condition: service_healthy
    volumes:
      - cassandra-node-3:/var/lib/cassandra:rw
    restart:
      on-failure
    healthcheck:
      test: ["CMD-SHELL", "nodetool status"]
      interval: 2m
      start_period: 2m
      timeout: 10s
      retries: 3

volumes:
  cassandra-node-1:
  cassandra-node-2:
  cassandra-node-3:
```

The main thing here are the healthcheck blocks :

```
healthcheck:
      test: ["CMD-SHELL", "nodetool status"]
      interval: 2m
      start_period: 2m
      timeout: 10s
      retries: 3
```

and the updated depends_on on each node :

```
depends_on:
      cassandra-2:
        condition: service_healthy
```

The modified Compose sets cassandra-3 to only start when cassandra-2 is healthy, and to only start cassandra-2 when cassandra-1 is healthy.

In that Compose file:

- Call nodetool status after 2 minutes (to give time for the node to bootup/ bootstrap)
- If it responds in <10s and the exit code is 0, the node is to be considered healthy
- Repeat the check every 2m and for 3 times.

There's also some extra env vars in there :

```
environment:
      - CASSANDRA_START_RPC=true        # default
      - CASSANDRA_RPC_ADDRESS=0.0.0.0   # default
      - CASSANDRA_LISTEN_ADDRESS=auto   # default, use IP addr of container # =
```

which may not be needed, since those are already the defaults on the cassandra Docker image (see section on Configuring Cassandra from the Dockerhub page. Basically, those explicitly set the IP address of the containers to be both the listen and broadcast address. I'm just noting it here in case the defaults change.

if you are running all the nodes in the same machine, you need to specify different ports for each of them :

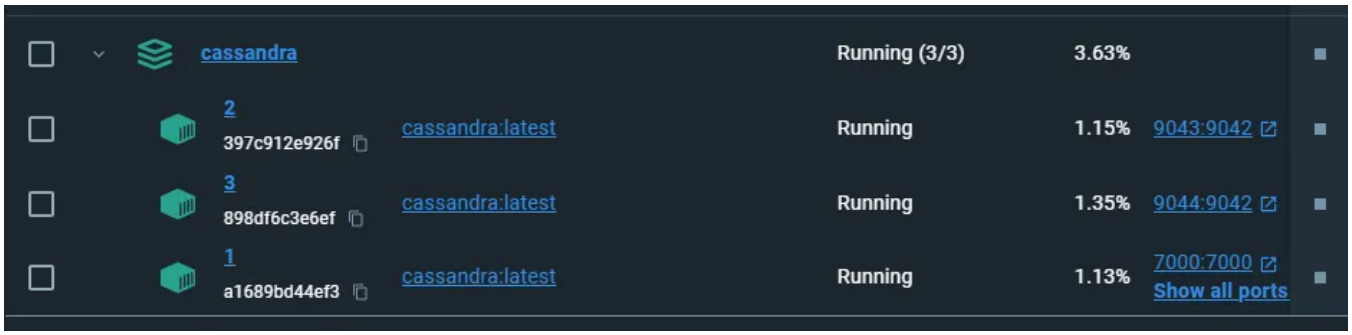
```
cassandra-1:
    ...
    ports:
      - 7000:7000
      - 9042:9042

  cassandra-2:
    ...
    ports:
      - 9043:9042

  cassandra-3:
    ...
    ports:
```

```
      - 9044:9042
```

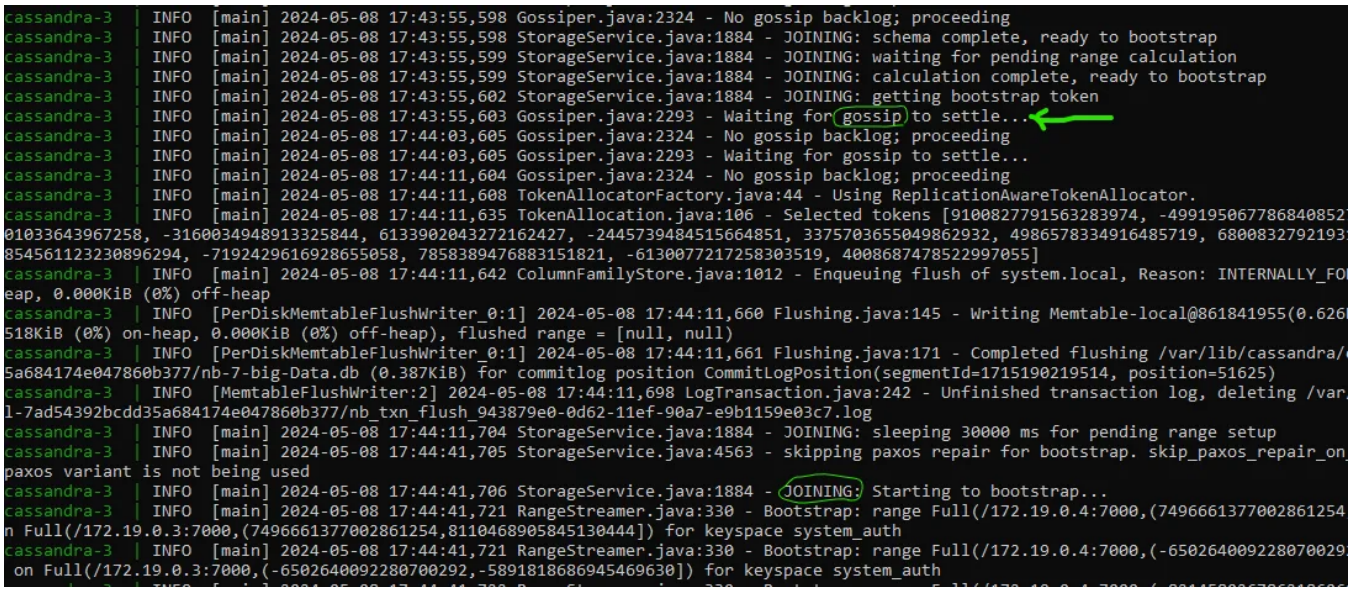
otherwise, the containers may not start correctly.

Start your deployment :

```
docker compose up
```



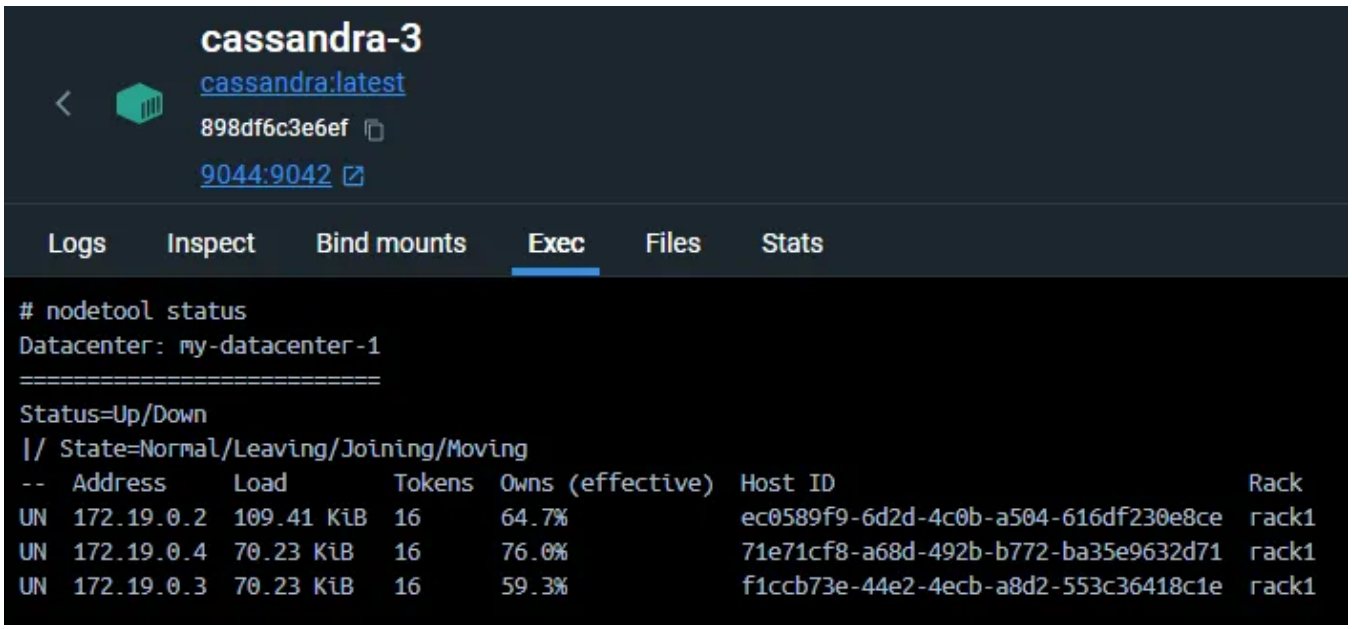
Check all containers log for being healthy and joining process :



The gossip protocol should detect if a particular node goes down and mark it as such, but still keep it in the list.

Use nodetool for check the status in all 3 nodes :

```
docker exec cassandra-3 nodetool status
```



The main problem with this config is that starting the nodes takes a long time. In that sample Compose file where healthcheck.interval is 2m, it takes about ~5mins for all 3 nodes to properly start-up.

The Cassandra Query Language (CQL) is very similar to SQL but suited for the JOINless structure of Cassandra.

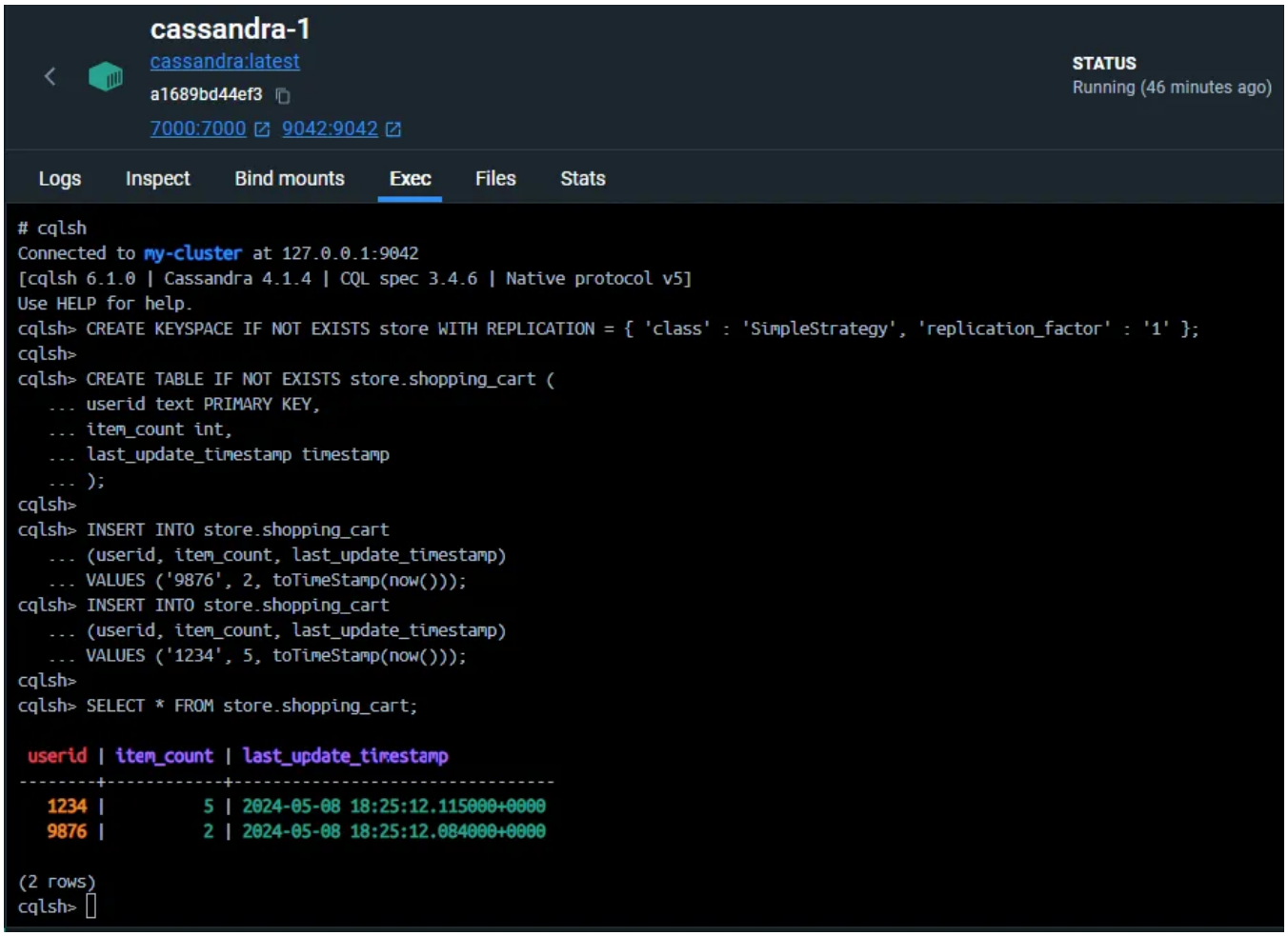
Start to working with the cluster with cqlsh ...

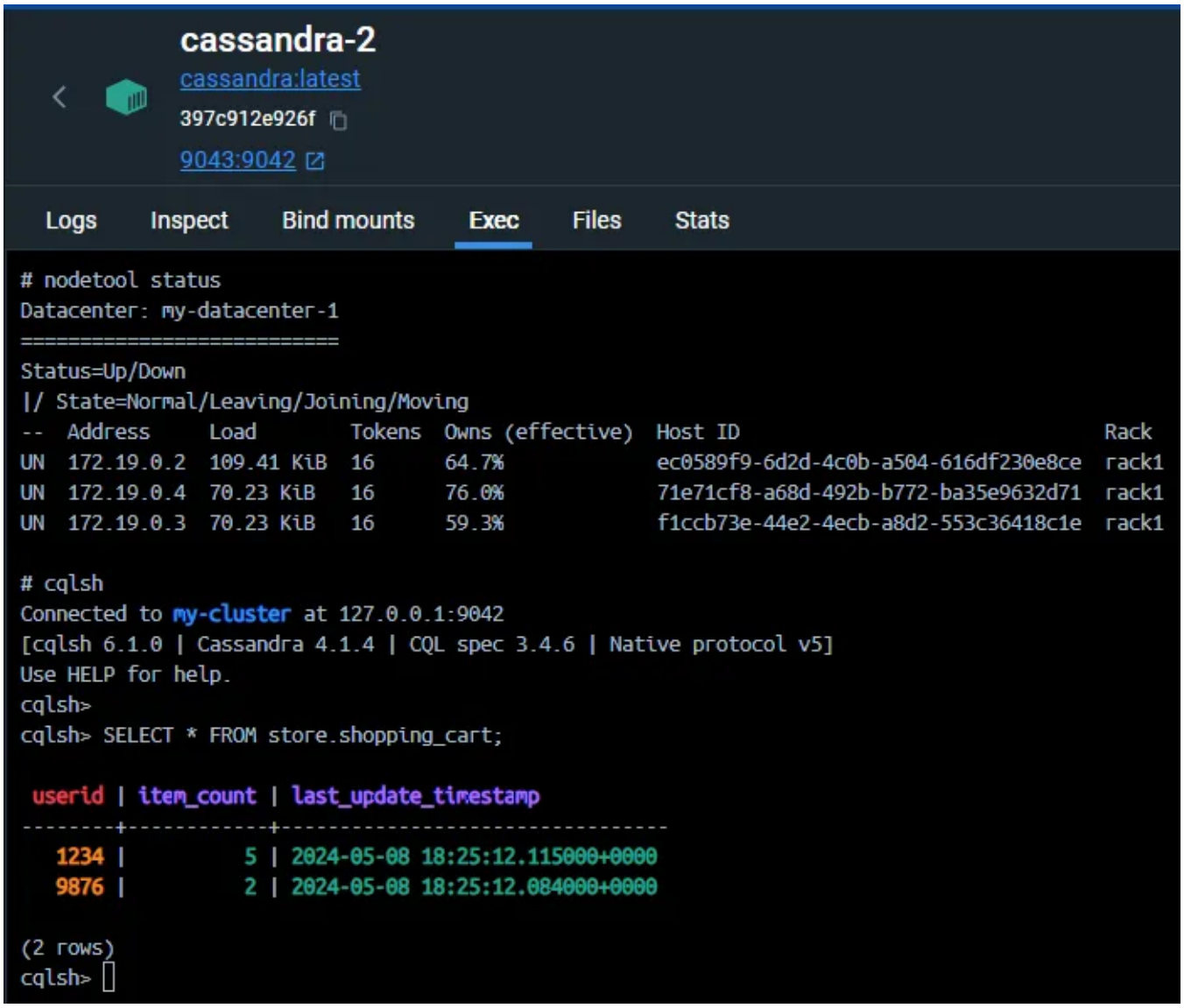
We want to create a keyspace, the layer at which Cassandra replicates its data, a table to hold the data, and insert some data into that table :

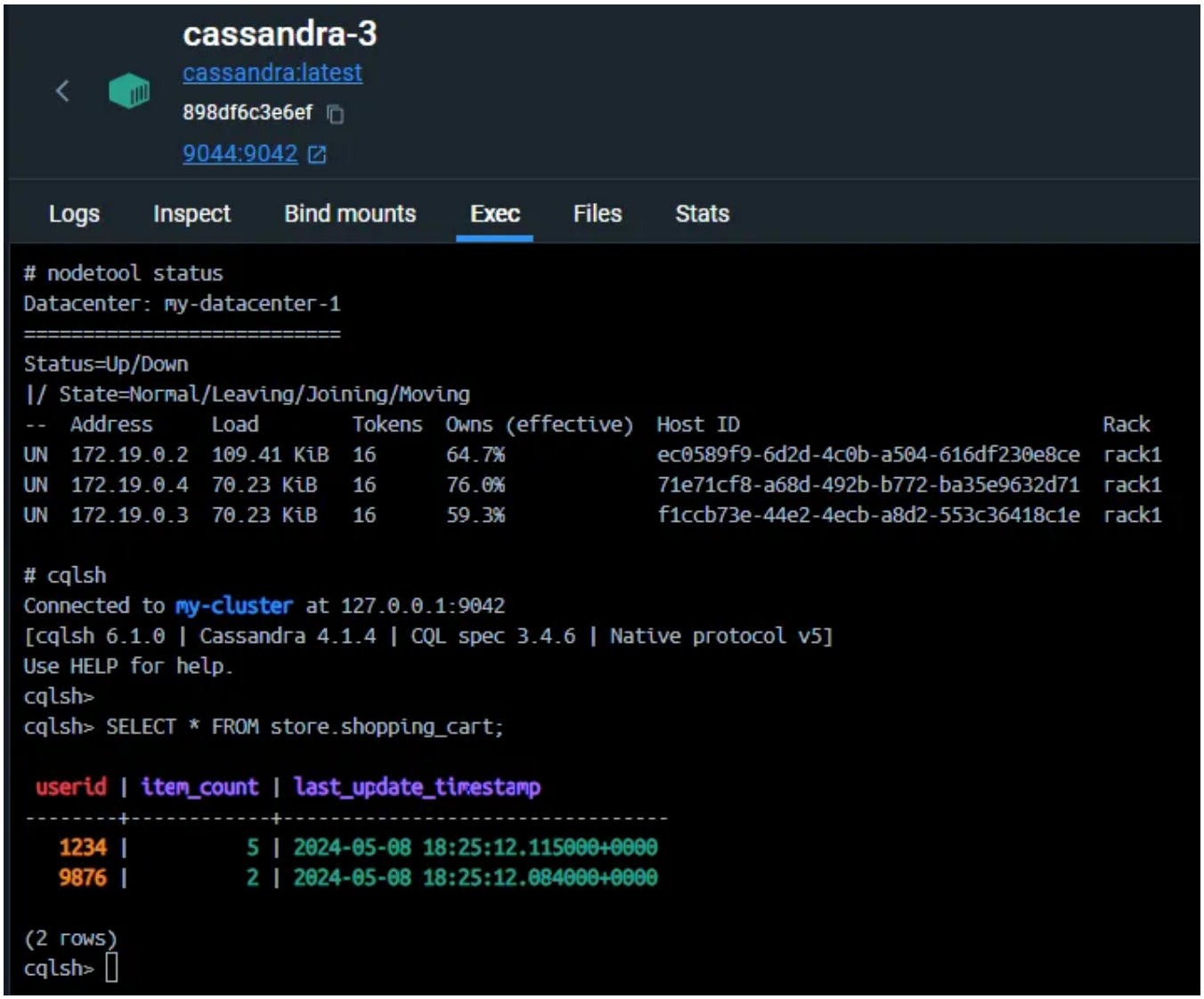
```
docker exec -it cassandra-1 bash
```

```
# cqlsh
...
cqlsh> CREATE KEYSPACE IF NOT EXISTS store WITH REPLICATION = { 'class' : 'Simp

cqlsh> CREATE TABLE IF NOT EXISTS store.shopping_cart (
        userid text PRIMARY KEY,
        item_count int,
        last_update_timestamp timestamp
);

cqlsh> INSERT INTO store.shopping_cart
      (userid, item_count, last_update_timestamp)
      VALUES ('9876', 2, toTimeStamp(now()));

cqlsh> INSERT INTO store.shopping_cart
      (userid, item_count, last_update_timestamp)
      VALUES ('1234', 5, toTimeStamp(now()));

cqlsh> SELECT * FROM store.shopping_cart;
```

the result in all 3 nodes :


cassandra-1
cassandra:latest
a1689bd44ef3
7000:7000 9042:9042
STATUS
Running (46 minutes ago)
Logs Inspect Bind mounts Exec Files Stats
cqlsh
Connected to my-cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.4 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> CREATE KEYSPACE IF NOT EXISTS store WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
cqlsh>
cqlsh> CREATE TABLE IF NOT EXISTS store.shopping_cart (
... userid text PRIMARY KEY,
... item_count int,
... last_update_timestamp timestamp
...);
cqlsh>
cqlsh> INSERT INTO store.shopping_cart
... (userid, item_count, last_update_timestamp)
... VALUES ('9876', 2, toTimeStamp(now()));
cqlsh> INSERT INTO store.shopping_cart
... (userid, item_count, last_update_timestamp)
... VALUES ('1234', 5, toTimeStamp(now()));
cqlsh>
cqlsh> SELECT * FROM store.shopping_cart;
userid | item_count | last_update_timestamp
1234 | 5 | 2024-05-08 18:25:12.115000+0000
9876 | 2 | 2024-05-08 18:25:12.084000+0000
(2 rows)
cqlsh>


cassandra-2
cassandra:latest
397c912e926f
9043:9042
Logs
Inspect
Bind mounts
Exec
Files
Stats
nodetool status
Datacenter: my-datacenter-1
==========
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
-- Address Load Tokens Owns (effective) Host ID Rack
UN 172.19.0.2 109.41 KiB 16 64.7% ec0589f9-6d2d-4c0b-a504-616df230e8ce rack1
UN 172.19.0.4 70.23 KiB 16 76.0% 71e71cf8-a68d-492b-b772-ba35e9632d71 rack1
UN 172.19.0.3 70.23 KiB 16 59.3% f1ccb73e-44e2-4ecb-a8d2-553c36418c1e rack1
cqlsh
Connected to my-cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.4 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh>
cqlsh> SELECT * FROM store.shopping_cart;
userid | item_count | last_update_timestamp
1234 | 5 | 2024-05-08 18:25:12.115000+0000
9876 | 2 | 2024-05-08 18:25:12.084000+0000
(2 rows)
cqlsh>



Apache Cassandra | Docker Compose | Nodetool | Cqlsh | Cql

Follow

Written by Keivan Soleimani

36 Followers · 7 Following

Backend (C# | SQL Server | Java Spring) | Frontend (React) | Big Data (Spark on Kubernetes | Kafka) |

Kubernetes (On-Premise Cluster)

Responses (2)

Giorgio Zoppi

What are your thoughts?

Alan Nunes
Sep 19, 2024

Very good, I could get my first cassandra cluster up in my machine and tried it for the first time :)

2 Reply

Seb Schab
Sep 24, 2024

How come the data can be found in every node if the replication factor is 1?

1 reply Reply

More from Keivan Soleimani

Keivan Soleimani

Deploy Elasticsearch, Kibana & Logstash (ELK Stack) with Docker Compose

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with...

Jun 1, 2024 8

Keivan Soleimani

Spark On Kubernetes

Spark On Kubernetes Cluster

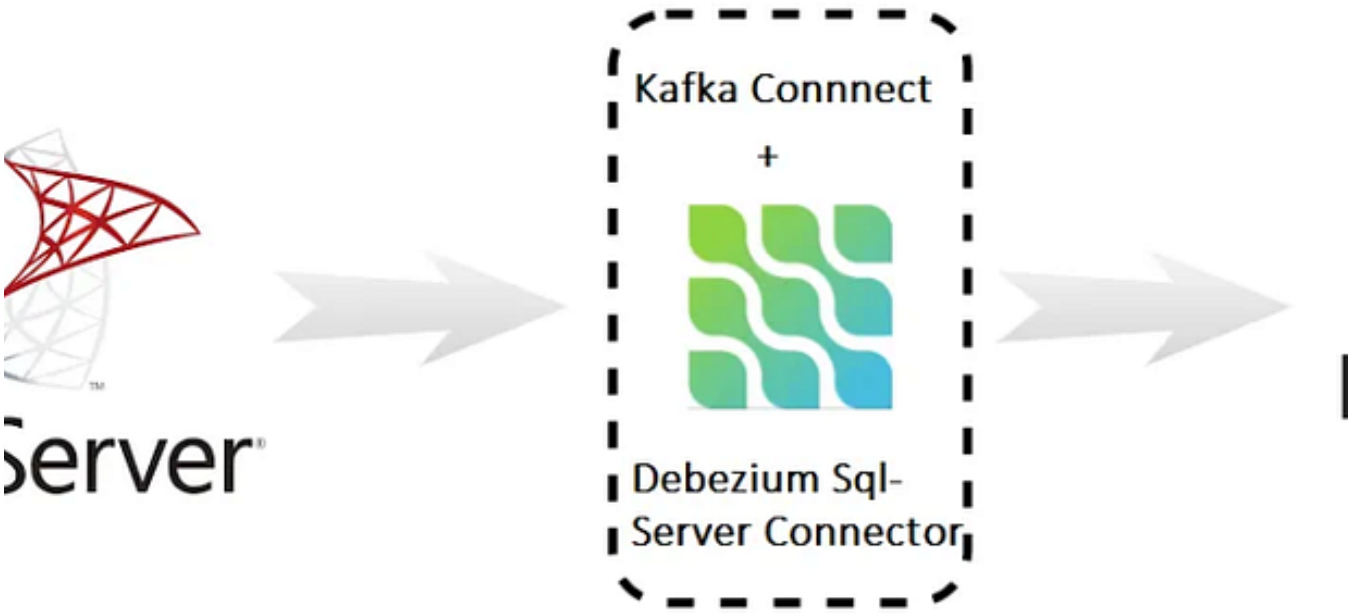
Mar 28, 2024 11 1

Keivan Soleimani

PySpark & Jupyter Notebooks Deployed On Kubernetes

PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides...

Apr 16, 2024 7



Keivan Soleimani

Debezium source connector from SQL Server to Apache Kafka

The Debezium SQL Server connector is based on the change data capture feature that is available in SQL Server 2016 Service Pack 1 (SP1) and...

May 1, 2024 58

See all from Keivan Soleimani

Recommended from Medium

Bora Köstem

Monitoring Your Web App with Prometheus and Grafana: A Step-by-Step Guide

Monitoring web applications is essential for ensuring optimal performance and reliability. As applications grow in complexity, tracking...

Oct 27, 2024 20

In DevOps.dev by Mohit Rathore

Monitoring FastAPI Using Grafana and Prometheus

Monitoring APIs is crucial to ensure their health, performance, and reliability. In this guide, we'll walk through setting up monitoring...

Sep 12, 2024 49

Lists

Natural Language Processing

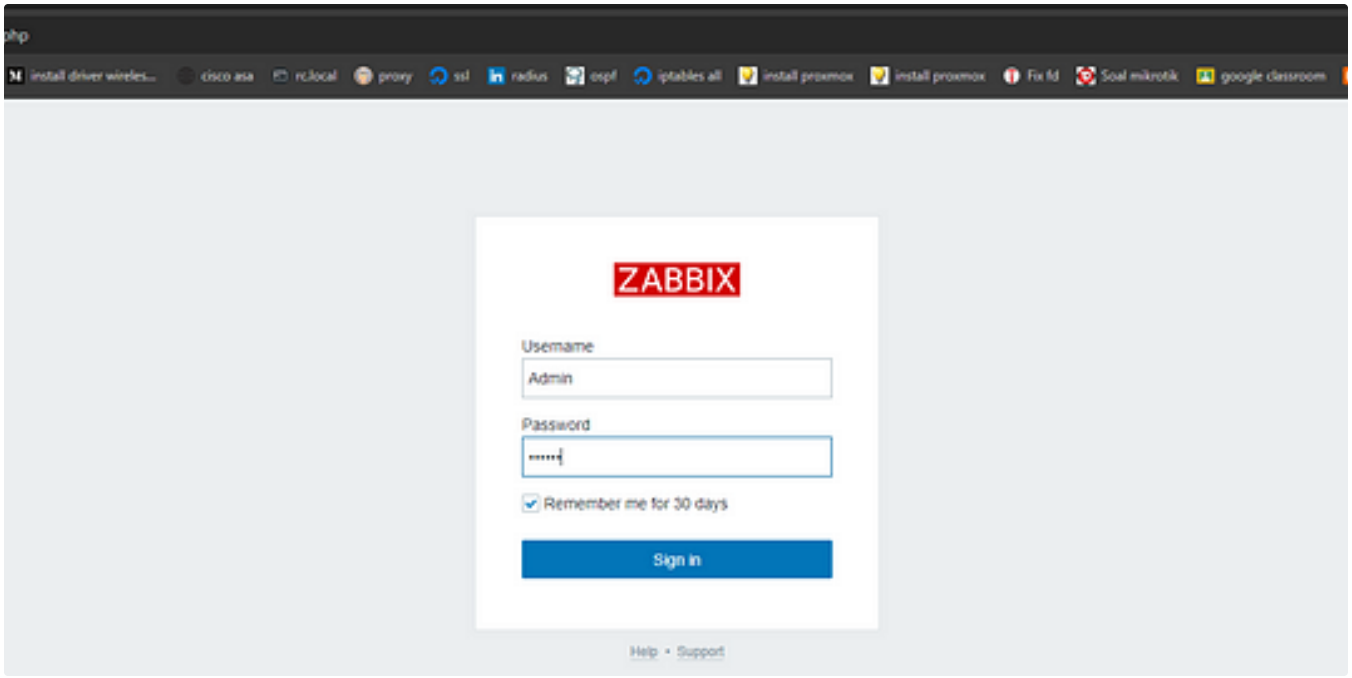
1958 stories · 1605 saves

In itversity by Durga Gadiraju

SQL Performance Tuning: 10 Real-World Scenarios (and Interview Prep Tips)

Boost query speed, reduce bottlenecks, and impress hiring managers with practical SQL optimization strategies.

Feb 13 1 1

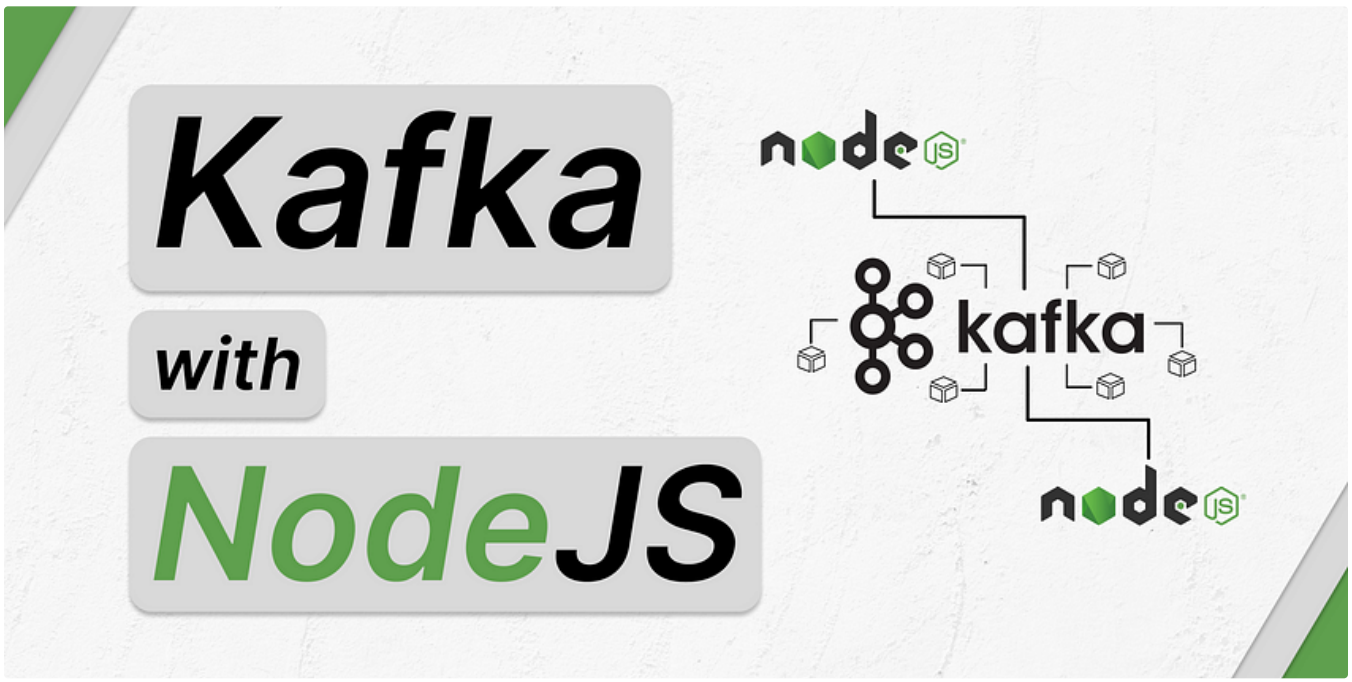


Btech Engineering

Deploy Monitoring Server Using Zabbix With External PostgreSQL-16

In this scenario, the administrator wants to monitor the database server and router core and report monthly. The administrator planned to...

Oct 2, 2024 4



Dinesh Murali

Integrate Kafka with NodeJS

Apache Kafka is a powerful distributed event streaming platform widely used for building real-time data pipelines and streaming...

Nov 9, 2024 20

Service	Purpose	Transformation?	Time?	Use Case
Amazon Kinesis Firehose	Streams data to S3, Redshift, Elasticsearch	No	Yes	Capture real-time data and store it in S3 or other services.
Amazon Kinesis Data Streams	Ingests and streams data in real time	No	Yes	Custom streaming applications (requires manual processing).
Amazon Kinesis Data Analytics	Real-time SQL-based transformations	Yes	Yes	Transform streaming data on the fly (e.g., filter, aggregate, clean data).
AWS Batch	Batch compute for non-real-time workloads	Yes	No	Periodic data transformations (e.g., once per hour/day).
AWS DMS	Database migration service	Limited	No	Migrate data between databases, not for real-time

Anix Lynch, MBA, ex-VC

5 Amazon Kinesis Family For Real-Time Data Streaming 🚜

· 1. Amazon Kinesis Firehose · 2. Amazon Kinesis Data Streams · 3. Amazon Kinesis Data Analytics · 4. AWS Batch · 5. AWS DMS (Database...

Jan 10 3

See more recommendations