Multidatabase

The Future of the data storage in Particle Physics and Astronomy

Julius Hřivnáč[1,*] and *Julien* Peloton[1]

[1]Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405 Orsay, France

Abstract. Particle Physics experiments and Astronomy telescopes store a large amount of data, most of those data are usually stored in simple files in various formats. Databases are used only to a limited extend, while database technology has made a tremendous progress in recent years, offering a large spectrum of applications in all possible domains. Those possibilities are still largely underused.
The article demonstrates the data architecture transparently using multiple database types, namely SQL database, NoSQL database and Graph database. Each database is used for the domain where it is the most appropriate and the cross-technology access is offered to users. The strong and weak points of all three technologies are discussed. Several possible ways of organizing interaction between databases are described.
The architecture is illustrated on the concrete implementation of the data storage of the Fink project, using JanusGraph and HBase technologies to store alerts from the Rubin Observatory. HBase is used to store the bulk data and the structural information is stored in Graphs, allowing transparent access to all data and deep analyses of relation between alerts and other objects.
Ways for implementing heterogeneous database architecture in general HEP applications are outlined.

1 HEP Data

In the High Energy Physics, the handling of data has long relied on conventional data structures. These include tuples, tables, and datagrams, as well as more complex structures like trees and nested tuples.

However, a significant portion of HEP data exhibits graph-like characteristics and lacks a rigid schema. These data often consist of entities interconnected through intricate relations, making them ill-suited for standard tree-ntuple storage or conventional relational databases.

To address this, a fundamental shift is required. Instead of having a fixed predefined schema or handling the relations externally, it becomes necessary to allow for a dynamic handling of relation between elements of data.

This approach, however, presents its own set of challenges. Traditional relational databases, which excel at managing well-defined relations, struggle when dealing with these dynamic, evolving relationships. Object-oriented (OO) databases and serialization methods also fall short in effectively managing such data, as they grapple with distinguishing essential relations from volatile ones.

*e-mail: Julius.Hrivnac@cern.ch

2 Databases

In the field of database management, three prominent paradigms exist: SQL, NoSQL, and Graph databases, each possessing distinct characteristics and advantages.

SQL databases are an obvious choice for managing well-structured, homogeneous data. They excel in ensuring data integrity and provide powerful querying capabilities. However, their strength lies in their rigidity, making them less adaptable to changing data schemas and less suited for handling unstructured or semi-structured data.

On the other hand, NoSQL databases offer flexibility and scalability. They excel in scenarios where data structures evolve, allowing for easy adaptation to changing requirements. However, this flexibility often comes at the cost of lacking a standardized query language or higher-level API, which can make complex querying and analysis more difficult.

Graph databases represent a distinct approach. They are designed to capture essential relationships within data. Graph databases are flexible, allowing for fast access to interconnected data, but they may have slower data injection processes. While they excel at representing complex relationships, they can introduce unnecessary overhead when dealing with straightforward data structures.

Storing data in a graph database typically involves organizing it into vertices and edges, denoted as $G = (V, E)$. Both vertices and edges can have associated properties. The concept of graph databases has existed for some time but has gained prominence more recently, especially with the advent of Big Data and AI, notably in the context of Graph Neural Networks. Today, there are well-established implementations and de-facto standards available, and the field is rapidly evolving.

2.1 Graph DB Performance

Requests in the context of data retrieval in Graph DB typically undergo three distinct phases:

- **Initial Entry Point Search:** The first phase involves the search for an initial entry point within the dataset. This phase has the potential for optimization, often benefiting from natural ordering, indexing, and technologies like Elasticsearch or Apache Spark to enhance the efficiency of this initial search.
- **Hierarchical Navigation:** This phase involves more hierarchical exploration of the data graph. It is characterized by very fast access to interconnected data, often facilitated by the graph database's capabilities. This stage allows for efficient traversal of relationships and exploration of related data nodes.
- **Accumulation of Results:** The final phase is concerned with accumulating and processing the results obtained during the navigation phase.

2.2 Graph DB Problems

While graph databases offer unique advantages, they also come with their own set of challenges and limitations that should be considered:

- **Slow Insert/Import:** One of the primary challenges in graph databases is the relatively slow speed of inserting or importing data. This can be particularly problematic when dealing with large volumes of data that need to be ingested into the graph.
- **Very Slow Cleaning (Memory Management):** Managing data efficiently in graph databases can be challenging, leading to slow cleaning processes. Clearing up unused or unnecessary data structures can be a time-consuming task.

- **Anarchical Edges Creation:** Allowing for the creation of arbitrary edges between nodes can sometimes result in the creation of unmanageable, overly complex graph structures. Without careful management, this can lead to a tangled web of relationships that are difficult to navigate.
- **Persistent and Volatile Data Mixing:** Graph databases may struggle with effectively separating persistent data from volatile data. This can impact data organization and make it challenging to maintain historical records while accommodating real-time updates.
- **Limited Mass Parallel Processing:** Graph databases may not be the optimal choice for mass, parallel processing of huge homogeneous tables. Other database systems, such as columnar databases, might be more efficient for such use cases.
- **Unknown Schema and Chaotic Relations:** Graph databases excel in handling flexible and evolving schemas, but this can also introduce complexity, especially when dealing with unknown or chaotic data relationships.
- **Advanced Query Languages:** Advanced query languages like Gremlin[1], which are commonly used with graph databases, can be cryptic and require a steep learning curve. Their multi-dimensional functional syntax may not be intuitive for all users, necessitating specialized expertise.

3 Hybrid Solution

In pursuit of an effective and versatile data management approach, a hybrid solution emerges, offering the best of all worlds by combining the strengths of different storage paradigms.

The hybrid solution starts by storing unstructured or raw data in traditional tables, such as SQL or NoSQL databases. These systems are chosen for their suitability for intensive, parallel processing, making them well-suited for data-intensive operations, including those involving technologies like Spark for distributed data processing.

The data stored in the table-like structures can be organized and accessed in a way that resembles the simplicity and interpretability of datagram-like APIs. This means that users can access and manipulate the data with relative ease, as if they were working with straightforward data packages or messages.

While the raw data remains in table-like storage, the hybrid solution introduces the concept of a graph to express and manage persistent data structures. This graph represents complex relationships and connections within the data.

The hybrid solution allows for the addition of ad-hoc, a priori volatile graph relations. These dynamic relationships can be established and modified as needed, providing flexibility to adapt to changing data requirements. These relations may even exist in separate but interconnected graphs, serving as "playgrounds" or "whiteboards" for experimentation and exploration.

Crucially, the hybrid solution connects all of these components behind a common API. This unified interface allows users to interact seamlessly with both the structured, table-like data and the dynamic, graph-based relationships. It abstracts away the underlying complexities, providing a user-friendly experience.

In essence, this hybrid approach offers a versatile way to manage data. It leverages the efficiency of table-like storage for raw data and the expressive power of graphs for representing and exploring complex relationships. By connecting these elements with a common API, it enables users to navigate, query, and analyze data in a manner that suits their specific needs, whether for structured, persistent data or for dynamic, experimental scenarios.

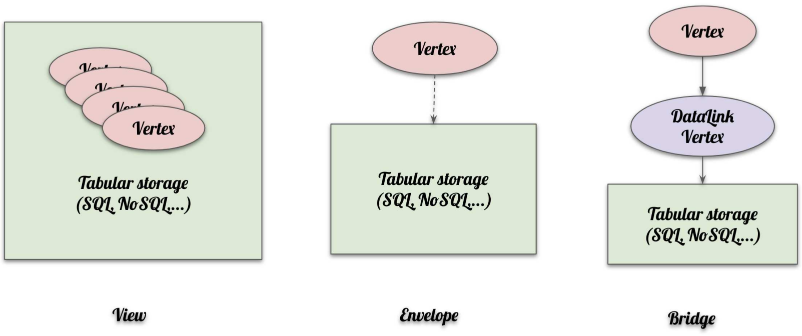
Vertex
Tabular storage (SQL, NoSQL,...)
Vertex
Tabular storage (SQL, NoSQL,...)
Vertex
DataLink Vertex
Tabular storage (SQL, NoSQL,...)
View
Envelope
Bridge

Figure 1. Three architectures for connecting tabular and graph database into hybrid solution.

3.1 Graph View

The "Graph View" approach involves interpreting existing tabular data as vertices within a graph. Additional edges are then added to the graph to express structural relationships between these vertices. However, implementing this approach can be challenging. To effectively implement the Graph View, a full-featured and rather generic implementation of a graph storage system is required. Developing this kind of system can be complex as it involves translating the existing tabular data into a graph format and ensuring the correct representation of relationships.

It's worth noting that while most graph database implementations use tabular storage as a backend, they often impose their own schema and mechanisms for handling graph data.

3.2 Graph Envelope

The "Graph Envelope" approach involves enhancing the concept of a vertex by adding additional methods to fill it from external tabular storage. This approach has been implemented in some cases. However, it comes with its own set of challenges.

Maintaining consistency between the data stored within the enhanced vertex and the original data in external tabular storage can be challenging. Changes made to one may not be immediately reflected in the other. Determining the search semantics, such as how queries are routed and executed, can be complex. It may lead to unpredictable performance and behavior. Users may not always be aware of where the data actually resides and whether it will be copied into the enhanced vertex or accessed remotely. This can impact performance and data management decisions.

```
// *** An example of the Graph Envelope approach
// Create an alert vertex
v = g.addV().property('lbl', 'alert')
// Dress it as (a subtype of) Hertex (= HBase backed Vertex)
// which _is_a_ Vertex so it has all Vertex properties
h = Hertex.enhance(v)
```

3.3 Bridge

The "Bridge" approach involves creating a special kind of DataLink Vertex that represents relations to external data stored in any type of storage system. These DataLink Vertexes can be attached to any other vertex in the graph, effectively forming bridges to external data sources. This approach offers several advantages.

The Bridge approach is relatively easy to implement compared to the other two approaches. It doesn't require complex data transformations or schema changes. It provides transparent logic for connecting data from different sources. Users can work with the graph as if all data were integrated seamlessly. It can work between any pair of databases with any technology, including connecting to other graph databases, making it highly versatile.

```
// *** An example of the Graph Bridge approach
// Create DataLink Vertexes with associated data in another database
// (Phoenix/SQL, Graph, HBase,...)
w = g.addV().property('lbl', 'datalink').
             property('technology', 'HBase').
             property('url', '134.158.74.54:2183:ztf:schema').
             property('query', "...")
// Connect DataLink to any Vertex
theVertex.addEdge('externalData').to(w)
// Get associated data
externalData = Lomikel.getDataLink(w)
```

4 Real-life example - Fink/LSST

4.1 LSST

The Vera C. Rubin Observatory, home to the Legacy Survey of Space and Time (LSST) in Chili employs an 8.4-meter telescope and a camera with 3.2 gigapixels resolution. It will generate 10 million alerts every night, which equates to approximately 1 terabyte of alert data with around 20 terabytes of image data. Over the course of 10 years, LSST is expected to accumulate about 60 petabytes of data and approximately 3 petabytes of alerts data. [2]

The alerts produced by LSST are disseminated globally through a network of 'brokers.' These brokers play a crucial role in distributing and managing the vast amount of alert data generated by the observatory.

LSST's operation is expected to start in 2024.

4.2 Fink

The Fink is recognized as one of the official brokers within the Rubin Observatory[3]. While this observatory is under construction, the Fink broker is receiving and analyzing the data from the Zwicky Transient Facility (ZTF)[4] which acts as a pathfinder by using similar technology to produce alerts. Since 2019, the ZTF survey is sending on average 200,000 alerts per night, and Fink is processing and redistributing these alerts in real-time.

4.3 Alert Data in Fink

In the Fink project, all incoming alert data are systematically stored in HBase[5] tables.

The structure of the alert data is created and managed in JanusGraph[6], a powerful graph database. This structure contains not only the relation to the core data but also the most critical attributes associated with each alert.

An essential aspect of data management in Fink storage architecture is the presence of datalinks that connect the data in JanusGraph to the corresponding data in HBase. These datalinks serve as bridges between the structured, graph-oriented data and the raw, tabular data stored in HBase, allowing for seamless access and retrieval of information.

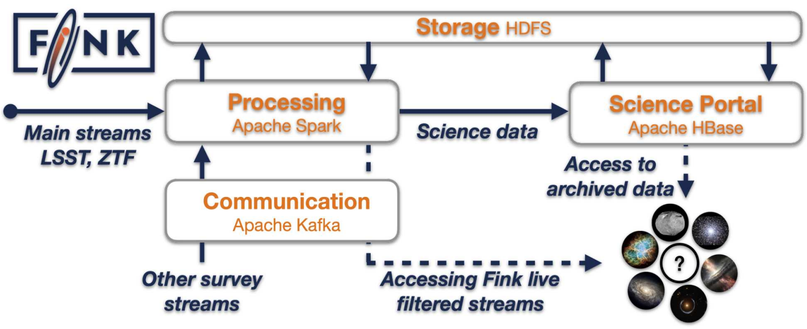

Figure 2. The architecture of the Fink broker, taken from [3].

Overall, the combination of HBase, JanusGraph, and datalinks forms a robust data management system within Fink, enabling efficient storage, organization, and retrieval of alert data.

```
// *** An example of accessimg Fink data in hybrid storage
// Get data (from HBase) attached in 'candidate's
g.V().has('lbl',      'source'      ).
      has('objectId', 'ZTF18abimyys').out().
      has('lbl',      'alert'       ).out().
      has('lbl',      'candidate'   ).out().
      has('lbl',      'datalink'    ).
      each {println(FinkBrowser.getDataLink(it))}
```

4.4 Analyses Example

```
// *** Search for interesting relations and store them in Graph as Edges for later analys
// *** Do it in your private subgraph.
// Create a new personal Graph.
g1 = Lomikel.myGraph().traversal()
// GremlinRecipies is a class with various useful Gremlin methods.
gr = new GremlinRecipies(g)
// Clone 'source' Vertexes in the private Graph 'g1'.
g.V().has('lbl', 'source').each {source -> gr.gimme(source, g1, -1, -1)}
// Get GremlinRecipies for the private graph 'g1'.
gr1 = new GremlinRecipies(g1)
// Find all pairs of 'candidate' Vertexes, where difference between their 'rb'
// fields is bigger or equal to 0.01.
// Connect them with the Edge 'distance' having a 'difference' property equal to
// the difference between 'rt' fields.
gr1.structurise(g1.V().has('lbl', 'candidate'), 'rb[0]-rb[1]', 0.01,
                                                 'distance', 'difference', ...)
// Get some statistics about newly created Edges.
g1.E().hasLabel('distance').values('difference').union(min(), max(), sum(), mean())
```

5 Usage

5.1 Graph Databases for Functional Programming

Graph databases offer a unique perspective on data modeling and processing that aligns well with functional programming paradigms.

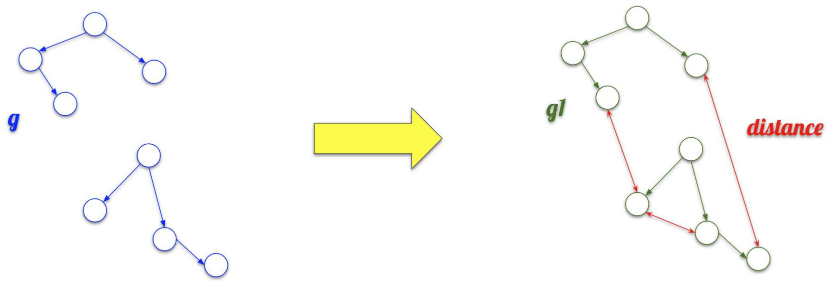

Figure 3. Adding new (distance) edges into an existing graph.

In a graph database, relations or edges between vertices can be thought of as functions. These functions define the connections and interactions between data points. This analogy allows for the application of functional programming principles to graph data. When navigating a graph, moving from one vertex to another can be seen as executing a function that follows a specific edge.

Functional programs can themselves be modeled as graphs, with functions as edges between data. This representation provides a clear visual understanding of program logic and can facilitate optimization and analysis.

Graph databases inherently support parallel access to data. Functional programming, with its focus on immutability and purity, can easily take advantage of parallel processing capabilities. This synergy enables efficient parallel access and processing of graph data.

Gremlin[1], a query language and traversal language used in many graph databases, is particularly well-implemented for combining functional programming and graph navigation. Gremlin provides a functional syntax for querying and traversing graph data, making it a powerful tool for graph-oriented functional programming.

5.2 Graph Databases for Deep Learning

Graph Database is a suitable storage for Deep Learning:

Neural networks, at their core, are graph structures. They consist of interconnected layers of neurons, forming a directed acyclic graph where each neuron is a vertex, and connections are edges.

Many real-world problems involve data represented as graphs, where entities (verteces) are connected by relationships (edges). Neural networks can handle such graph-structured data, which can include objects with complex relations. This capability extends to tasks that focus on individual nodes (node-centric tasks) or analyze the entire graph as a whole (graph-centric tasks).

Graph databases provide a structured way to represent domain-specific knowledge and constraints. This knowledge can be seamlessly integrated into GNNs, allowing them to incorporate inductive biases and semantic information, thereby improving learning and inference.

6 Graph and Hybrid Databases for HEP

The article highlights the importance of structuring data effectively in the context of High Energy Physics (HEP) and the advantages of using graph databases and hybrid storage solutions.

Graph databases offer several advantages:

- **Transparency:** They make code more transparent by representing complex relationships of data in an intuitive manner.
- **Stable Data Structure:** The storage layer handles data structure, ensuring data stability and consistency.
- **Functional Style and Parallelism:** Graph databases are suitable for functional programming and parallel processing, aligning with modern data processing techniques.
- **Deep Learning:** They are well-suited for deep learning tasks, especially when dealing with graph-structured data.
- **Declarative Analyses:** Graph databases facilitate declarative analyses, where queries describe what to retrieve, making it easier to understand and process data.
- **Analysis Preservation:** They can help preserve data analysis workflows, ensuring reproducibility.
- **Language and Framework Neutrality:** Graph databases are often language and framework-neutral, allowing flexibility in integration.

Hybrid storage solutions combine the expressiveness and flexibility of graph databases with the performance and simplicity of tabular storage, providing the best of both worlds. A transparent interface makes it easier to work with hybrid storage, abstracting away complexities.

Graph and hybrid databases can improve how data is handled in High Energy Physics experiments. These databases make it easier to organize and understand complex relationships within the data. By combining the strengths of multiple types of databases, researchers can manage data more efficiently.A real-life example using the Fink project demonstrates the benefits of using these database technologies. Described approaches could enhance data management in HEP experiments.

References

[1] *Gremlin*, `https://tinkerpop.apache.org/gremlin.html`

[2] LSST Science Collaboration, P.A. Abell, J. Allison, Anderson, ArXiv e-prints arXiv:0912.0201 (2009), `0912.0201`

[3] A. Möller, J. Peloton, E.E.O. Ishida, C. Arnault, E. Bachelet, T. Blaineau, D. Boutigny, A. Chauhan, E. Gangler, F. Hernandez et al., Monthly Notices of the Royal Astronomical Society **501**, 3272 (2020)

[4] *Zwicky transient facility (ztf)*, `https://www.ztf.caltech.edu`

[5] *Apache hbase*, `https://hbase.apache.org`

[6] *Janusgraph*, `https://janusgraph.org`