ontotext

When we design something, if we want it to be functional, it's good to involve the people who will use it. This idea is the premise of Christopher Alexander's book *A Pattern Language: Towns, Buildings, Construction*, which became very influential in both construction and computer science after its publication in 1977.
Alexander was a civil architect and he believed that some of the greatest places in the world were designed not by architects but by ordinary people. So, he simply observed how different people designed their homes and then tried to make a system out of it.

Data Integration Patterns with Ontotext's GraphDB

For years, I've been watching different users and data engineers who design knowledge graphs and I've seen seven main design patterns that can be applied in GraphDB:

1. Load
2. Extract Transform Load
3. Extract Load Transform
4. Extract Transform Load Transform
5. Upstream Replication
6. Downstream Replication

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

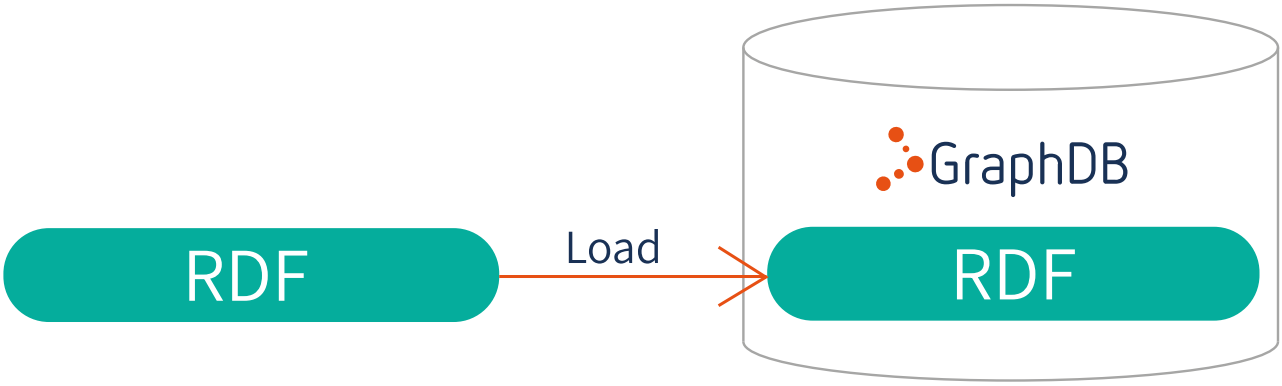
ontotext

7. Data Virtualization

Load

Context: Updates are already in RDF format
Problem: The batch needs to replace the old records
Solution: Choose one of the standard provenance models

The Load pattern is the most fundamental one controlling the synchronization of external RDF data with GraphDB. The challenge is how to replace your old records with new ones. The solution is choosing one of the standard provenance models.



Standard provenance models

Graph Replace is probably the most straightforward model. The <http://mySource> graph groups the provenance of all statements. The update is to drop and re-import the same graph data into a single atomic transaction.

```
CLEAR GRAPH <http://mysource>
INSERT DATA { GRAPH <http://mysource> { <newdata> } }
```

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

ontotext

In use cases when the named graph has other meanings or the granularity of the updates is smaller like on the business object level, the user can design an **explicit DELETE/INSERT template**. The simplified SPARQL query below describes how to delete the old data and insert the new one.

```
DELETE {
    ?s <http://value> ?oldValue .
} INSERT {
    ?s <http://value> ?newValue .
} WHERE {
    ?s ?p ?oldValue .
}
```

The third approach is **soft deletes and versioning,** where every resource contains the date of creation and optional date of deletion. The deletion of an object is to create a statement when it is deleted. The biggest advantage of the approach is that the user can query the database at any given past timestamp. Still, this is also a disadvantage because every time the user queries for information, the query should identify what's the point of time to query.

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

ontotext

```
INSERT DATA <http://object/v1>
   <http://deletedOn> "2022-09-25"^^xsd:dateTime

INSERT DATA
   <http://object/v2> <http://replaces> <http://object/v1>;
   <http://value> "New value";
   <http://createdOn> "2022-09-25"^^xsd:dateTime .
```

Trade-offs of the standard provenance models

Graph Replace is fast and simple to implement and we recommend it to people with batch updates. The only negative aspect here is that if you have billions of statements in a single named graph, the repository will be locked during long-running transactions.
To deal with this issue, GraphDB implements a smart Graph Replace optimization that helps you calculate the internal data and only shows you the newly added and removed statements.
The **explicit DELETE/INSERT template** provides fast updates, but the load depends on the schema. The GraphQL API addresses this issue on top of GraphDB, which can read the schema and generate such triples to update the data.
The **Soft Deletes and Versioning** has the benefit of keeping track of the full history of your data, but your repository will become extremely big. This pattern will work well only with data that is not deleted frequently and it will also require implementing versioning. All this makes the approach expensive and complex.

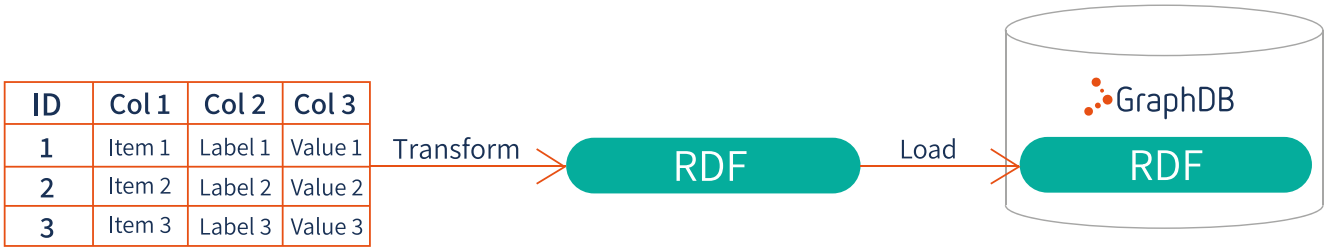
Extract Transform Load

Context: Updates are NOT in RDF format
Problem: Transform the structure into RDF schema and generate identifiers for the instances
Solutions: Reuse or generate persistent identifiers; generate non-persistent identifiers

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

ontotext

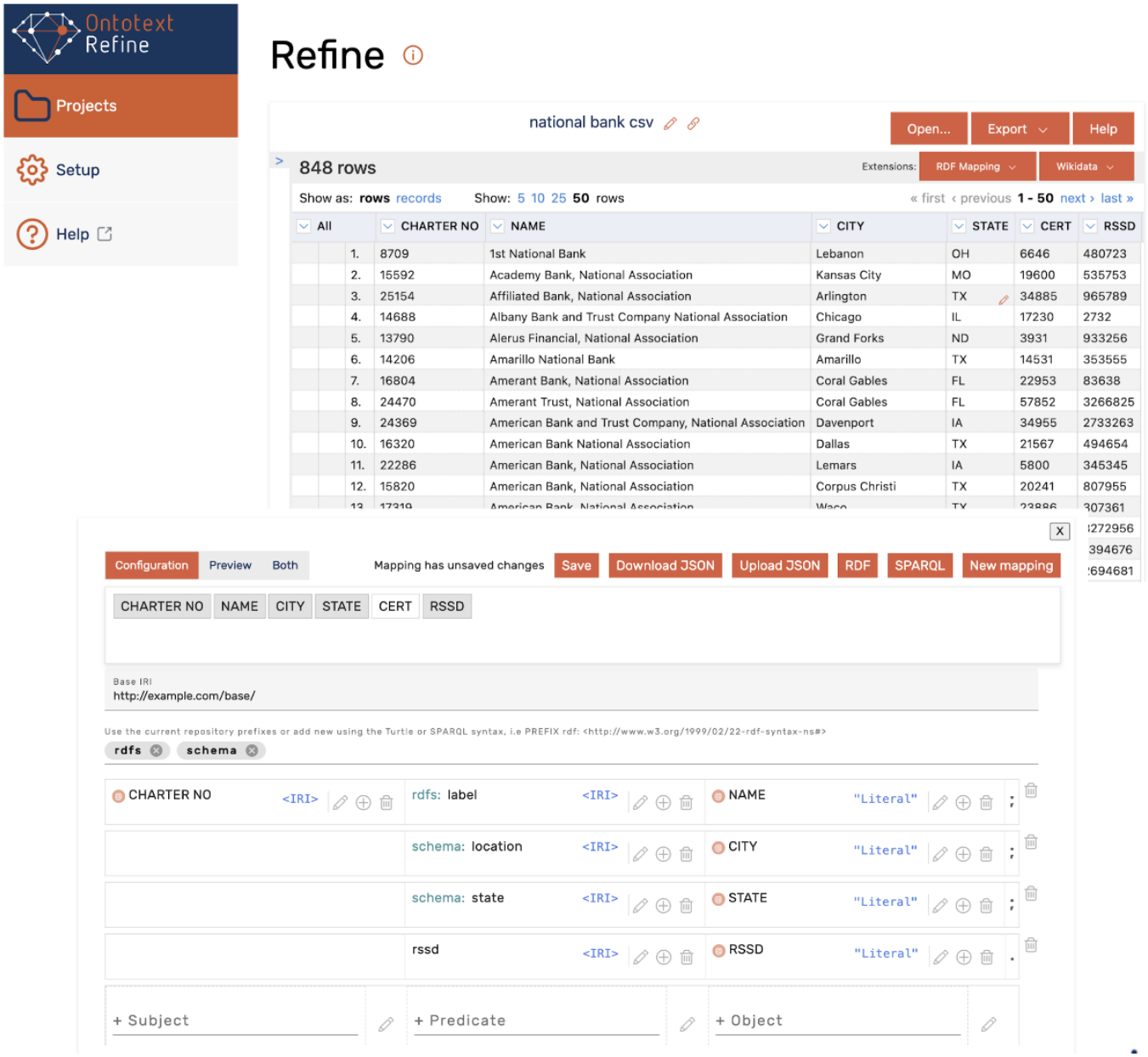
In the Extract Transform Load (ETL) design pattern, the updates are not in RDF but in a table format. The pattern solves two tasks – how to connect the tabular values into a graph and how to generate RDF values like IRIs or Literals. When generating IRIs, it's always important to consider whether they will be persistent or transient. For more details check the section below on persistent identifiers.



ETL tools and their trade-offs

Ontotext offers multiple solutions for creating knowledge graphs.
The first solution is with **Ontotext Refine** – a free tool with advanced data cleaning and transformation capabilities packaged into an intuitive user interface. The transformed table becomes a virtual SPARQL endpoint accessed over the federation. The approach is mainly used for fast prototyping since it's limited to an in-memory model practical for up to a few hundred million RDF triples.

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



The second approach is to use some **Data Integration Platform**. As an enterprise-supported tool, it has already established how to make all data transformations. Please note that your platform may not natively support RDF. Then the recommended approach is to use one of the many JSON to RDF transformation frameworks to produce RDF data.
Finally, the third and often the most common way is to write **custom code**. This approach

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

offers the benefit of using a custom convention of your own, but many things can go wrong, especially regarding maintainability and versioning.

Persistent or non-persistent IDs?

The first and the most important FAIR principle is to generate global persistent identifiers. This makes it possible for other users to reference the information without losing this link after an update. For example, in the screenshot below, you have Chicago as a string on the right side and an ID standing for Chicago on the left side. Using such an identifier where the identity of the city is not the label of the entity guarantees that even if this label changes in the future, the ID will still refer to the same object.

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

ontotext

Reconcile each cell to an entity of one of these types:

	Name	Count	Score
◉	city of the United States Q1093829	108	74.43
○	county seat Q62049	36	12.73
○	big city Q1549591	23	10.18
○	human Q5	19	14.33
○	town of the United States Q15127012	15	9.33
○	unincorporated community in the United States Q17343829	13	4.83
○	census-designated place Q498162	11	6.5
○	human settlement Q486972	9	2.79

Chicago (Q1297)

city and county seat of Cook County, Illinois, United States of America

Ideally, the data in the data source should come with persistent identifiers. However, quite

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

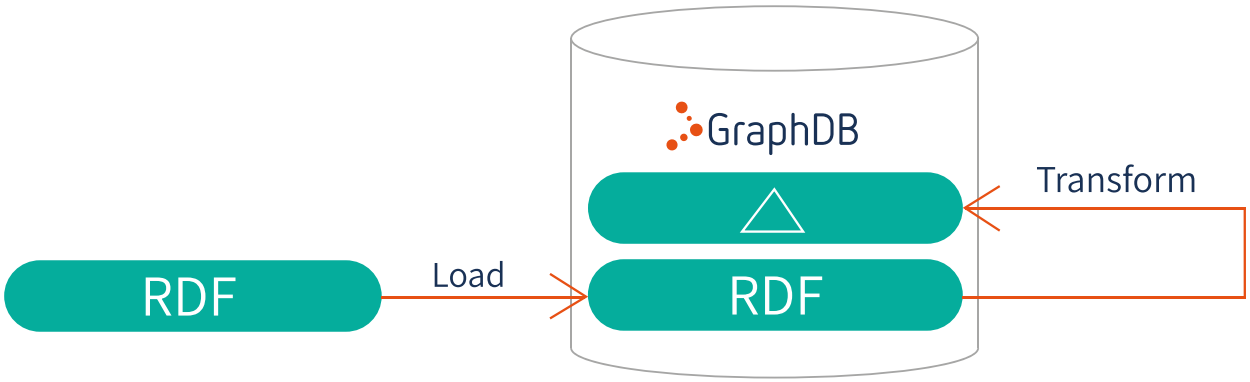
ontotext

often, either the data has been simply dumped in the database, or the identifiers are not reliable. In this case, you can either use the Ontotext Refine's reconciliation capabilities or implement custom entity resolution.
The alternative is to use non-persistent IDs, which will do the ID generation quite straightforward but will also prevent any other application to reference them reliably.

Extract Load Transform

Context: Updates are in RDF but schema/IDs differ from the final target
Problem: The alignment of the schema and/or the instances depends on other semantic data and cannot be computed externally
Solutions: Import all data in GraphDB and implement a sequence of transformations in the repository using semantic similarity, ontology alignment, or reconciliation as searches

The Extract Load Transform (ELT) pattern follows the same logic as ETL, but the data first enters into GraphDB and then undergoes a series of transformations with SPARQL queries to meet the final output requirements.
This pattern is useful when the transformation should consider other graph information to compute the final output. The solution is to Import all data in GraphDB and implement a sequence of transformations in the repository using semantic similarity, ontology alignment, or reconciliation as searches.



Choosing between ETL or ELT

Before deciding on ELT or ETL, you have to consider their advantages and disadvantages.

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

ontotext

Criteria	ETL	ELT
Transformation language	Any language	Chain of SPARQLs
Transformation code	Stateless	Stateless or Stateful
Processing type	Streaming or in-memory	In-memory
Persist each step	No	Yes
Performance	Fast	Slow
Scalability	High	Low
Exception handling	Established by the tool/language	Custom conventions

In ETL, the **transformation language** can be any language or data platform. In ELT, this is most likely implemented as a sequence of SPARQL queries to transform the data from one named graph to another named graph. This step may repeat many times, so the chain of SPARQLs should also keep the provenance of the information for easier debugging.

Extract Transform Load Transform

Combines both ETL and ELT
Applied for complex entity linking / record linking
Track provenance and the evolution of data

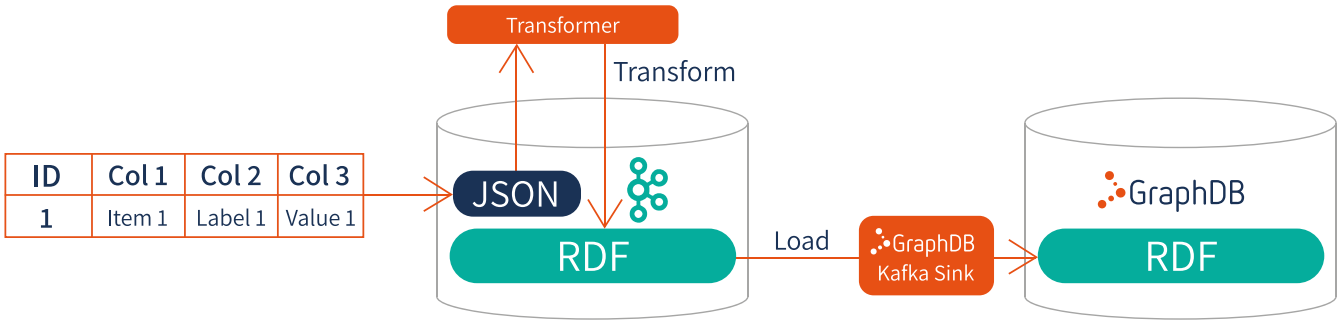
The final option of this batch series is the Extract Transform Load Transform, which is a composite pattern running on top of the previous two patterns. It can be applied to complex entity linking where the data comes in different formats. Here, you have to track the provenance and the evolution of the data clearly.

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

Upstream Replication

Context: Updates are executed not in batches but in a sequence of objects
Problem: Missing an update will break the data consistency
Solution: GraphDB Kafka Sink can automatically apply the updates

If the data is dynamically combined with high requirements for data freshness, then batching may not be an option. The upstream replication pattern persists each data change in a Kafka Stream to allow asynchronous updates to the database. The persistence in Kafka prevents you from losing updates due to service interruptions. The solution Ontotext recommends is to use GraphDB Kafka Sink, an automatic subscription mechanism that listens for data changes and applies them to one of the load patterns into GraphDB.



The main benefit here is that the infrastructure is already available and the only piece of code to implement is the transformer. Another benefit is that Kafka can buffer the extreme spikes of data changes reliably without losing any updates. Then the GraphDB Kafka Sync will consume all updates chronologically until it fully catches up.

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

ontotext

Downstream Replication

Context: Supply updates to a downstream system where batching is impossible
Problem: Select and filter the updates; never break the stream sequence
Solutions: Setup a Kafka Connector to propagate the updates

There is also a downstream replication, where GraphDB is the source of the information targeting another system. The task is to select only the part of the graph relevant to the downstream system and once again to not break the sequence of the updates.
The recommended solution is GraphDB's Kafka Connector, which listens for all data changes and fires notifications for the changed objects into Kafka.

Data virtualization

Context: Data lives in an external repository due to security or another management reason
Problem: Query non-semantic data with SPARQL queries
Solutions: Create a virtual repository and provide OBDA/R2RML descriptor to translate SPARQL to SQL queries at runtime

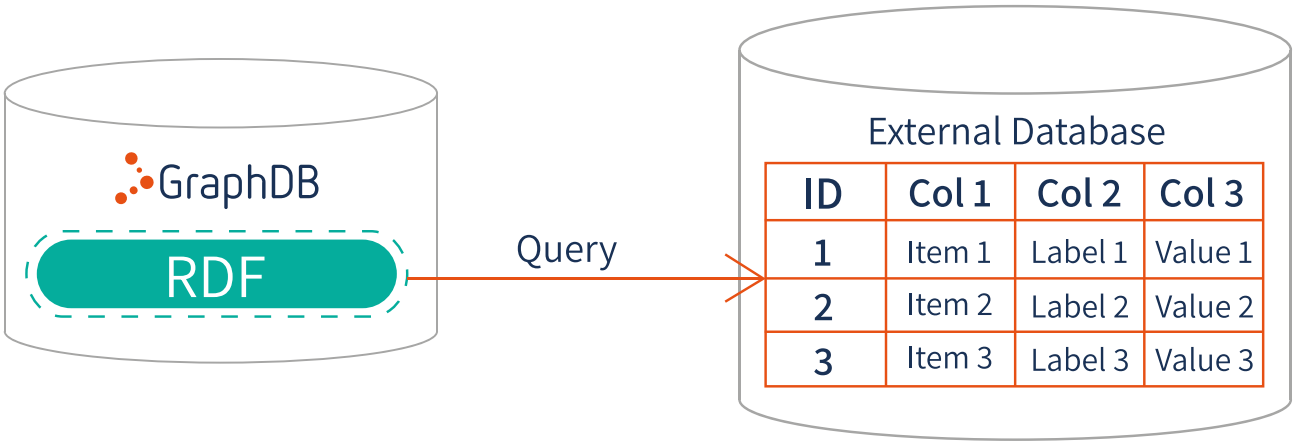
In data virtualization, the data lives in an external database. The challenge is how to make a virtual RDF graph that goes to the external database at query time, gets the information, and returns a valid SPARQL result.
GraphDB supports virtual repositories initialized with a SQL database endpoint and an OBDA or an R2RML descriptor to explain how the database tables are mapped to RDF. This functionality comes from the ONTOP project, which has support for 20+ different sources.
The pattern is convenient for dynamic data where it is not practical to transform it or in cases when due to legal resorts we can not copy it.

Advantages and Limitations of Data Virtualization

On the surface, it may seem advantageous to use data virtualization. For a start, graph and transaction data have no code integration. Also, every time you get the latest copy of the database, the queries are executed against the latest update. And there is no need for batch replication. So, from an operational point of view, it's easy to deploy.
But data virtualization brings many limitations. First of all, it is impossible to implement queries that are not supported by the underlying system. For instance, if the underlying

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

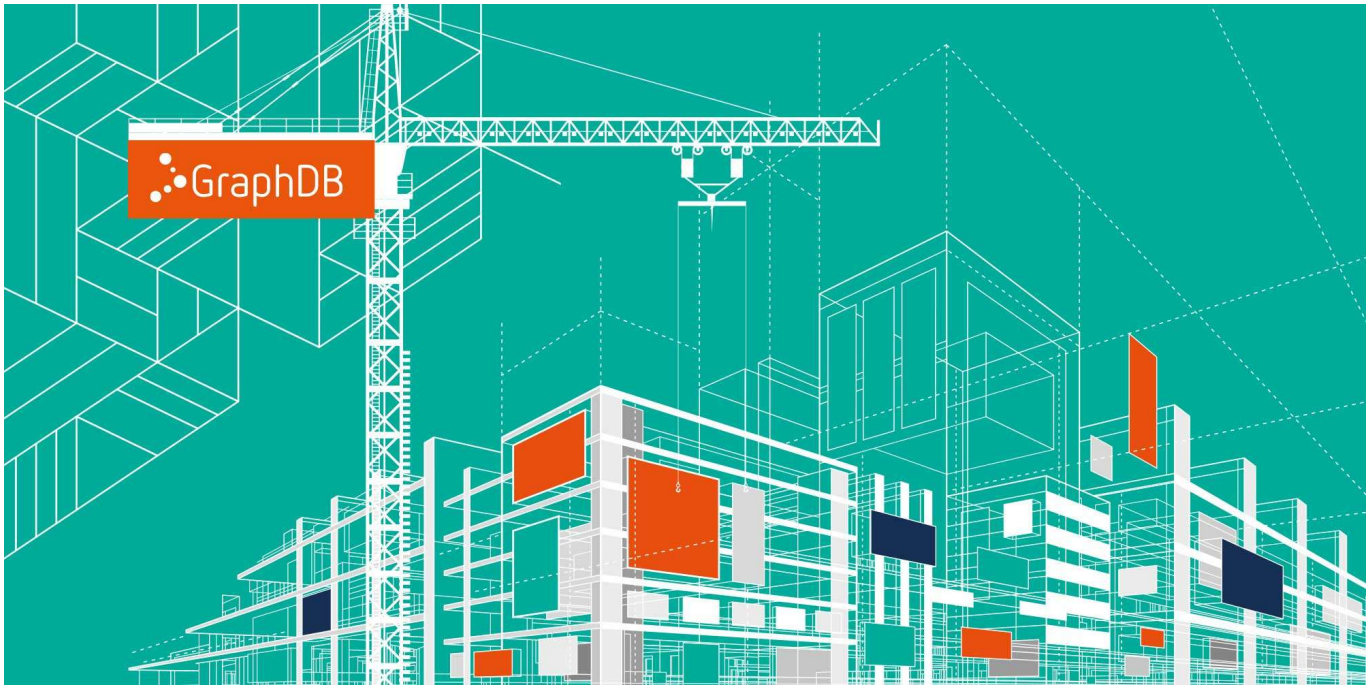
system has no full-text support or graph path search, GraphDB will also miss this functionality. Also, when the data is remote, any remote joins would pull all data locally to perform the join, which will have suboptimal performance. Overall, data virtualization works well only if the data is already normalized and have cross-database persistent identifiers, which makes it good for navigational exploration use cases.



To Sum It Up

Let’s have a quick summary of the seven data integration patterns again.
The Load integration pattern supports various provenance models and optimizes graph replace or single atomic INSERT/DELETE operations. This pattern is essential because this impacts all the other patterns.
For ETL, you can use Ontotext Refine or another tool to clear and transform structured data into RDF. It’s also critical to generate persistent identifiers to enable their reuse. For ELT, there is a big collection of GraphDB plugins for entity resolution, such as the Semantic Similarity and Search plugins.
You can use GraphDB Kafka Sink and the GraphDB Kafka Connector for the upstream and downstream replications. Finally, data virtualization is a powerful technique with severe limitations from the underlying systems.

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



The good news is that GraphDB also supports these patterns, allowing you to choose the one that best suits your needs.

Try the data integration pattern that's best for you!

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

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

Data Integration Patterns in Knowledge Graph Building with GraphDB

Author: Vassil Momtchev

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