ontotext

A graph database is a type of database that stores data as nodes and relationships instead of tables. Nodes represent entities, while edges represent the relationships between these entities. Graph databases excel in handling complex and interconnected data and have become the go to solution for many enterprise challenges.

The benefits of a graph database

The reason graph databases are becoming more and more popular among all types of businesses and organizations is due to several factors.

Big Data: With the growth of big data, traditional relational databases have become less effective at handling complex and interconnected data. Graph databases provide a more efficient way to manage and analyze large datasets with many-to-many relationships.

Social networks: The explosion of social media and online communities has created a need for tools that can analyze complex networks of relationships. Graph databases provide a way to store and analyze these relationships, allowing companies to gain insights into how information flows through social networks.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

Fraud detection: Fraud detection is becoming increasingly important for companies across many industries. Graph databases can be used to detect fraudulent activity by analyzing the relationships between different entities, such as customers, transactions and accounts.

Machine learning: Graph databases are also gaining popularity in the field of machine learning. By using graph databases to store and analyze data, machine learning algorithms can gain insights into complex relationships and patterns in the data, leading to more accurate predictions and better decision-making.

However, integrating different data consumers and providers in a complex IT ecosystem can be a challenge. This is where Kafka comes in as it simplifies the ETL (Extract, Transform, Load) process by coordinating all the participants using a message bus.

In this blog post, we will explain how you can easily integrate a graph database into such an ecosystem using Kafka. We will also describe several different ways to import your on-premise data in Ontotext's RDF database for knowledge graphs GraphDB together with some examples how to do it.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

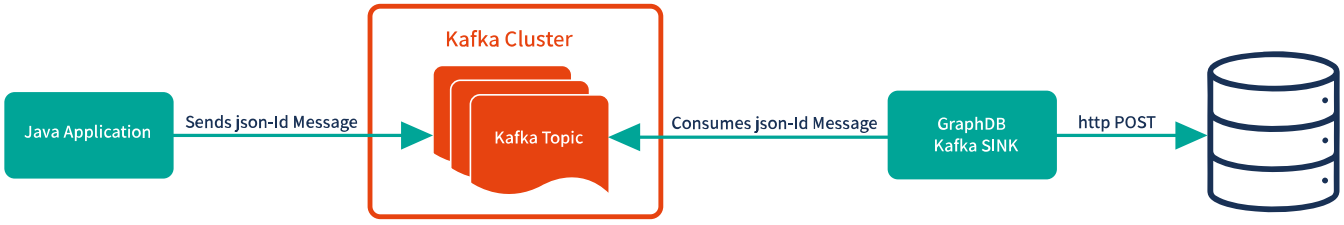
Ingesting data using the Kafka Sink Connect

Now let's say your organization decides to adopt a graph database. As a first step, you'll have to think of how to ingest some of the data that you already have into the empty database.

Kafka is a scalable, fault-tolerant system for processing and storing such data and can be used to reliably import data into GraphDB. It is designed to handle high volumes of data and can easily scale to handle peak loads without compromising on performance or data integrity. Kafka's reliability and fault-tolerance ensure that no message updates are missed or lost, making it a reliable and robust platform for real-time data processing. On top of that, Kafka's integration capabilities make it a flexible and versatile platform that can seamlessly integrate with other systems and tools, enabling users to build end-to-end processing pipelines that can handle complex data processing tasks.

Integrating Kafka and GraphDB is achieved through Kafka Connect, which is an ecosystem of connectors and an API allowing Apache Kafka to seamlessly integrate with various external systems like GraphDB. This integration does not require developers to write any additional code, thereby making the process more scalable and reliable. With Kafka Connect, data can be easily ingested and exported from Kafka topics, enabling a more streamlined and efficient data integration process.

GraphDB has its implementation of Kafka Sink Connect. The Kafka Sink Connector is a separate component that does the work for you. Once started, it listens to a topic in the Kafka broker and processes the messages from it. As you can see in the diagram below, each message that is consumed is processed and written using RDF4J to GraphDB (for more detailed information on the topic, go to the official GraphDB documentation and the open-source repo of the GraphDB Kafka Sink Connector.)



Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

The Kafka Connect Sink can process different data formats. Here, we use mostly JSON-LD but Kafka Sink supports a whole range of different data formats. The only thing that needs to be changed to work with TURTLE for example is to change the Kafka Sink Connector creation.

Starting the components

First you need to start GraphDB, Kafka and the Kafka Sink Connector as described here.

You can use the Docker compose given in the repository or the one specially created for this post – docker compose.

Alternatively, you can deploy the services separately. If you decide to deploy them separately, you need to start Kafka, GraphDB and GraphDB Kafka Sink Connect. For detailed configuration on Kafka Connect, you can refer to the official documentation or use the docker-compose file as a reference.

So, once everything is started, regardless of the chosen method, the following components should be running without errors in the logs:

- GraphDB
- Kafka and Zookeeper
- GraphDB Kafka Sink

The next step in the process is to create a Kafka Sink Connector for a specific topic, which will be responsible for transferring data from a Kafka topic to GraphDB. This connector will read messages from the specified topic and write them into GraphDB. A sample REST call can be found in the official GraphDB documentation. GraphDB supports 3 connector types for different operations – one for adding new data, one for graph replace and one for updating data.

Creating the connector

In this example, we will demonstrate the ADD and REPLACE_GRAPH operations, so we need two connectors. We will also use JSON-LD for importing the data. The following is the configuration of the connectors that we have used.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

Creating the ADD Connector

The first connector for the ADD operation will be created in the GraphDB Sink Connector running on port 8083. Here is an explanation about the connector configurations:

- In the calls below it is important to set a proper batch size in case we want to import a lot of data. In this example, we use batches of 1000 and around 1.2 Mil statements import for around 2-3 minutes on a desktop machine. This is the configuration: `graphdb.batch.size`.
- Another important configuration is `graphdb.update.rdf.format`, where we have to specify `jsonld` so this example can work.
- The `.graphdb.batch.commit.limit.ms` sets the transaction commit limit. If this is exceeded, the commit will be interrupted and the Kafka messages will not get acknowledged.
- `tasks.max` defines how many parallel tasks process the messages. By default it is one and we have included it here for completeness.
- The topics should contain the topic to which we subscribe in order to read the Kafka messages. Once this command is executed, the topic will be automatically created, so there is no need for additional configuration.
- `graphdb.auth.type` is used only in case Kafka is secured. In our example it is not, so we set it to NONE.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

```
curl -H 'Content-Type: application/json' --data '
                {
                        "name": "kafka-sink-add",
                        "config": {
                                "connector.class":"com.ontotext.kafka.GraphDBSinkConnector",
                                "key.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
                                "value.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
                                "value.converter.schemas.enable": "false",
                                "topics":"add-data",
                                "tasks.max":"1",
                                "offset.storage.file.filename": "/tmp/storage",
                                "graphdb.server.url": "http://graphdb:7200",
                                "graphdb.server.repository": "test",
                                "graphdb.batch.size": 1000,
                                "graphdb.batch.commit.limit.ms": 5000,
                                "graphdb.auth.type": "NONE",
                                "graphdb.update.type": "ADD",
                                "graphdb.update.rdf.format": "jsonld"
                        }
}' http://localhost:8083/connectors -w "\n"
```

Creating the REPLACE_GRAPH Connector

The second configuration is similar to the first one and only several properties are different. The operation specified in `graphdb.update.type` should be REPLACE_GRAPH. The `topics` should be different and, of course, the address of the GraphDB Kafka Sink should match your deployed connector. The rest of the configurations are very similar to the connector above.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

```
curl -H 'Content-Type: application/json' --data '
                {
                        "name": "sink-graphdb-replace"`
                        "config": {
                                "connector.class":"com.ontotext.kafka.GraphDBSinkConnector",
                                "key.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
                                "value.converter": "com.ontotext.kafka.convert.DirectRDFConverter",
                                "value.converter.schemas.enable": "false",
                                "topics":"replace-data",
                                "tasks.max":"1",
                                "offset.storage.file.filename": "/tmp/storage",
                                "graphdb.server.url": "http://graphdb:7200",
                                "graphdb.server.repository": "test",
                                "graphdb.batch.size": 100,
                                "graphdb.batch.commit.limit.ms": 5000,
                                "graphdb.auth.type": "NONE",
                                "graphdb.update.type": "REPLACE_GRAPH",
                                "graphdb.update.rdf.format": "jsonld"
                        }
}' http://localhost:8084/connectors -w "\n"
```

Once these configurations are successfully created, we are ready to ingest some data.

The Java application

We have provided a sample Java console application that uploads data to GraphDB using the newly created connectors. The sample application is located in this repository. To start the application, clone the repository and build the project using Maven. To modify the ingestion parameters, open the `application.properties` in the `src/main/resources` folder. Please, make sure that the configuration of the URLs for the Kafka broker and GraphDB are correct.

Adding new data

To test the ADD connector you need to run RunAddData.java. The implemented logic there is really simple – create a Kafka producer and write in a loop the JSON-LD files to the topic that is used by Kafka Connect. The snippet from the repository provided below shows the most important part of the logic. The only thing we need to do is:

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

- create a new producer
- create a new record
- send the record and flush the data to the topic

From there on GraphDB Kafka Sink takes over and inserts the data in GraphDB.

```
KafkaProducer<String, String> producer = new KafkaProducer<>(createConfiguration());

...

ProducerRecord<String, String> producerRecord = new
ProducerRecord<>(topicName, producedObject);

producer.send(producerRecord);

...

producer.flush();

producer.close();
```

Updating data using Graph Replace

To test the REPLACE_GRAPH, you need to start RunReplaceGraph. This scenario inserts a single JSON-LD in a named graph and then replaces the data in the graph. This example first uses the ADD topic to insert the data and then the REPLACE_GRAPH topic to update it.

The code snippet below shows:

- Creating a Kafka message for inserting new data in a named graph with specific IRI. This is basically the same scenario as the one for adding data.
- After the data is inserted, we create a new Kafka record with the same graph IRI but different data. Then we send the second object to replace the graph topic with a key, the IRI.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

ontotext

```
KafkaProducer<String, String> producer = new
KafkaProducer<>(createConfiguration());

. . .

String producedObject = JsonldBuilder.build(graphIRI);

ProducerRecord<String, String> insertRecord = new
ProducerRecord<>(addTopicName, producedObject);

producer.send(insertRecord);

producer.flush();

. . .

String replacedObject = JsonldBuilder.build(graphIRI);

ProducerRecord<String, String> updateRecord = new
ProducerRecord<>(replaceTopicName, graphIRI, replacedObject);

producer.send(updateRecord);

producer.flush();



producer.close();
```

Inserting Json-ld using Rdf4j

In certain situations, due to various factors, Apache Kafka and Connect can be seen as overhead.

First of all, the complexity of setting up Kafka can be a challenge. It involves configuring and managing multiple components such as brokers, producers, consumers and coordination tools like ZooKeeper. This process can be time-consuming and complex, especially for those with limited experience in distributed systems.

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

The second issue is that operating a Kafka cluster requires ongoing monitoring, capacity planning and ensuring high availability and fault tolerance. Tasks such as managing topics, partitions, replication and addressing potential issues like network failures and data rebalancing can add to the operational overhead, especially in smaller or resource-constrained environments.

Finally, working with Kafka may involve a steep learning curve. Developers and administrators need to familiarize themselves with Kafka's concepts, APIs and ecosystem tools to effectively utilize its capabilities. This learning curve can be particularly steep for those transitioning from traditional messaging systems or lacking experience with distributed streaming platforms.

This is why it's also possible to skip Kafka and directly ingest your organization's data in GraphDB using Rdf4j. This is a less reliable solution but in some cases, it could work better.

The `RunAddDataRDF4J` class demonstrates this scenario. It also uses batch processing and the ingestion speed is similar to GraphDB Kafka Sink.

To wrap it up

Using Kafka Sink for ingesting data in a graph database offers several benefits and enables efficient data integration. The Kafka Sink Connector acts as a bridge between Apache Kafka and your graph database, facilitating the seamless flow of data from Kafka topics to the graph database.

Need a reliable and robust RDF graph database for your use case?

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com

Author: Ivo Rusev

ontotext

Ontotext's GraphDB
Give it a try today!
Download Now

Copyright: Ontotext New York | Sofia | Basel
Reach out to us: info@ontotext.com