ontotext

In this post, we are going to focus on two features of GraphDB – Smart Updates and the Kafka Sink Connector – and how they can help you save time and effort when you stream updates to GraphDB.

First, we'll talk about different ways to manage RDF documents in GraphDB. Then, we'll explain what smart updates with SPARQL templates are. We'll also compare direct vs decentralized updates in GraphDB. And finally, we'll introduce the Kafka Sink Connector and its three update modes.

Managing RDF documents in GraphDB

Managing RDF documents in GraphDB (or in any other RDF database) is always a bit tricky. One of the great features of RDF databases is that they don't have a fixed schema but that same advantage creates some difficulties when it comes to updating the content of an RDF document. Namely, it's not easy to decide what defines a document and what part of the old content can be removed when inserting a new one.

So, people have resorted to two major strategies for managing documents in GraphDB. The first is to store each RDF document in a single named graph and the second – to store it as a collection of triples where multiple RDF documents exist in the same graph.

Updating RDF document per named graph

When we have a single RDF document per named graph, updating it is straightforward. You just have to delete the content of the named graph and then insert the updated RDF document into the same named graph. Here, GraphDB provides an internal optimization and if you do this in a single transaction, it will not delete the statements that didn't change between the old and the new version.

Updating RDF document as a collection of triples

Updating a document that is a collection of triples spanning in one or more named graphs (or also in the default graph), isn't as trivial, however. First, you have to remove the old content of the document. And here, many people think it's about a simple "delete all triples" operation where the subject is the ID of the document. But in practice this isn't the case

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

ontotext

because, very often, documents contain sub-documents and you want to manage the sub-documents together with the main documents. So you can't just delete the statements that have a given subject.

To update such a document, people typically use a handcrafted SPARQL template and it's usually a SPARQL DELETE WHERE updates. Or it could be a SPARQL DELETE/INSERT WHERE updates. The template deletes the content and then you can insert the new data. The drawback here is that this template must be the same on every system that executes updates. So, whenever you need to change the template, you need to go to every instance of the application. This is difficult to manage and it's also prone to errors.

Smart updates with SPARQL templates

Because of all these challenges, we have decided to introduce Smart Updates using SPARQL templates. The Smart Update is a way of defining a server-side SPARQL template you can use when you want to update a document. Every template corresponds to a single document type and defines the SPARQL update that needs to be executed in order to delete the old contents of the document. It can also include an INSERT part that can be used to generate new metadata triples. For example, this could be the exact timestamp when the document was updated or some other metadata that you want to generate based on the update event.

Executing updates with SPARQL templates

When you have such a template defined and you want to execute a Smart Update, first of all, you have to notify the system. This happens by using a special system INSERT that provides the IRI identifying the template and the IRI identifying the document. Then, the new content of the document can be added to the database in any of the supported ways – SPARQL INSERT, add statements, etc.

Why are Smart Updates an improvement? They are an improvement because they remove the burden of managing the handcrafted SPARQL in the individual applications or clients, and push it to the server. Now, if you have many applications that have to update the same type of document, you keep a single copy of this template on the server.

It also means that you get consistent templates across the whole system and that you can

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

manage the model together with the data. You can import the templates together with the data and if you do a backup of the repository, the templates will be backed up. If you restore an older version from backup, it will restore the templates as well. So, the template will always match the data that's in the repository.

Direct vs decentralized updates

Now, let's have a look at different ways to transport updates to GraphDB.

First of all, why are updates so important? They are important because business is hungry for data. And what makes data interesting is also what makes it difficult to manage. It often comes from multiple and diverse systems that provide it in different formats, which usually have to be transformed into another format. And another important thing about updates is that data often changes and you want to have the latest data in your repository.

Normally, applications that deal with RDF write directly to GraphDB and this has advantages and disadvantages. Some of the disadvantages are that you need to write verbose code specific to an application and a client (Java, Node.js, REST API, etc.). Another big drawback is that every update in GraphDB regardless of its size has to be a transaction and transactions are expensive. And when you have many small transactions, this can also slow down your database.

Why Kafka?

Over the recent years, we've thought about how we can improve all that and we decided that Kafka is the best choice.

First of all, Kafka is scalable and decentralized, which means that you can easily add more nodes to a Kafka cluster to make it bigger and able to handle more data. You don't have to think about how much more memory to give to GraphDB to make it work. Kafka is also very robust because, if a node goes down, there are more nodes to handle the data that you push to it.

But what makes Kafka really great is that it supports pipelines. You can push a message to Kafka and then a processor can take that message, perform some transformation on it (e.g.,

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

convert JSON to RDF), push it back to Kafka, and then another processor can take the RDF message and write it to GraphDB. Kafka is also easier to use in general because it has a much simpler interface. You just post messages that have a key and a value. Finally, you can schedule your updates to be executed in GraphDB at a convenient time – for example, you can batch together small updates in a single transaction to make it more efficient.

Kafka Sink Connector

To make it possible to use Kafka with GraphDB for decentralized updates, we created the Kafka Sink Connector.

The Kafka Sink Connector is a plugin for Kafka Connect. Here, it's important to note that the Kafka Sink Connector is a Kafka Connect connector and not a GraphDB connector. It's sometimes confusing because they are both called a connector and it's even more confusing because we have something called the Kafka GraphDB Connector, which is different.

The Kafka Sink Connector is open source and it's extensible if you want to customize something or do it in a different way.

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

ontotext

What it does is read RDF messages from a Kafka topic and then write these RDF messages in GraphDB. You can do this in three different update modes: Add, Replace Graph and Smart Update with SPARQL template.

Kafka Sink Connector: Add

The first mode is very simple. It allows you to add more data to GraphDB. In a way, it's an update but it doesn't update existing data. It just adds more data. For the Add mode configuration you need a Kafka topic to read from and the RDF format of the messages that will be read from the topic.

The way it works: the key of the message is irrelevant and the value is the data in the configured RDF format. This data will be read from the message and it will be written in GraphDB without any transformations.

Kafka Sink Connector: Replace Graph

The second update mode is the Replace Graph mode. To configure it, you need to provide again a Kafka topic to read from and the RDF format of the expected messages.

The way it works: the key of the message will be the document IRI to update and the value will be the new document contents in the configured RDF format. This corresponds directly to managing documents where one document is stored in one named graph. In this update mode, the named graph identified by the document IRI will be cleared and the RDF will be written into that named graph.

Kafka Sink Connector: Smart Update

The last update mode is the Smart Update using the SPARQL templates we already talked about. Unlike the previous two modes, in addition to the Kafka topic and the RDF format, here we also have to provide the template IRI that identifies the template we want to use.

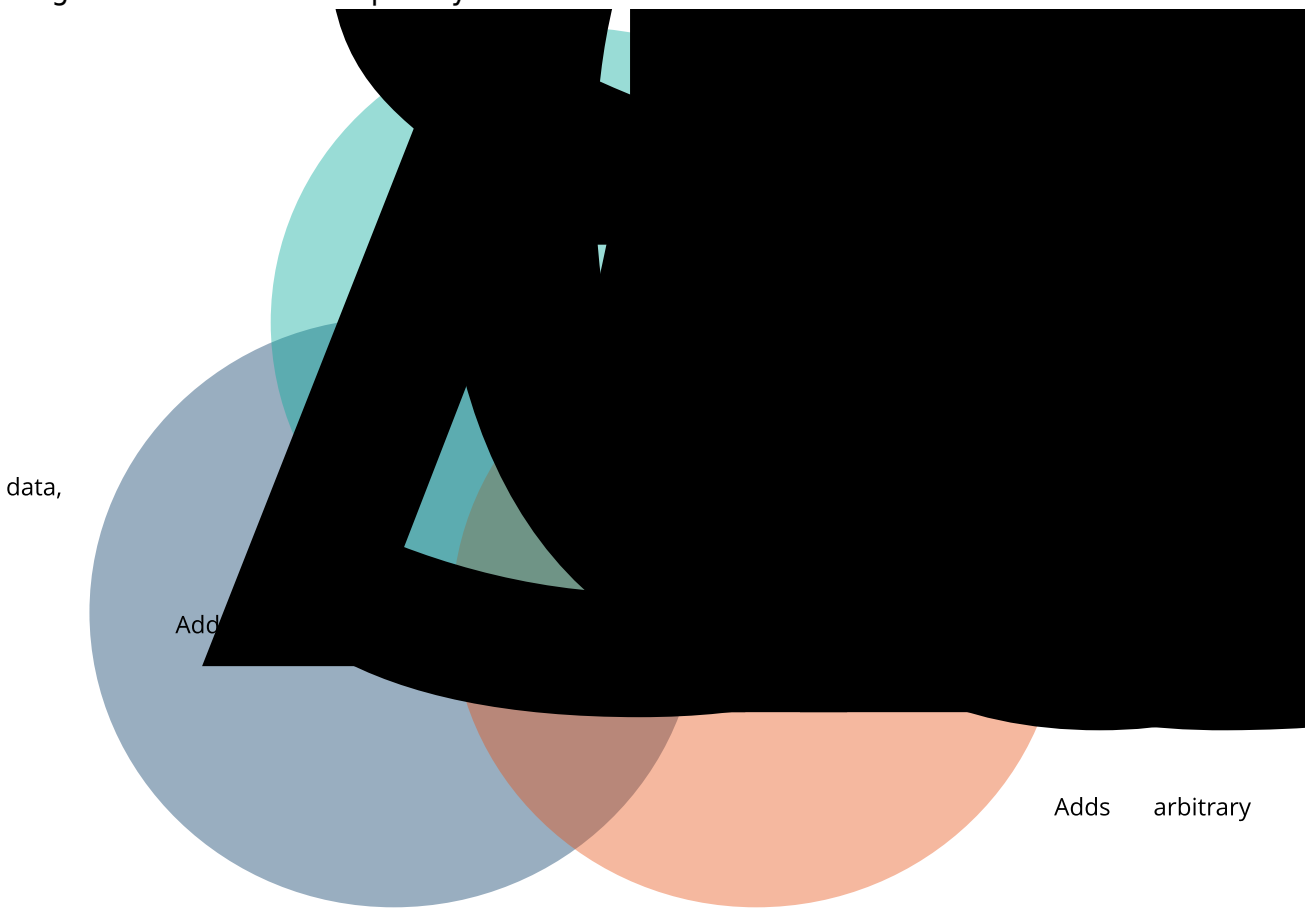
The way it works: the key of the message will be again the document IRI that we want to update and the value will be the new document contents in the configured RDF format. Once the message is read, the SPARQL template will be executed to clear the old contents of the

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

document and the new RDF will be written in GraphDB without any transformation.

Putting it all together

The diagram below should hopefully make it easier to understand.



In bold, we have the required configuration elements for each mode. All three modes require a Kafka topic to read from and the RDF format for the messages they expect. In addition, the Smart Update requires the SPARQL template IRI.

It’s easy to see that both Smart Update and Replace Graph modes can update a document by its IRI. The Replace Graph mode is more suitable for updating documents where one document is stored in one named graph whereas Smart Update is good for updating

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

documents where the documents can share named graphs with other documents.

You can see as well that the Add mode just adds arbitrary data to GraphDB and documents aren't relevant. Here, it's also important to note that both in Smart Update and Add modes, data will be written exactly as provided. This means that if your data contains named graphs (e.g., the TriG format allows you to specify named graphs), these named graphs will be kept and written in GraphDB. Whereas if you use the Replace Graph mode (where the data is written into a specific named graph), all the named graphs that come from the data will be lost even if they are provided there.

Wrap Up

If you are wondering whether using direct updates is better than using Kafka or vice versa, the answer is that they have different advantages and disadvantages depending on your specific use case.

If you use direct updates, the advantages are that you get immediate and synchronous updates. In other words, whenever you update something, as soon as you commit it, it's visible in GraphDB. However, these updates don't scale well for many small updates and there is no pre-processing. You have to send the RDF directly to GraphDB.

If you decide to go the Kafka way, the disadvantages are that updates are eventual and asynchronous. So, whenever you push something to Kafka, it will get to GraphDB and become visible at some point, but you can't be sure exactly when. On the upside, these updates will scale really well because the Kafka Sink connector can batch small updates together and you will be able to transform your data through the Kafka pipeline.

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