**1. Introduction to Debezium**

**What is Debezium?**

* **Definition**: Debezium is an **open-source distributed platform** designed for **Change Data Capture (CDC)**.
* **Purpose**: It captures **real-time changes** (inserts, updates, deletes) from databases and streams them to various data stores or messaging systems.
* **Use Case**: It is used to keep data in sync across systems and build **reactive, data-driven applications**.

**Why Debezium?**

* **Problem**: In systems like WhatsApp or Instagram, data is continuously collected and stored in databases (SQL or NoSQL).

However, this data is not only stored but also needs to be sent to **analytics engines** (e.g., Google Cloud Platform) for processing.

* **Challenge**: Sending the **entire dataset** every time is inefficient. Instead, only the **changes** (inserts, updates, deletes) need to be captured and sent.
* **Solution**: Debezium automates the process of **Change Data Capture (CDC)**, eliminating the need for manual procedures like writing stored procedures or custom functions.

**Key Benefit:**

* Debezium simplifies CDC, making it easier to capture and stream database changes in real-time without reinventing the wheel.

**2. High-Level Scenario: How Debezium Fits into a System**

**Scenario:**

* **System**: Instagram or WhatsApp.
* **Data Flow**:
  1. Data is collected and stored in a **database** (e.g., MySQL, PostgreSQL, MongoDB).
  2. This data needs to be sent to an **analytics engine** (e.g., Google Cloud Platform) for analysis.
  3. Instead of sending the entire dataset, only the **changes** (inserts, updates, deletes) are captured and sent.

**Role of Debezium:**

* Debezium acts as a **middleware** that captures changes from the database and streams them to the analytics engine.
* It ensures that the analytics engine is always updated with the latest changes from the database.

**3. Architecture of Debezium**

**Key Components:**

1. **Change Data Capture (CDC)**:
   * **Definition**: CDC refers to the process of capturing and tracking changes (inserts, updates, deletes) in a database.
   * **How Debezium Works**:
     + Debezium **monitors** the database and extract changes in real time.
     + These changes are **streamed** to other systems (e.g., analytics engines, data warehouses).
2. **Debezium Connectors**:
   * **Definition**: Connectors are specific to different databases and handle the extraction of change data.
   * **Examples**:
     + MySQL Connector
     + PostgreSQL Connector
     + MongoDB Connector
   * **Custom Connectors**: If a database is not natively supported, custom connectors can be built.
3. **Kafka Connect**:
   * **Definition**: Debezium runs on **Kafka Connect**, which provides the infrastructure for running and managing connectors.
   * **Role**:

Kafka Connect handles the **distribution** and **scaling** of data ingestion tasks.

Connectors are configured through Kafka Connect's configuration files or APIs.

**4. Features of Debezium**

**1. Real-Time Data Streaming:**

* **Event Capture**: Debezium captures changes from the source database in **real-time**.
* **Event Metadata**: Each change event includes metadata such as:
  + **Type of change** (insert, update, delete).
  + **Data before and after the change**.
  + **Timestamp** of the change.

**2. Support for Multiple Databases:**

* **Supported Databases**: MySQL, PostgreSQL, SQL Server, Oracle, MongoDB, etc.
* **Custom Connectors**: If a database is not natively supported, custom connectors can be created.

**3. Schema Evolution:**

* **Definition**: Debezium captures **schema changes** (e.g., new columns, changes in data types) from the source database.
* **Importance**: Ensures that downstream systems are aware of schema updates.
* **Schema Metadata**: Captured events include schema metadata, allowing downstream systems to interpret the data accurately.

**4. Integration with Kafka Ecosystem:**

* **Kafka Topics**: Changes are streamed to **Kafka topics**, which can be consumed by various applications and data stores.
* **Kafka Streams**: Kafka Streams or other processing frameworks can be used to process and transform the data in real-time.

**5. Technical Details of Debezium**

**1. Database-Specific Protocols:**

* **Log-Based CDC**:
  + **How It Works**: Debezium reads changes from the **database transaction logs**.
  + **Advantage**: Captures all changes without impacting database performance.
* **Polling-Based CDC**:
  + **How It Works**: Periodically queries the database for changes.
  + **Disadvantage**: Less efficient compared to log-based CDC.
  + **Use Case**: Used for databases that do not support log-based CDC.

**2. Connector Configuration:**

* **Configuration Parameters**:
  + **Database URL**, **username**, **password**.
  + **Filters** for specific tables or columns.
  + **Options** for handling historical data.
* **Example**: Configuring a MySQL connector to capture changes from specific tables.

**3. Data Formats:**

* **Change Events**: Usually formatted in **JSON**.
* **Fields Included**:
  + **Operation type** (insert, update, delete).
  + **Table name**.
  + **Primary key**.
  + **Before and after values** of the change.
  + **Timestamp** indicating when the change occurred.

**8. Challenges of Debezium**

**1. Complex Configuration:**

* **Challenge**: Setting up and configuring connectors can be complex, especially for large or diverse database environments.

**2. Resource Usage:**

* **Challenge**: CDC can consume significant resources, especially for high-throughput databases, requiring careful management of database performance and connector resources.

**3. Compatibility:**

* **Challenge**: Some databases may not support log-based CDC, necessitating alternative approaches or custom solutions.

**10. Conclusion**

* **Debezium** is a powerful tool for **Change Data Capture (CDC)**, enabling real-time data synchronization across systems.
* It simplifies the process of capturing and streaming database changes, making it easier to build **reactive, data-driven applications**.
* By mastering Debezium, you can handle complex data integration challenges and design scalable, efficient systems.

**Scenario: Real-Time Database Synchronization Using Debezium and Kafka**

**Problem Statement:**

* You have two databases, **DB1** and **DB2**.
* Whenever a **new transaction** or **change** occurs in **DB1**, you want the **same update** to happen in **DB2** in **real-time**.
* **Kafka** is used to transfer updates between the two databases.
* You need a tool to **monitor and capture changes** in **DB1** and send them to **Kafka**, which then passes them to **DB2**.

**Solution: Debezium for Change Data Capture (CDC)**

Debezium is the perfect tool for this scenario. It will **monitor DB1**, **capture changes** (inserts, updates, deletes), and **stream them to Kafka**. Kafka will then forward these changes to **DB2**.

**Workflow of Debezium in This Scenario**

**Step 1: Set Up the Environment**

1. **Databases**:
   * **DB1**: The source database where transactions are initially stored.
   * **DB2**: The target database that needs to stay in sync with DB1.
2. **Kafka**:
   * Acts as the **message broker** to transfer changes from DB1 to DB2.
3. **Debezium**:
   * Monitors DB1, captures changes, and sends them to Kafka.

**Step 2: Install and Configure Debezium**

1. **Install Debezium**:
   * Debezium runs as a **Kafka Connect plugin**. Install it on the same machine where Kafka is running.
   * Download the Debezium connector for your database (e.g., MySQL, PostgreSQL).
2. **Configure Debezium Connector**:
   * Create a configuration file for the Debezium connector. This file tells Debezium:
     + **Which database to monitor** (DB1).
     + **What tables to capture changes from**.
     + **Where to send the changes** (Kafka topics).

Example configuration for a MySQL database:

{

"name": "db1-connector",

"config": {

"connector.class": "io.debezium.connector.mysql.MySqlConnector",

"database.hostname": "localhost",

"database.port": "3306",

"database.user": "root",

"database.password": "password",

"database.server.id": "1",

"database.server.name": "dbserver1",

"database.include.list": "db1",

"table.include.list": "db1.transactions",

"database.history.kafka.bootstrap.servers": "localhost:9092",

"database.history.kafka.topic": "dbhistory.dbserver1"

}

}

**Step 3: How Debezium Captures Changes**

1. **Log-Based Change Data Capture (CDC)**:
   * Debezium uses the **transaction logs** of DB1 (e.g., MySQL's binary log or PostgreSQL's write-ahead log) to capture changes.
   * These logs record every **insert**, **update**, and delete operation in the database
   * Debezium **reads these logs** in real-time and extracts the changes.
2. **Polling-Based CDC (if log-based is not supported)**:
   * If the database does not support log-based CDC, Debezium can periodically **poll the database** for changes (less efficient).

**Step 4: Sending Changes to Kafka**

1. **Debezium Publishes Changes to Kafka Topics**:
   * Debezium converts each change into a **JSON message** and publishes it to a **Kafka topic**.
   * Example of a change event:

{

"before": null,

"after": {

"id": 101,

"amount": 100.50,

"timestamp": "2023-10-01T12:34:56Z"

},

"source": {

"db": "db1",

"table": "transactions"

},

"op": "c", // 'c' for create (insert), 'u' for update, 'd' for delete

"ts\_ms": 1696166096000

}

* + The Kafka topic name is typically derived from the database and table name (e.g., dbserver1.db1.transactions).

1. **Kafka Stores the Changes**:
   * Kafka acts as a **message queue**, storing the change events until they are consumed by the downstream system (DB2).

**Step 5: Consuming Changes from Kafka and Updating DB2**

1. **Kafka Consumer**:
   * A **Kafka consumer** is set up to read the change events from the Kafka topic.
   * This consumer can be a custom application or a Kafka Connect sink connector.
2. **Apply Changes to DB2**:
   * The Kafka consumer reads each change event and applies it to **DB2**.
   * For example:
     + If the event is an **insert**, the consumer inserts the new record into DB2.
     + If the event is an **update**, the consumer updates the corresponding record in DB2.
     + If the event is a **delete**, the consumer deletes the record from DB2.
3. **Example of Applying Changes**:
   * For the change event shown earlier, the consumer would execute the following SQL in DB2:

INSERT INTO transactions (id, amount, timestamp)

VALUES (101, 100.50, '2023-10-01T12:34:56Z');

**Step 6: Real-Time Synchronization**

* As soon as a change occurs in **DB1**, Debezium captures it and sends it to Kafka.
* Kafka forwards the change to the consumer, which applies it to **DB2**.
* This entire process happens in **real-time**, ensuring that **DB2** is always in sync with **DB1**.

**Tools Used in This Workflow**

1. **Debezium**:
   * Captures changes from **DB1** and sends them to Kafka.
2. **Kafka**:
   * Acts as the **message broker** to transfer changes between DB1 and DB2.
3. **Kafka Connect**:
   * Provides the infrastructure for running and managing Debezium connectors.
4. **Kafka Consumer**:
   * Reads change events from Kafka and applies them to **DB2**.

**Why This Workflow Works**

1. **Real-Time Updates**:
   * Debezium captures changes in real-time, ensuring that **DB2** is always up-to-date.
2. **Scalability**:
   * Kafka can handle high volumes of change events, making the system scalable.
3. **Fault Tolerance**:
   * Kafka stores change events until they are successfully processed, ensuring no data loss.

**Example Walkthrough**

1. **Change in DB1**:
   * A new transaction is inserted into the transactions table in **DB1**:

INSERT INTO transactions (id, amount, timestamp)

VALUES (101, 100.50, '2023-10-01T12:34:56Z');

1. **Debezium Captures the Change**:
   * Debezium reads the change from the transaction log and converts it into a JSON message.
2. **Kafka Receives the Change**:
   * Debezium publishes the JSON message to the Kafka topic dbserver1.db1.transactions.
3. **Kafka Consumer Reads the Change**:
   * The Kafka consumer reads the JSON message from the Kafka topic.
4. **Change Applied to DB2**:
   * The consumer executes the corresponding SQL in **DB2**:

INSERT INTO transactions (id, amount, timestamp)

VALUES (101, 100.50, '2023-10-01T12:34:56Z');

**Summary**

* **Debezium** monitors **DB1**, captures changes, and sends them to **Kafka**.
* **Kafka** acts as the middleman, storing and forwarding the changes.
* A **Kafka consumer** reads the changes and applies them to **DB2**.
* This ensures **real-time synchronization** between **DB1** and **DB2** without manual intervention.